# DSA LAB PROGRAMS

1.a) Define a structure called Student with the members: Name, Reg_no, marks in 3 tests and average_ marks. Develop a menu driven program to perform the following by writing separate function for each operation:
a) read information of N students
b) display student's information
c) to calculate the average of best two test marks of each student.
Note: Allocate memory dynamically and illustrate the use of pointer to an array of structure.

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

// Define global variables
int i, j;

// Define a structure to represent a student
struct Student {
    char name[50];
    int regno;
    int marks[3];
    float avgmarks;
};

// Function to read student information
void read(struct Student *ptr, int n) {
    for (i = 0; i < n; i++) {
        printf("\nEnter details of student %d\n", i + 1);
        printf("Name : \t");
        scanf("%49s", (ptr + i)->name);
        printf("Register number : \t");
        scanf("%d", &(ptr + i)->regno);
        for (int j = 0; j < 3; j++) {
            printf("Marks in test %d :\t", j + 1);
            scanf("%d", &(ptr + i)->marks[j]);
        }
    }
}

// Function to calculate average marks of best two tests
void calculateAverage(struct Student *ptr, int n) {
    int a, b, c;
    for (i = 0; i < n; i++) {
        a = (ptr + i)->marks[0];
        b = (ptr + i)->marks[1];
        c = (ptr + i)->marks[2];
        float avg;
        if (a >= b && a >= c) {
            avg = (a + (b > c ? b : c)) / 2.0;
        } else if (b >= a && b >= c) {
            avg = (b + (a > c ? a : c)) / 2.0;
        } else {
            avg = (c + (a > b ? a : b)) / 2.0;
        }
        (ptr + i)->avgmarks = avg;
    }
}

// Function to display student information
void display(struct Student *ptr, int n) {
    calculateAverage(ptr, n);
    for (i = 0; i < n; i++) {
        printf("\nDetails of Student %d\n", (i + 1));
        printf("Name : %s\n", (ptr + i)->name);
        printf("Registration number : %d\n", (ptr + i)->regno);
        for (j = 0; j < 3; j++) {
            printf("Marks in test %d is %d\n", j + 1, (ptr + i)->marks[j]);
        }
        printf("Average best two test marks of %s is %.2f :\n", (ptr + i)->name, (ptr +
i)->avgmarks);
    }
}

int main() {
    int n, choice;
    printf("Enter the number of students :\t");
    scanf("%d", &n);
    struct Student *p;
    p = (struct Student *)malloc(n * sizeof(struct Student));
    if (p == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    do {
        printf("\nMenu :\n");
        printf("1. Read student Information \n");
        printf("2. Display student Information \n");
        printf("3. Calculate Average Marks of best 2 tests\n");
        printf("4. Exit \n\n");
        printf("Enter your choice\n");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                read(p, n);
                break;
            case 2:
                display(p, n);
                break;
            case 3:
                calculateAverage(p, n);
                for(i = 0; i < n; i++) {
                    printf("Average best two test marks of %s is %.2f :\n", (p+i)->name,
(p+i)->avgmarks);
                }
                break;
            case 4:
                printf("Exiting the program\n");
                free(p);
                return 0;
            default:
                printf("Please enter a correct choice\n");
                break;
        }
    } while (choice != 4);
}
```

1.b) Define a structure called Time containing 3 integer members (Hour, Minute, Second). Develop a menu driven program to perform the following by writing separate function for each operation.
a) Read (T) :To read time
b) Display (T):To display time
c) update(T):To Update time
d) Add (T1, T2) : Add two time variables.
Update function increments the time by one second and returns the new time (if the increment results in 60 seconds, then the second member is set to zero and minute member is incremented by one. If the result is 60 minutes, the minute member is set to zero and the hour member is incremented by one. Finally, when the hour becomes 24, Time should be reset to zero. While adding two time variables, normalize the resultant time value as in the case of update function.
Note: Illustrate the use of pointer to pass time variable to different functions.

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

// Define structure to represent time
struct time {
    int hour;
    int minute;
    int second;
};

// Function to read time from user input
struct time readTime(struct time *t) {
    printf("Enter hour, minute, and second (separated by spaces): ");
```

```c
    scanf("%d %d %d", &(t->hour), &(t->minute), &(t->second));
    return *t;
}

// Function to display time
void displayTime(struct time *t) {
    printf("%02d:%02d:%02d\n", t->hour, t->minute, t->second);
}

// Function to update time by adding one second
void updateTime(struct time *t) {
    t->second += 1;
    if (t->second > 59) {
        t->second = 0;
        t->minute += 1;
        if (t->minute > 59) {
            t->minute = 0;
            t->hour += 1;
            if (t->hour == 24) {
                t->hour = 0;
            }
        }
    }
    printf("The updated time is: ");
    displayTime(t);
}

// Function to add two times
void addTimes(struct time t1, struct time t2) {
    struct time sum;
    sum.second = t1.second + t2.second;
    sum.minute = t1.minute + t2.minute;
    sum.hour = t1.hour + t2.hour;
    if (sum.second > 59) {
        sum.second -= 60;
        sum.minute++;
    }
    if (sum.minute > 59) {
        sum.minute -= 60;
        sum.hour++;
    }
    if (sum.hour > 23) {
        sum.hour -= 24;
    }
    printf("\nResult of adding Time 1 and Time 2: ");
    displayTime(&sum);
}

int main() {
    struct time *t, t1, t2;
    int choice;
    t = (struct time *)malloc(sizeof(struct time));

    do {
        printf("\nMenu:\n");
        printf("1. Read Time\n");
        printf("2. Display Time\n");
        printf("3. Update Time\n");
        printf("4. Add Two Times\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                *t = readTime(t);
                break;
            case 2:
                displayTime(t);
                break;
            case 3:
                updateTime(t);
                break;
            case 4:
                printf("Enter time 1:\n");
                t1 = readTime(&t1);
                printf("Enter time 2:\n");
                t2 = readTime(&t2);
                addTimes(t1, t2);
                break;
            case 5:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 5);

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

2. Develop a menu driven program to implement the following operations on an array of integers with dynamic memory allocation. Display the array contents after each operation.

i) Insert by position

ii) Delete by key

iii) Search by position

iv) Reverse the contents.

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

// Function prototypes
void insertByPosition(int *arr, int *n, int pos, int key);
void deleteByKey(int *arr, int *n, int key);
void searchByPosition(int *arr, int *n, int pos);
void reverse(int *arr, int *n);
void printArray(int *arr, int *n);

// Function to insert an element at a specified position
void insertByPosition(int *arr, int *n, int pos, int key) {
    if (pos < 0 || pos > *n + 1) {
        printf("Invalid position\n");
        return;
    }

    (*n)++;

    for (int i = *n - 1; i > pos - 1; i--) {
        arr[i] = arr[i - 1];
    }

    arr[pos - 1] = key;
    printf("Element %d inserted at position %d\n", key, pos);
}

// Function to delete an element by its value
void deleteByKey(int *arr, int *n, int key) {
    int found = 0;

    for (int i = 0; i < *n; i++) {
        if (arr[i] == key) {
            found = 1;

            for (int j = i; j < *n - 1; j++) {
                arr[j] = arr[j + 1];
            }

            (*n)--;
            printf("Element %d deleted\n", key);
            break;
        }
    }

    if (found == 0) {
        printf("Key not found\n");
    }
}

// Function to search for an element by its position
void searchByPosition(int *arr, int *n, int pos) {
    if (pos >= 1 && pos <= *n) {
        printf("Element at position %d is %d\n", pos, arr[pos - 1]);
    } else {
        printf("Invalid position\n");
    }
}

// Function to reverse the elements of the array
void reverse(int *arr, int *n) {
    for (int i = 0; i < *n / 2; i++) {
        int temp = arr[i];
        arr[i] = arr[*n - i - 1];
        arr[*n - i - 1] = temp;
    }

    printf("Array reversed\n");
}

// Function to print the elements of the array
void printArray(int *arr, int *n) {
    printf("Current array: ");

    for (int i = 0; i < *n; i++) {
        printf("%d ", arr[i]);
    }

    printf("\n");
}

int main() {
    int *arr, n, c, choice;
    int pos, key;

    printf("Enter the capacity and size of the array: ");
    scanf("%d %d", &c, &n);

    arr = (int *)malloc(c * sizeof(int));

    printf("Enter %d elements of the array:\n", n);

    for (int i = 0; i < n; i++) {
        scanf("%d", (arr + i));
    }

    do {
        printf("\nMenu: \n");
        printf("1. Insert by position.\n");
        printf("2. Delete by Key.\n");
        printf("3. Search By Position.\n");
        printf("4. Reverse the content.\n");
        printf("5. Exit\n\n");
        printf("Enter your choice:\t");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the position and element to insert: ");
                scanf("%d %d", &pos, &key);
                insertByPosition(arr, &n, pos, key);
                break;
            case 2:
                printf("Enter the key element to delete: ");
                scanf("%d", &key);
                deleteByKey(arr, &n, key);
                break;
            case 3:
                printf("Enter the position to search: ");
                scanf("%d", &pos);
                searchByPosition(arr, &n, pos);
                break;
            case 4:
                printf("Reversing the array...\n");
                reverse(arr, &n);
                break;
            case 5:
                printf("Exiting the program\n");
                free(arr);
                return 0;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }

        printArray(arr, &n);
    } while (choice != 5);

    return 0;
}
```

3. Develop a menu driven program to implement the following operations on an array of integers with dynamic memory allocation. Display the array contents after each operation.

i) Insert by order

ii) Delete by position

iii) Search by key

iv) Reverse the contents.

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

// Function prototypes
void insertByOrder(int *arr, int *n, int key);
void deleteByPosition(int *arr, int *n, int pos);
void searchByKey(int *arr, int *n, int key);
void reverse(int *arr, int *n);
void printArray(int *arr, int *n);

// Function to insert an element in sorted order
void insertByOrder(int *arr, int *n, int key) {
    int i = *n - 1;

    // Shift elements to make space for new element
    while (i >= 0 && arr[i] > key) {
        arr[i + 1] = arr[i];
        i--;
    }

    // Insert the new element
    arr[i + 1] = key;
    (*n)++;
}

// Function to delete an element by position
void deleteByPosition(int *arr, int *n, int pos) {
    if (pos < 1 || pos > *n) {
        printf("Invalid position\n");
        return;
    }

    // Shift elements to overwrite the deleted element
    for (int i = pos - 1; i < *n - 1; i++) {
        arr[i] = arr[i + 1];
    }

    (*n)--;
}
```

```c
// Function to search for an element by its value
void searchByKey(int *arr, int *n, int key) {
    for (int i = 0; i < *n; i++) {
        if (arr[i] == key) {
            printf("Key %d found at position %d\n", key, i + 1);
            return;
        }
    }

    printf("Key %d not found\n", key);
}

// Function to reverse the elements of the array
void reverse(int *arr, int *n) {
    for (int i = 0; i < *n / 2; i++) {
        int temp = arr[i];
        arr[i] = arr[*n - i - 1];
        arr[*n - i - 1] = temp;
    }
}

// Function to print the elements of the array
void printArray(int *arr, int *n) {
    printf("Current array: ");
    for (int i = 0; i < *n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int *arr, n, c, choice;
    int pos, key;

    // Input capacity and size of the array
    printf("Enter the capacity and size of the array: ");
    scanf("%d %d", &c, &n);

    arr = (int *)malloc(c * sizeof(int));

    // Input elements of the array in sorted order
    printf("Enter %d elements of the array in sorted order:\n", n);
    for(int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Menu-driven program loop
    do {
        printf("\nMenu: \n");
        printf("1. Insert by order.\n");
        printf("2. Delete by position.\n");
        printf("3. Search by key.\n");
        printf("4. Reverse the content.\n");
        printf("5. Exit\n\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to insert: ");
                scanf("%d", &key);
                insertByOrder(arr, &n, key);
                break;
            case 2:
                printf("Enter the position to delete: ");
                scanf("%d", &pos);
                deleteByPosition(arr, &n, pos);
                break;
            case 3:
                printf("Enter the key to search: ");
                scanf("%d", &key);
                searchByKey(arr, &n, key);
                break;
            case 4:
                printf("The contents of the array after reversing are:\n");
                reverse(arr, &n);
                break;
            case 5:
                printf("Exiting the program\n");
                free(arr);
                return 0;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }

        printArray(arr, &n);
    } while (choice != 5);

    return 0;
}
```

4. Implement circular single linked list to perform the following operations
i) Insert by order
ii ) Delete by position
iii) Search for an item by key
iv) Delete by key
Display the list contents after each operation

6. Implement circular single linked list to perform the following operations
i) Insert front
ii) Insert rear
iii) Delete a node with the given key
iv) Search for an item by position
Display the list contents after each operation

8. Implement circular single linked list to perform the following operations
i) Insert by position
ii) Delete rear
iii) Delete Front
iv) Search for an item by value
Display the list contents after each operation

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

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

struct CircularList {
    struct Node *head;
};

typedef struct Node Node;
typedef struct CircularList List;

Node *createNode(int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if(newNode == NULL){
        printf("Memory allocation failed\n");
        return NULL;
    }
    newNode->data = data;
    newNode->link = NULL;
    return newNode;
}

void displayList(List *list) {
    if(list->head->link == NULL){
        printf("Empty\n");
        return;
    }
    Node *current = list->head->link;
    while (current->link != list->head->link) {
        printf("%d -> ", current->data);
        current = current->link;
    }
    printf("%d\n", current->data);
}

void freeList(List *list) {
    if(list->head->link == NULL)
        return;
    Node *current = list->head->link;
    while (current->link != list->head->link) {
        Node *temp = current;
        current = current->link;
        free(temp);
    }
    free(current);
    free(list->head);
    free(list);
}

List *insertAtBeginning(List *list, int data) {
    Node *newNode = createNode(data);
    Node *current = list->head->link;
    if (list->head->link == NULL) {
        list->head->link = newNode;
        newNode->link = newNode;
        list->head->data++;
        return list;
    }
    while(current->link != list->head->link){
        current = current->link;
    }
    current->link = newNode;
    newNode->link = list->head->link;
    list->head->link = newNode;
    list->head->data++;
    return list;
}

List *insertAtEnd(List *list, int data) {
    Node *newNode = createNode(data);
    Node *current = list->head->link;
    if (list->head->link == NULL) {
        list->head->link = newNode;
        newNode->link = newNode;
        list->head->data++;
        return list;
    }
    while (current->link != list->head->link) {
        current = current->link;
    }
    current->link = newNode;
    newNode->link = list->head->link;
    list->head->data++;
    return list;
}

List *insertAtPosition(List *list, int pos, int data) {
    if (pos == 1) {
        list = insertAtBeginning(list, data);
        return list;
    }
    else if (pos == list->head->data + 1){
        list = insertAtEnd(list, data);
        return list;
    }
    Node *current = list->head->link;
    for (int i = 1; i < pos - 1 && current->link != list->head->link; i++) {
        current = current->link;
    }
    if (current->link == list->head->link) {
        printf("Invalid position\n");
        return list;
    }
    Node *newNode = createNode(data);
    newNode->link = current->link;
    current->link = newNode;
    list->head->data++;
    return list;
}

List *deleteAtBeginning(List *list) {
    if (list->head->link == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return list;
    }
    Node *current = list->head->link;
    Node *tail = current->link;
    while (tail->link != list->head->link) {
        tail = tail->link;
    }
    if (tail == current) {
        list->head->link = NULL;
    } else {
        tail->link = current->link;
        list->head->link = current->link;
    }
    list->head->data--;
    free(current);
    return list;
}

List *deleteAtEnd(List *list) {
    if (list->head->link == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return list;
    }
    Node *current = list->head->link;
    Node *previous = NULL;
    while (current->link != list->head->link) {
        previous = current;
        current = current->link;
    }
    if (previous == NULL) {
        list->head->link = NULL;
    } else {
        previous->link = list->head->link;
    }
    list->head->data--;
    free(current);
    return list;
}

List *deleteAtPosition(List *list, int pos) {
    if (pos == 1) {
        list = deleteAtBeginning(list);
        return list;
    }
    if (pos == list->head->data) {
        list = deleteAtEnd(list);
        return list;
    }
    Node *current = list->head->link;
    Node *previous = NULL;
    for (int i = 1; i < pos && current->link != list->head->link; i++) {
        previous = current;
        current = current->link;
    }
    if (current->link == list->head->link) {
        printf("Invalid position\n");
    }
    else {
        previous->link = current->link;
        free(current);
    }
    list->head->data--;
    return list;
}

int searchByKey(List *list, int key) {
```

```c
    if(list->head->link == NULL){
        return -1;
    }
    Node *current = list->head->link;
    int position = 1;

    while (current->link != list->head->link && current->data != key) {
        current = current->link;
        position++;
    }

    if (current->data == key) {
        return position;
    } else {
        return -1;
    }
}

List *deleteByKey(List *list, int key) {
    int pos = searchByKey(list, key);
    if(pos == -1)
        printf("key not found\n");
    else
        list = deleteAtPosition(list, pos);
    return list;
}

List *createOrderedList(List *list, int key){
    if(list->head->link == NULL || key < list->head->link->data){
        list = insertAtBeginning(list, key);
        return list;
    }
    int pos = 1;
    Node *current = list->head->link;
    while(key > current->link->data && current->link != list->head->link){
        current = current->link;
        pos++;
    }
    list = insertAtPosition(list,pos + 1,key);
    return list;
}

List *copyList(List *list) {
    if(list->head->link == NULL){
        return list;
    }
    List *coList = (List *)malloc(sizeof(List));
    coList->head = createNode(0);

    Node *current = list->head->link;
    while (current->link != list->head->link) {
        coList = insertAtEnd(coList, current->data);
        current = current->link;
    }
    coList = insertAtEnd(coList, current->data);
    return coList;
}

void reverseList(List *list){
    if(list->head->link == NULL){
        printf("List is empty. Cannot reverse.\n");
        return;
    }
    Node *current = list->head->link;
    Node *previous = NULL;
    Node *next = NULL;
    Node *start = list->head->link;
    do {
        next = current->link;
        current->link = previous;
        previous = current;
        current = next;
    } while (current != list->head->link);
    list->head->link = previous;
    start->link = previous;
}

void printMenu() {
    printf("\nMenu:\n");
    printf("1. Insert at the beginning\n");
    printf("2. Insert at the end\n");
    printf("3. Insert at a specific position\n");
    printf("4. Delete at the beginning\n");
    printf("5. Delete at the end\n");
    printf("6. Delete at a specific position\n");
    printf("7. Delete by key\n");
    printf("8. Search by key\n");
    printf("9. Create an ordered list\n");
    printf("10. Copy the list\n");
    printf("11. Reverse the list\n");
    printf("0. Exit\n\n");
}

int main() {
    List *list = (List *)malloc(sizeof(List));
    list->head = createNode(0);

    int choice;
    int data, pos, key;

    printMenu();

    do {

       // printMenu();
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
```

```c
        switch (choice) {
            case 1:
                printf("Enter data to insert at the beginning: ");
                scanf("%d", &data);
                list = insertAtBeginning(list, data);
                break;

            case 2:
                printf("Enter data to insert at the end: ");
                scanf("%d", &data);
                list = insertAtEnd(list, data);
                break;

            case 3:
                printf("Enter position and data to insert at that position: ");
                scanf("%d %d", &pos, &data);
                list = insertAtPosition(list, pos, data);
                break;

            case 4:
                list = deleteAtBeginning(list);
                break;

            case 5:
                list = deleteAtEnd(list);
                break;

            case 6:
                printf("Enter position to delete from that position: ");
                scanf("%d", &pos);
                list = deleteAtPosition(list, pos);
                break;

            case 7:
                printf("Enter key to delete the node with that key: ");
                scanf("%d", &key);
                list = deleteByKey(list, key);
                break;

            case 8:
                printf("Enter key to search for: ");
                scanf("%d", &key);
                pos = searchByKey(list, key);
                if (pos == -1)
                    printf("Key not found\n");
                else
                    printf("Key %d found at position: %d\n", key, pos);
                break;

            case 9:
                printf("Enter data to create an ordered list: ");
                scanf("%d", &data);
                list = createOrderedList(list, data);
                break;
            case 10:
                printf("Copying the list...\n");
                List *copy = copyList(list);
                printf("Original List: ");
                displayList(list);
                printf("Copied List: ");
                displayList(copy);
                freeList(copy);
                goto end;
                break;

            case 11:
                printf("Reversing the list...\n");
                reverseList(list);
                break;

            case 0:
                printf("Exiting the program...\n");
                freeList(list);
                return 0;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
                goto end;
        }

        printf("Current List: ");
        displayList(list);
        end:;

    } while (choice != 0);
}
```

5. Implement circular double linked list to perform the following operations
i) Insert by order
ii ) Delete by position
iii ) Delete by key
iv) Search by position
Display the list contents after each operation

7. Implement circular double linked list to perform the following operations
i) Insert front
ii) Insert rear
iii) Delete by position
iv) Search by key
Display the list contents after each operation

9. Implement circular double linked list to perform the following operations
i) Insert by order
ii) Delete rear
iii) Delete Front
iv) Search for an item by position
Display the list contents after each operation

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

struct Node {
    int data;
    struct Node *nlink;
    struct Node *plink;
};

struct CircularDoubleList {
    struct Node *head;
};

typedef struct Node Node;
typedef struct CircularDoubleList List;

Node *createNode(int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return NULL;
    }
    newNode->data = data;
    newNode->nlink = NULL;
    newNode->plink = NULL;
    return newNode;
}

void displayList(List *list) {
    if (list->head->nlink == NULL) {
        printf("Empty\n");
        return;
    }
    Node *current = list->head->nlink;
    while (current->nlink != list->head->nlink) {
        printf("%d <-> ", current->data);
        current = current->nlink;
    }
    printf("%d\n", current->data);
}

void freeList(List *list) {
    if (list->head->nlink == NULL){
        free(list->head);
        free(list);
        return;
    }
    Node *current = list->head->nlink;
    do {
        Node *temp = current;
        current = current->nlink;
        free(temp);
    } while (current->nlink != list->head->nlink);
    free(current);
    free(list->head);
    free(list);
}

List *insertAtBeginning(List *list, int data) {
    Node *newNode = createNode(data);
    if (list->head->nlink == NULL) {
        list->head->nlink = newNode;
        newNode->nlink = newNode;
        newNode->plink = newNode;
    } else {
        Node *last = list->head->nlink->plink;
        last->nlink = newNode;
        newNode->nlink = list->head->nlink;
        newNode->plink = last;
        list->head->nlink = newNode;
    }
    list->head->data++;
    return list;
}

List *insertAtEnd(List *list, int data) {
    Node *newNode = createNode(data);
    if (list->head->nlink == NULL) {
        list->head->nlink = newNode;
        newNode->nlink = newNode;
        newNode->plink = newNode;
        list->head->data++;
        return list;
    }
    Node *last = list->head->nlink->plink;
    last->nlink = newNode;
    newNode->plink = last;
    newNode->nlink = list->head->nlink;
    list->head->nlink->plink = newNode;
    list->head->data++;
    return list;
}

List *insertAtPosition(List *list, int pos, int data) {
    if (pos == 1) {
        list = insertAtBeginning(list, data);
        return list;
    }
    else if (pos == list->head->data + 1){
        list = insertAtEnd(list, data);
        return list;
    }
    Node *current = list->head->nlink;
    for (int i = 1; i < pos - 1 && current->nlink != list->head->nlink; i++) {
        current = current->nlink;
    }
    if (current->nlink == list->head->nlink) {
        printf("Invalid position\n");
        return list;
    }
    Node *newNode = createNode(data);
    newNode->nlink = current->nlink;
    newNode->plink = current;
    current->nlink->plink = newNode;
    current->nlink = newNode;
    list->head->data++;
    return list;
}

List *deleteAtBeginning(List *list) {
    if (list->head->nlink == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return list;
    }
    Node *current = list->head->nlink;
    Node *last = current->plink;
    if (current == last) {
        list->head->nlink = NULL;
    } else {
        last->nlink = current->nlink;
        current->nlink->plink = last;
        list->head->nlink = current->nlink;
    }
    list->head->data--;
    free(current);
    return list;
}

List *deleteAtEnd(List *list) {
    if (list->head->nlink == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return list;
    }
    Node *current = list->head->nlink;
    Node *last = current->plink;
    Node *last2 = last->plink;
    if (current == last) {
        list->head->nlink = NULL;
    } else {
        last2->nlink = list->head->nlink;
        list->head->nlink->plink = last2;
    }
    list->head->data--;
    free(last);
    return list;
}

List *deleteAtPosition(List *list, int pos) {
    if (pos == 1) {
        list = deleteAtBeginning(list);
        return list;
    }
    if (pos == list->head->data) {
        list = deleteAtEnd(list);
        return list;
    }
    Node *current = list->head->nlink;
    Node *previous = NULL;
    for (int i = 1; i < pos && current->nlink != list->head->nlink; i++) {
        previous = current;
        current = current->nlink;
    }
    if (current->nlink == list->head->nlink) {
        printf("Invalid position\n");
        return list;
    }
    previous->nlink = current->nlink;
    current->nlink->plink = previous;
    free(current);
    list->head->data--;
    return list;
}

int searchByKey(List *list, int key) {
    if(list->head->nlink == NULL){
        return -1;
    }
    Node *current = list->head->nlink;
    int position = 1;

    while (current->nlink != list->head->nlink && current->data != key) {
        current = current->nlink;
        position++;
    }

    if (current->data == key) {
        return position;
    } else {
        return -1;
    }
}

List *deleteByKey(List *list, int key) {
    int pos = searchByKey(list, key);
    if(pos == -1)
        printf("key not found\n");
    else
        list = deleteAtPosition(list, pos);
    return list;
}

List *createOrderedList(List *list, int key){
    if(list->head->nlink == NULL || key < list->head->nlink->data){
        list = insertAtBeginning(list, key);
        return list;
    }
    int pos = 1;
    Node *current = list->head->nlink;
    while(key > current->nlink->data && current->nlink != list->head->nlink){
        current = current->nlink;
        pos++;
    }
    list = insertAtPosition(list,pos + 1,key);
    return list;
}

List *copyList(List *list) {
    if(list->head->nlink == NULL){
        return list;
    }
    List *coList = (List *)malloc(sizeof(List));
    coList->head = createNode(0);

    Node *current = list->head->nlink;
    while (current->nlink != list->head->nlink) {
        coList = insertAtEnd(coList, current->data);
        current = current->nlink;
    }
    coList = insertAtEnd(coList, current->data);
    return coList;
}

void reverseList(List *list){
    if(list->head->nlink == NULL){
        printf("List is empty. Cannot reverse.\n");
        return;
    }
    Node *current = list->head->nlink;
    Node *next = NULL;
    do {
        next = current->nlink;
        current->nlink = current->plink;
        current->plink = next;
        next->plink = current;
        current = current->plink;
    } while (current != list->head->nlink);
    list->head->nlink = current->nlink;
}

void printMenu() {
    printf("\nMenu:\n");
    printf("1. Insert at the beginning\n");
    printf("2. Insert at the end\n");
    printf("3. Insert at a specific position\n");
    printf("4. Delete at the beginning\n");
    printf("5. Delete at the end\n");
    printf("6. Delete at a specific position\n");
    printf("7. Delete by key\n");
    printf("8. Search by key\n");
    printf("9. Create an ordered list\n");
    printf("10. Copy the list\n");
    printf("11. Reverse the list\n");
    printf("0. Exit\n\n");
}

int main() {
    List *list = (List *)malloc(sizeof(List));
    list->head = createNode(0);

    int choice;
    int data, pos, key;

    printMenu();

    do {

       // printMenu();
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert at the beginning: ");
                scanf("%d", &data);
                list = insertAtBeginning(list, data);
                break;

            case 2:
                printf("Enter data to insert at the end: ");
                scanf("%d", &data);
                list = insertAtEnd(list, data);
                break;

            case 3:
                printf("Enter position and data to insert at that position: ");
                scanf("%d %d", &pos, &data);
                list = insertAtPosition(list, pos, data);
                break;

            case 4:
                list = deleteAtBeginning(list);
                break;

            case 5:
                list = deleteAtEnd(list);
                break;

            case 6:
                printf("Enter position to delete from that position: ");
                scanf("%d", &pos);
                list = deleteAtPosition(list, pos);
                break;

            case 7:
                printf("Enter key to delete the node with that key: ");
                scanf("%d", &key);
                list = deleteByKey(list, key);
                break;

            case 8:
                printf("Enter key to search for: ");
                scanf("%d", &key);
                pos = searchByKey(list, key);
                if (pos == -1)
                    printf("Key not found\n");
                else
                    printf("Key %d found at position: %d\n", key, pos);
                break;

            case 9:
                printf("Enter data to create an ordered list: ");
                scanf("%d", &data);
                list = createOrderedList(list, data);
                break;

            case 10:
```

```c
                printf("Copying the list...\n");
                List *copy = copyList(list);
                printf("Original List: ");
                displayList(list);
                printf("Copied List: ");
                displayList(copy);
                freeList(copy);
                goto end;
                break;

            case 11:
                printf("Reversing the list...\n");
                reverseList(list);
                break;

            case 0:
                printf("Exiting the program...\n");
                freeList(list);
                return 0;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
                goto end;
        }

        printf("Current List: ");
        displayList(list);
        end:;

    } while (choice != 0);
}
```

10. Develop a menu driven program to convert infix expression to postfix expression using stack and evaluate the postfix expression. (Test for nested parenthesized expressions)
Note: Define Stack structure and implement the operations. Use different stacks for conversion and evaluation.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

#define MAX_SIZE 100

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

void initialize(Stack *s) {
    s->top = -1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

int isFull(Stack *s) {
    return s->top == MAX_SIZE - 1;
}

void push(Stack *s, char c) {
    if (isFull(s)) {
        printf("Stack overflow\n");
        exit(1);
    }
    s->items[++(s->top)] = c;
}

char pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        exit(1);
    }
    return s->items[(s->top)--];
}

char peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
        exit(1);
    }
    return s->items[s->top];
}

int precedence(char op) {
    if (op == '^') {
        return 3;
    } else if (op == '*' || op == '/' || op == '%') {
        return 2;
    } else if (op == '+' || op == '-') {
        return 1;
    } else {
        return -1;
    }
}

void infixToPostfix(char *infix, char *postfix) {
    Stack stack;
    initialize(&stack);
    int i, j;
    for (i = 0, j = 0; infix[i] != '\0'; i++) {
        if (isalnum(infix[i])) {
            postfix[j++] = infix[i];
        } else if (infix[i] == '(') {
            push(&stack, infix[i]);
        } else if (infix[i] == ')') {
            while (!isEmpty(&stack) && peek(&stack) != '(') {
                postfix[j++] = pop(&stack);
            }
            if (!isEmpty(&stack) && peek(&stack) == '(') {
                pop(&stack); // Discard the '('
            } else {
                printf("Invalid expression\n");
                exit(1);
            }
        } else { // Operator
            while (!isEmpty(&stack) && precedence(infix[i]) <= precedence(peek(&stack)) &&
infix[i] != '^') {
                postfix[j++] = pop(&stack);
            }
            push(&stack, infix[i]);
        }
    }
    while (!isEmpty(&stack)) {
        if (peek(&stack) == '(' || peek(&stack) == ')') {
            printf("Invalid expression\n");
            exit(1);
        }
        postfix[j++] = pop(&stack);
    }
    postfix[j] = '\0';
}
```

```
int evaluatePostfix(char *postfix) {
    Stack stack;
    initialize(&stack);
    int i, operand1, operand2, result;
    for (i = 0; postfix[i] != '\0'; i++) {
        if (isdigit(postfix[i])) {
            push(&stack, postfix[i] - '0');
        } else {
            operand2 = pop(&stack);
            operand1 = pop(&stack);
            switch(postfix[i]) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    result = operand1 / operand2;
                    break;
                case '%':
                    result = operand1 % operand2;
                    break;
                case '^':
                    result = pow(operand1,operand2);
                    break;
                default:
                    printf("Invalid operator\n");
                    exit(1);
            }
            push(&stack, result);
        }
    }
    return pop(&stack);
}

int main() {
    char infix[MAX_SIZE], postfix[MAX_SIZE];
    int choice, result;
    do {
        printf("\nMenu:\n");
        printf("1. Convert infix to postfix\n");
        printf("2. Evaluate postfix expression\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice) {
            case 1:
                printf("Enter the infix expression: ");
```

```c
                scanf("%s", infix);
                infixToPostfix(infix, postfix);
                printf("Postfix expression: %s\n", postfix);
                break;
            case 2:
                printf("Enter the postfix expression: ");
                scanf("%s", postfix);
                result = evaluatePostfix(postfix);
                printf("Result of evaluation: %d\n", result);
                break;
            case 3:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (choice != 3);
    return 0;
}
```

11. Develop a menu driven program to convert infix expression to prefix expression using stack and evaluate the prefix expression (Test for nested parenthesized expressions)
Note: Define Stack structure and implement the operations. Use different stacks for conversion and evaluation.

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>

#define MAX_SIZE 100

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

void initialize(Stack *s) {
    s->top = -1;
}
```

```
int isEmpty(Stack *s) {
    return s->top == -1;
}

int isFull(Stack *s) {
    return s->top == MAX_SIZE - 1;
}

void push(Stack *s, char c) {
    if (isFull(s)) {
        printf("Stack overflow\n");
        exit(1);
    }
    s->items[++(s->top)] = c;
}

char pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        exit(1);
    }
    return s->items[(s->top)--];
}

char peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
        exit(1);
    }
    return s->items[s->top];
}

int precedence(char op) {
    if (op == '^') {
        return 3;
    } else if (op == '*' || op == '/' || op == '%') {
        return 2;
    } else if (op == '+' || op == '-') {
        return 1;
    } else {
        return -1;
    }
}

void infixToPrefix(char *infix, char *prefix) {
    Stack stack;
    initialize(&stack);
    int i, j = 0;
    int length = strlen(infix);

    // Reverse the infix expression and convert it into nearly postfix
    for (i = length - 1; i >= 0; i--) {
        if (isalnum(infix[i])) {
            prefix[j++] = infix[i];
        } else if (infix[i] == ')') {
            push(&stack, infix[i]);
        } else if (infix[i] == '(') {
            while (!isEmpty(&stack) && peek(&stack) != ')') {
                prefix[j++] = pop(&stack);
            }
            if (!isEmpty(&stack) && peek(&stack) == ')') {
                pop(&stack); // Discard the ')'
            } else {
                printf("Invalid expression\n");
                exit(1);
            }
        } else { // Operator
            while (!isEmpty(&stack) &&
                    (precedence(infix[i]) < precedence(peek(&stack)) ||
                    peek(&stack) == '^' && infix[i] == '^')) {
                prefix[j++] = pop(&stack);
            }
            push(&stack, infix[i]);
        }
    }
    // Pop remaining operators from the stack
    while (!isEmpty(&stack)) {
        if (peek(&stack) == '(' || peek(&stack) == ')') {
            printf("Invalid expression\n");
            exit(1);
        }
        prefix[j++] = pop(&stack);
    }
    prefix[j] = '\0';

    // Reverse the prefix expression to get the correct result
    length = strlen(prefix);
    for (i = 0; i < length / 2; i++) {
        char temp = prefix[i];
        prefix[i] = prefix[length - i - 1];
        prefix[length - i - 1] = temp;
    }
}

int evaluatePrefix(char *prefix) {
    Stack stack;
    initialize(&stack);
    int i, result;
    int length = strlen(prefix);

    // Evaluate prefix expression from right to left
    for (i = length - 1; i >= 0; i--) {
        if (isdigit(prefix[i])) {
            push(&stack, prefix[i] - '0');
        } else {
            int operand1 = pop(&stack);
            int operand2 = pop(&stack);
```

```c
            switch(prefix[i]) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    result = operand1 / operand2;
                    break;
                case '%':
                    result = operand1 % operand2;
                    break;
                case '^':
                    result = pow(operand1, operand2);
                    break;
                default:
                    printf("Invalid operator\n");
                    exit(1);
            }
            push(&stack, result);
        }
    }
    return pop(&stack);
}


int main() {
    char infix[MAX_SIZE], prefix[MAX_SIZE];
    int choice, result;
    do {
        printf("\nMenu:\n");
        printf("1. Convert infix to prefix\n");
        printf("2. Evaluate prefix expression\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice) {
            case 1:
                printf("Enter the infix expression: ");
                scanf("%s", infix);
                infixToPrefix(infix, prefix);
                printf("Prefix expression: %s\n", prefix);
                break;
            case 2:
                printf("Enter the prefix expression: ");
                scanf("%s", prefix);
                result = evaluatePrefix(prefix);
                printf("Result of evaluation: %d\n", result);
                break;
            case 3:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (choice != 3);
    return 0;
}
```

12. Develop a menu driven program to implement the following types of Queues by allocating memory dynamically.
i) Circular Queue
ii) Double ended Queue
Note: Define Queue structure and implement the operation

13 . Develop a menu driven program to implement the following types of Queues by allocating memory dynamically.
i) Circular Queue
ii) Priority Queue
Note: Define Queue structure and implement the operation

```c
//implementation of Circular Queue using static memory allocation.

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

#define MAX 10

typedef struct{
    int front, rear;
    int arr[MAX];
} Que;

void initQ(Que *q) {
    q->front = -1;
    q->rear = -1;
}

int isEmpty(Que* q) {
    return (q->front == -1 && q->rear == -1);
}

int isFull(Que* q) {
    return ((q->rear + 1) % MAX == q->front);
}
```

```c
void enque(Que* q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow!!!\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
        q->rear = 0;
    } else {
        q->rear = (q->rear + 1) % MAX;
    }
    q->arr[q->rear] = value;
    printf("%d inserted into the Queue\n", value);
}

int deque(Que* q) {
    if (isEmpty(q)) {
        printf("Queue Underflow!!!\n");
        return -1;
    }
    int data = q->arr[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX;
    }
    return data;
}

void display(Que* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    int temp = q->front;
    do {
        printf("%d ", q->arr[temp]);
        temp = (temp + 1) % MAX;
    } while (temp != (q->rear + 1) % MAX);
    printf("\n");
}

int main() {
    Que q;
    initQ(&q);
    int choice, data;
    do {
        printf("\nQueue Operations Menu\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice:  ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the data to insert: ");
                scanf("%d", &data);
                enque(&q, data);
                break;
            case 2:
                data = deque(&q);
                if(data != -1)
                    printf("Dequeued element is : %d\n", data);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("Exiting program...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

```c
//implementation of Double Ended Queue using static memory allocation.

#include<stdio.h>
#include<stdlib.h>
#define MAX 10

typedef struct Queue{
    int arr[MAX];
    int front;
    int rear;
}Que;

void initQue(Que* q) {
    q->front = -1;
    q->rear = -1;
}

int isEmpty(Que* q) {
    return ((q->front == -1) && (q->rear == -1));
}
```

```c
void display(Que* q){
    if(isEmpty(q)){
        printf("Queue is Empty\n");
        return;
    }
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d ", q->arr[i]);
    }
    printf("\n");
}

void insertAtFront(Que* q, int val){
    if(isEmpty(q)){
        q->front = 0;
        q->arr[++q->rear] = val;
        printf("Inserted %d at front\n", val);
    }
    else if(q->front > 0){
        q->arr[--q->front] = val;
        printf("Inserted %d at front\n", val);
    }
    else
        printf("Front insertion not possible\n");
}

void insertAtRear(Que* q, int val){
    if(isEmpty(q)){
        q->front = 0;
        q->arr[++q->rear] = val;
        printf("Inserted %d at rear\n", val);
    }
    else if(q->rear != MAX-1){
        q->arr[++q->rear] = val;
        printf("Inserted %d at rear\n", val);
    }
    else
        printf("Rear insertion is not possible\n");
}

int deleteAtFront(Que* q){
    if(isEmpty(q)){
        printf("Empty Queue\nDelete At Front Not Possible\n");
        return -1;
    }
    int item = q->arr[q->front++];
    if(q->front > q->rear){
        q->front = -1;
        q->rear = -1;
    }
    return item;
}

int deleteAtRear(Que* q){
    if(isEmpty(q)){
        printf("Empty Queue\nDelete At Rear Not Possible\n");
        return -1;
    }
    int item = q->arr[q->rear--];
    if(q->front > q->rear){
        q->front = -1;
        q->rear = -1;
    }
    return item;
}

int main(){
    Que q;
    initQue(&q);
    int choice, val;
    do{
        printf("\n1- Insert at front\n2- Insert at rear\n");
        printf("3- Delete at front\n4- Delete at rear\n");
        printf("5- Display\n6- Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("Enter value to insert at front: ");
                    scanf("%d",&val);
                    insertAtFront(&q, val);
                    break;
            case 2: printf("Enter value to insert at rear: ");
                    scanf("%d",&val);
                    insertAtRear(&q, val);
                    break;
            case 3: val = deleteAtFront(&q);
                    if(val != -1)
                        printf("Deleted element at front is: %d\n",val);
                    break;
            case 4: val = deleteAtRear(&q);
                    if(val != -1)
                        printf("Deleted element at rear is: %d\n",val);
                    break;
            case 5: display(&q);
                    break;
            case 6: printf("Exiting...\n");
                    break;
            default : printf("Invalid Choice\n... Retry");
        }
    } while(choice!=6);
    return 0;
}
```

```c
//Priority Queue - Design 2 : Dequeue is normal, Enqueue is ordered.

#include<stdio.h>
```

```c
#include<stdlib.h>
#define MAX 10

struct Queue{
    int front, rear;
    int arr[MAX];
};

typedef struct Queue Que;

void initQ(Que *q) {
    q->front = -1;
    q->rear = -1;
}

int isEmpty(Que* q) {
    return (q->front == -1 && q->rear == -1);
}

int isFull(Que* q) {
    return (q->rear == MAX - 1);
}

void enque(Que* q, int val){
    if(isFull(q)){
        printf("Queue Overflow\n");
        return;
    }
    if(isEmpty(q)){
        q->front = 0;
    }
    int j = q->rear;
    while(j >= q->front && val < q->arr[j]){
        q->arr[j+1] = q->arr[j];
        j--;
    }
    q->arr[j+1] = val;
    q->rear++;
}

int deque(Que* q) {
    if (isEmpty(q)) {
        printf("Queue Underflow!!!\n");
        return -1;
    }
    int data = q->arr[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    } else {
        q->front++;
    }
    return data;
}
```

```c
void display(Que* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d ", q->arr[i]);
    }
    printf("\n");
}

int main() {
    Que q;
    initQ(&q);
    int choice, data;
    do {
        printf("\nQueue Operations Menu\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n\n");
        printf("Enter your choice:  ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the data to insert: ");
                scanf("%d", &data);
                enque(&q,data);
                break;
            case 2:
                data = deque(&q);
                if(data != -1)
                    printf("Dequeued element is : %d\n", data);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("Exiting program...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

14. Develop a menu driven program to implement Binary Search tree with the following operations.
i) Construction
ii) Traversals (Pre, In and Post Order)
iii) Searching a node by key and displaying its information along with its parent if exists, otherwise a suitable message.
iv)Counting all nodes.
v) Finding height.
vi) Finding node with the Maximum key value and printing the node details along with the parent.

//Not there for lab internals

15. Develop a menu driven program to implement Binary Search tree with the following operations.

i) Construction

ii) Traversals( Pre, In and Post Order)

iii) Searching a node by key and deleting if exists ( node to be deleted may be leaf or non- leaf with one child or two children)

//Not there for lab internals