```
//1)
#include <stdio.h>
#include <string.h>
#define MAX 50
struct Library
{
   char title[50];
   int id;
   int sfreq;
};
struct Library s[MAX];
int total_searches = 0;
// Check if entered book ID already exists
int isDuplicate(int id, int n)
{
   for (int i = 0; i < n; i++)
   {
      if (s[i].id == id)
         return 1;
   }
   return 0;
}
// Input book details
void inputDetails(int n)
{
   for (int i = 0; i < n; )
   {
      printf("Book %d Title: ", i + 1);
      fgets(s[i].title, sizeof(s[i].title), stdin);
      s[i].title[strcspn(s[i].title, "\n")] = '\0';
      printf("Book %d ID: ", i + 1);
      scanf("%d", &s[i].id);
      getchar();


      if (isDuplicate(s[i].id, i))
      {
         printf("Duplicate ID found. Please re-enter.\n");
         continue;
      }


      s[i].sfreq = 0;
      i++;
   }
}
```

```
// Display all books
void displayDetails(int n)
{
   for (int i = 0; i < n; i++)
   {
      printf("Book title : %s\t\t\tBook id : %d\n", s[i].title, s[i].id);
   }
}
// Bubble sort books by their IDs
void sort(int n)
{
   for (int i = 0; i < n - 1; i++)
   {
      for (int j = 0; j < n - 1 - i; j++)
      {
         if (s[j].id > s[j + 1].id)
         {
            struct Library temp = s[j];
            s[j] = s[j + 1];
            s[j + 1] = temp;
         }
      }
   }
}
// Binary search to find book by ID, increments search frequency
void binSearch(int n, int key)
{
   int l = 0, u = n - 1, mid;
   while (l <= u)
   {
      mid = (l + u) / 2;
      if (s[mid].id == key)
      {
         printf("Book Found at index: %d\n", mid);
         printf("Title: %s\n", s[mid].title);
         s[mid].sfreq++;
         total_searches++;
         return;
      } else if (key < s[mid].id)
         u = mid - 1;
      else
         l = mid + 1;
   }
   printf("Book with ID %d not found.\n", key);
}
// Show top 3 most searched books
```

```
void topSearches(int n)
{
   struct Library tempArr[MAX];
   for (int i = 0; i < n; i++)
   {
      tempArr[i] = s[i]; // Copy original data
   }

   // Sort the copy by frequency (descending)
   for (int i = 0; i < n - 1; i++)
   {
      for (int j = 0; j < n - 1 - i; j++)
      {
         if (tempArr[j].sfreq < tempArr[j + 1].sfreq)
         {
            struct Library temp = tempArr[j];
            tempArr[j] = tempArr[j + 1];
            tempArr[j + 1] = temp;
         }
      }
   }
   // Show top 3 with frequency > 0
   for (int i = 0; i < n && i < 3; i++)
   {
      if (tempArr[i].sfreq > 0)
         printf("ID: %d\tTitle: %s\tFrequency: %d\n",
               tempArr[i].id, tempArr[i].title, tempArr[i].sfreq);
   }
}
void summary(int n)
{
   printf("\n Summary \n");
   printf("Total Searches Done: %d\n", total_searches);
   topSearches(n);
   printf("\nThank you for using the system \n");
}
int main()
{
   int n;
   printf("Input the number of books: ");
   scanf("%d", &n);
   getchar();
   char ch;
   int option, key;
   inputDetails(n);
   do
```

```
    {
        printf("---MENU---\n");
        printf("1. Sort Books by ID\n");
        printf("2. Search Book by ID (Binary Search)\n");
        printf("3. Show Top 3 Searched Books\n");
        printf("4. Show Performance Info\n");
        printf("5. Exit\n");
        printf("Input your choice: ");
        scanf("%d", &option);
        switch (option)
        {
            case 1:
                sort(n);
                displayDetails(n);
                break;
            case 2:
                printf("Input the book id to be found: ");
                scanf("%d", &key);
                binSearch(n, key);
                break;
            case 3:
                topSearches(n);
                break;
            case 4:
                printf("\nBubble Sort:\nTime Complexity: O(n^2)");
                printf("\nBinary Search:\nTime Complexity: O(log n)\n");
                break;
            case 5:
                summary(n);
                return 0;
            default:
                printf("Invalid option.\n");
        }
        printf("Do you wish to continue(Y/N):");
        scanf(" %c", &ch);
    } while (ch == 'y' || ch == 'Y');
    return 0;
}
```

```
//2)
#include <stdio.h>
struct Term
{
   int row;
   int col;
   int value;
};
void readMatrix(int rows, int cols, struct Term sparse[])
{
   int matrix[rows][cols];
   int i, j, k = 1;
   printf("Enter matrix :\n");
   for (i = 0; i < rows; i++)
   {
      for (j = 0; j < cols; j++)
      {
         scanf("%d", &matrix[i][j]);
         if (matrix[i][j] != 0)
         {
            sparse[k].row = i;
            sparse[k].col = j;
            sparse[k].value = matrix[i][j];
            k++;
         }
      }
   }
   sparse[0].row = rows;
   sparse[0].col = cols;
   sparse[0].value = k - 1;
}
void displaySparse(struct Term sparse[])
{
   int i;
   printf("\nRow\tCol\tValue\n");
   for (i = 0; i <= sparse[0].value; i++)
   {
      printf("%d\t%d\t%d\n", sparse[i].row, sparse[i].col, sparse[i].value);
   }
}
void transpose(struct Term original[], struct Term transposed[])
{
   int i, j, k = 1;
   int numCols = original[0].col;
   transposed[0].row = original[0].col;
```

```
    transposed[0].col = original[0].row;
    transposed[0].value = original[0].value;
    for (i = 0; i < numCols; i++)
    {
        for (j = 1; j <= original[0].value; j++)
        {
            if (original[j].col ==i)
            {
                transposed[k].row = original[j].col;
                transposed[k].col = original[j].row;
                transposed[k].value = original[j].value;
                k++;
            }
        }
    }
}
int main()
{
    int rows, cols;
    struct Term sparse[100], transposed[100];
    printf("Enter rows and cols:\n ");
    scanf("%d %d", &rows, &cols);
    readMatrix(rows, cols, sparse);
    printf("\nOriginal Sparse Matrix:");
    displaySparse(sparse);
    transpose(sparse, transposed);
    printf("\nTransposed Sparse Matrix:");
    displaySparse(transposed);
    return 0;
}
```

```
//3)
#include <stdio.h>
#define MAX 100
struct Term
{
    int coeff;
    int exp;
};
// Reads polynomial terms from user
void readPoly(struct Term poly[], int *n)
{
    int i;
```

```
    printf("Enter number of terms: ");
    scanf("%d", n);
    printf("Enter coefficient and exponent for each term:\n");
    for (i = 0; i < *n; i++) {
        printf("Term %d (coeff exp): ", i + 1);
        scanf("%d %d", &poly[i].coeff, &poly[i].exp);
    }
}
// Displays polynomial in readable format
void displayPoly(struct Term poly[], int n)
{
    for (int i = 0; i < n; i++)
    {
        if (poly[i].coeff == 0)
            continue;
        if (i > 0 && poly[i].coeff > 0)
            printf(" + ");
        if (poly[i].exp == 0)
            printf("%d", poly[i].coeff);
        else if (poly[i].coeff == 1)
            printf("x^%d", poly[i].exp);
        else if (poly[i].coeff == -1)
            printf("-x^%d", poly[i].exp);
        else
            printf("%dx^%d", poly[i].coeff, poly[i].exp);
    }
    printf("\n");
}
// Sorts polynomial terms by exponent descending
void sortPoly(struct Term poly[], int n)
{
    int i, j;
    for (i = 0; i < n -1; i++)
    {
        for (j = i +1; j < n; j++)
        {
            if (poly[i].exp < poly[j].exp)
            {
                struct Term temp = poly[i];
                poly[i] = poly[j];
                poly[j] = temp;
            }
        }
    }
}
// Adds two polynomials and returns number of terms in result
```

```
int addPoly(struct Term p1[], int n1, struct Term p2[], int n2, struct Term result[])
{
   int i = 0, j = 0, k = 0;
   sortPoly(p1, n1);
   sortPoly(p2, n2);
   while (i < n1 && j < n2)
   {
      if (p1[i].exp > p2[j].exp)
      {
         result[k++] = p1[i++];
      }
      else if (p1[i].exp < p2[j].exp)
      {
         result[k++] = p2[j++];
      }
      else
      {
         int sumCoeff = p1[i].coeff + p2[j].coeff;
         if (sumCoeff != 0)
         {
            result[k].exp = p1[i].exp;
            result[k].coeff = sumCoeff;
            k++;
         }
         i++; j++;
      }
   }
   while (i < n1)
      result[k++] = p1[i++];
   while (j < n2)
      result[k++] = p2[j++];
   return k;
}
// Subtracts p2 from p1 and returns number of terms in result
int subtractPoly(struct Term p1[], int n1, struct Term p2[], int n2, struct Term result[])
{
   int i = 0, j = 0, k = 0;
   sortPoly(p1, n1);
   sortPoly(p2, n2);
   while (i < n1 && j < n2)
   {
      if (p1[i].exp > p2[j].exp)
      {
         result[k++] = p1[i++];
      }
      else if (p1[i].exp < p2[j].exp)
```

```
        {
            result[k].exp = p2[j].exp;
            result[k].coeff = -p2[j].coeff;
            k++; j++;
        }
        else
        {
            int diffCoeff = p1[i].coeff - p2[j].coeff;
            if (diffCoeff != 0)
            {
                result[k].exp = p1[i].exp;
                result[k].coeff = diffCoeff;
                k++;
            }
            i++; j++;
        }
    }
    while (i < n1)
        result[k++] = p1[i++];
    while (j < n2)
    {
        result[k].exp = p2[j].exp;
        result[k].coeff = -p2[j].coeff;
        k++; j++;
    }
    return k;
}
// Multiplies two polynomials and returns number of terms in result
int multiplyPoly(struct Term p1[], int n1, struct Term p2[], int n2, struct Term result[])
{
    struct Term temp[MAX * MAX];
    int count = 0;
    // Multiply each term of p1 by each term of p2
    for (int i = 0; i < n1; i++)
    {
        for (int j = 0; j < n2; j++)
        {
            temp[count].coeff = p1[i].coeff * p2[j].coeff;
            temp[count].exp = p1[i].exp + p2[j].exp;
            count++;
        }
    }
    // Combine like terms with same exponent
    int k = 0;
    for (int i = 0; i < count; i++)
    {
```

```
        if (temp[i].coeff == 0) continue;
        int coeffSum = temp[i].coeff;
        int exp = temp[i].exp;
        for (int j = i + 1; j < count; j++)
        {
            if (temp[j].exp == exp) {
                coeffSum += temp[j].coeff;
                temp[j].coeff = 0; // mark as combined
            }
        }
        if (coeffSum != 0)
        {
            result[k].coeff = coeffSum;
            result[k].exp = exp;
            k++;
        }
    }
    sortPoly(result, k);
    return k;
}
int main()
{
    struct Term poly1[MAX], poly2[MAX], result[MAX];
    int n1 = 0, n2 = 0, n3 = 0;
    int choice;
    while (1)
    {
        printf("\nMenu:\n");
        printf("1. Input Polynomials\n");
        printf("2. Add Polynomials\n");
        printf("3. Subtract Polynomials\n");
        printf("4. Multiply Polynomials\n");
        printf("5. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("First Polynomial:\n");
                readPoly(poly1, &n1);
                printf("Second Polynomial:\n");
                readPoly(poly2, &n2);
                break;
            case 2:
                n3 = addPoly(poly1, n1, poly2, n2, result);
                printf("Sum: ");
```

```
            displayPoly(result, n3);
            break;
        case 3:
            n3 = subtractPoly(poly1, n1, poly2, n2, result);
            printf("Difference: ");
            displayPoly(result, n3);
            break;
        case 4:
            n3 = multiplyPoly(poly1, n1, poly2, n2, result);
            printf("Product: ");
            displayPoly(result, n3);
            break;
        case 5:
            return 0;
        default:
            printf("Invalid choice. Try again.\n");
    }
  }
}
```

```
//4)
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#define SIZE 100
char stack[SIZE];
int top = -1;
int isOperator(char c)
{
   return (c == '+' || c == '-' || c == '*' || c == '/' || c == '%' || c == '^');
}
int precedence(char op)
{
   switch (op)
   {
      case '^': return 3;
      case '*': case '/': case '%': return 2;
      case '+': case '-': return 1;
      default: return 0;
   }
}
void push(char c)
{
   if (top < SIZE - 1)
      stack[++top] = c;
}
char pop()
{
   if (top >= 0)
      return stack[top--];
   return '\0';
}
char peek()
{
   if (top >= 0)
      return stack[top];
   return '\0';
}
void infixToPostfix(char infix[], char postfix[])
{
   int i = 0, j = 0;
   char ch;
   while ((ch = infix[i++]) != '\0')
   {
      if (isalnum(ch))
      {
```

```
            postfix[j++] = ch;
        }
        else if (ch == '(')
        {
            push(ch);
        }
        else if (ch == ')')
        {
            while (peek() != '(')
                postfix[j++] = pop();
            pop();
        }
        else if (isOperator(ch))
        {
          while (isOperator(peek()) &&
       (precedence(peek()) >= precedence(ch)))
        {
                postfix[j++] = pop();
            }
            push(ch);
        }
    }
    while (top != -1)
        postfix[j++] = pop();
    postfix[j] = '\0';
}
int evalStack[SIZE];
int evalTop = -1;
void evalPush(int num)
{
    if (evalTop < SIZE - 1)
        evalStack[++evalTop] = num;
}
int evalPop()
{
    if (evalTop >= 0)
        return evalStack[evalTop--];
    return 0;
}
int evaluatePostfix(char postfix[])
{
    int i = 0;
    char ch;
    while ((ch = postfix[i++]) != '\0')
    {
        if (isalnum(ch))
```

```
        {
            int val;
            if (isdigit(ch))
            {
                val = ch - '0';
            }
            else
            {
                printf("Enter value for %c: ", ch);
                scanf("%d", &val);
            }
            evalPush(val);
        }
        else if (isOperator(ch))
        {
            int b = evalPop();
            int a = evalPop();
            switch (ch)
            {
                case '+': evalPush(a + b); break;
                case '-': evalPush(a - b); break;
                case '*': evalPush(a * b); break;
                case '/': evalPush(a / b); break;
                case '%': evalPush(a % b); break;
                case '^': evalPush((int)pow(a, b)); break;
            }
        }
    }
    return evalPop();
}
int main()
{
    char infix[SIZE], postfix[SIZE];
    printf("Enter infix expression: ");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);
    printf("Postfix: %s\n", postfix);
    int result = evaluatePostfix(postfix);
    printf("Result: %d\n", result);
    return 0;
}
```

```
//5)
#include <stdio.h>
#define SIZE 5
int opCount = 0;  // Count total enqueue/dequeue operations performed
// Arrays and front/rear pointers for each queue type
int q[SIZE];
int qFront = -1, qRear = -1;
int cq[SIZE];
int cqFront = -1, cqRear = -1;
int dq[SIZE];
int dqFront = -1, dqRear = -1;
// Standard Queue enqueue
void enqueueQ(int val)
{
   if (qRear == SIZE - 1)
   {
      printf("Normal Queue is FULL!\n");
      return;
   }
   if (qFront == -1) qFront = 0;
   q[++qRear] = val;
   opCount++;
}
// Standard Queue dequeue
int dequeueQ()
{
   if (qFront == -1)
   {
      printf("Normal Queue is EMPTY!\n");
      return -1;  // Signal empty queue
   }
   int val = q[qFront++];
   if (qFront > qRear) qFront = qRear = -1;  // Reset when empty
   opCount++;
   return val;
}
// Circular Queue enqueue
void enqueueCQ(int val)
{
   if ((cqRear + 1) % SIZE == cqFront)
   {
      printf("Circular Queue is FULL!\n");
      return;
   }
   if (cqFront == -1) cqFront = 0;
```

```
    cqRear = (cqRear + 1) % SIZE;
    cq[cqRear] = val;
    opCount++;
}
// Circular Queue dequeue
int dequeueCQ()
{
    if (cqFront == -1)
    {
        printf("Circular Queue is EMPTY!\n");
        return -1;
    }
    int val = cq[cqFront];
    if (cqFront == cqRear)
    {
        cqFront = cqRear = -1;
    }
    else
    {
        cqFront = (cqFront + 1) % SIZE;
    }
    opCount++;
    return val;
}
// Insert at front of deque (used in output-restricted deque)
void insertFront(int val)
{
    if ((dqRear + 1) % SIZE == dqFront)
    {
        printf("Deque is FULL!\n");
        return;
    }
    if (dqFront == -1)
    {
        dqFront = dqRear = 0;
    }
    else if (dqFront == 0)
    {
        dqFront = SIZE - 1;
    }
    else
    {
        dqFront--;
    }
    dq[dqFront] = val;
    opCount++;
```

```
}
// Insert at rear of deque
void insertRear(int val)
{
   if ((dqRear + 1) % SIZE == dqFront)
   {
      printf("Deque is FULL!\n");
      return;
   }
   if (dqFront == -1)
   {
      dqFront = dqRear = 0;
   }
   else
   {
      dqRear = (dqRear + 1) % SIZE;
   }
   dq[dqRear] = val;
   opCount++;
}
// Delete from front of deque
int deleteFront()
{
   if (dqFront == -1)
   {
      printf("Deque is EMPTY!\n");
      return -1;
   }
   int val = dq[dqFront];
   if (dqFront == dqRear)
   {
      dqFront = dqRear = -1;
   }
   else
   {
      dqFront = (dqFront + 1) % SIZE;
   }
   opCount++;
   return val;
}
// Delete from rear of deque (used in input-restricted deque)
int deleteRear()
{
   if (dqFront == -1)
   {
      printf("Deque is EMPTY!\n");
```

```
        return -1;
    }
    int val = dq[dqRear];
    if (dqFront == dqRear)
    {
        dqFront = dqRear = -1;
    }
    else if (dqRear == 0)
    {
        dqRear = SIZE - 1;
    }
    else
    {
        dqRear--;
    }
    opCount++;
    return val;
}
int main()
{
    int choice, val, ch2, dqMode;
    char cont;
    printf("--- Buffer Management Simulation ---\n");
    do
    {
        printf("\n--- Main Menu ---\n");
        printf("1. Normal Queue\n");
        printf("2. Circular Queue\n");
        printf("3. Deque (Input-Restricted / Output-Restricted)\n");
        printf("4. Show Operation Count & Complexity\n");
        printf("5. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                printf("1. Enqueue\n2. Dequeue\n");
                scanf("%d", &ch2);
                if (ch2 == 1)
                {
                    printf("Value: "); scanf("%d", &val);
                    enqueueQ(val);
                }
                else if (ch2 == 2)
                {
                    int removed = dequeueQ();
```

```
            // Only print if dequeue was successful
            if (removed != -1) printf("Removed: %d\n", removed);
        }
        else printf("Invalid choice!\n");
        break;
    case 2:
        printf("1. Enqueue\n2. Dequeue\n");
        scanf("%d", &ch2);
        if (ch2 == 1)
        {
            printf("Value: "); scanf("%d", &val);
            enqueueCQ(val);
        }
        else if (ch2 == 2)
        {
            int removed = dequeueCQ();
            if (removed != -1) printf("Removed: %d\n", removed);
        }
        else printf("Invalid choice!\n");
        break;
    case 3:
        printf("Select Deque Type:\n");
        printf("1. Input-Restricted Deque (Insert Rear, Delete Front/Rear)\n");
        printf("2. Output-Restricted Deque (Insert Front/Rear, Delete Front)\n");
        scanf("%d", &dqMode);
        if (dqMode == 1)
        {
            printf("1. Insert Rear\n2. Delete Front\n3. Delete Rear\n");
            scanf("%d", &ch2);
            if (ch2 == 1)
            {
                printf("Value: "); scanf("%d", &val);
                insertRear(val);
            }
            else if (ch2 == 2)
            {
                int removed = deleteFront();
                if (removed != -1) printf("Removed: %d\n", removed);
            }
            else if (ch2 == 3)
            {
                int removed = deleteRear();
                if (removed != -1) printf("Removed: %d\n", removed);
            }
            else
            {
```

```
            printf("Invalid choice\n");
        }
    }
    else if (dqMode == 2)
    {
        printf("1. Insert Front\n2. Insert Rear\n3. Delete Front\n");
        scanf("%d", &ch2);
        if (ch2 == 1)
        {
            printf("Value: "); scanf("%d", &val);
            insertFront(val);
        }
        else if (ch2 == 2)
        {
            printf("Value: "); scanf("%d", &val);
            insertRear(val);
        }
        else if (ch2 == 3)
        {
            int removed = deleteFront();
            if (removed != -1) printf("Removed: %d\n", removed);
        }
        else
        {
            printf("Invalid choice\n");
        }
    }
    else
    {
        printf("Invalid deque type.\n");
    }
    break;
  case 4:
    printf("Total operations performed: %d\n", opCount);
    printf("Time Complexity:O(1)\n");
    printf("Space Complexity:O(n)\n");
    break;
  case 5:
    return 0;
  default:
    printf("Invalid choice!\n");
}
printf("Do you wish to continue (y/n)? ");
scanf(" %c", &cont);
```

```
    } while (cont == 'y' || cont == 'Y');
    return 0;
}
```

```
//6)
#include <stdio.h>
#include <stdlib.h>
struct patient
{
   int id;
   char name[50];
   int priority;
   struct patient *next;
};
struct patient *head=NULL;
void insertAtEnd() //insert at end
{
   struct patient *newnode=(struct patient*)malloc(sizeof(struct patient)),*temp=head;
   printf("Input name: ");
   getchar();
   scanf("%[^\n]", newnode->name);
   printf("Input priority: ");
   scanf("%d",&newnode->priority);
   printf("Input id: ");
   scanf("%d",&newnode->id);
   newnode->next=NULL;
   if(head==NULL)
   {
      head=newnode;
      return;
   }
   while(temp->next!=NULL)
   {
      temp=temp->next;
   }
   temp->next=newnode;
}
void deletePatientById(int id) //delete patient by id
{
   if(head==NULL)
   {
      printf("Patient Queue is empty\n");
      return;
   }
   struct patient *temp=head;
   if(head->id==id)
   {
      head=head->next;
```

```
        free(temp);
        printf("Patient with ID %d deleted.\n", id);
        return;
    }
    struct patient *prev=NULL;
    while(temp->next!=NULL&&temp->id!=id)
    {
        prev=temp;
        temp=temp->next;
    }
    if(temp==NULL)
    {
        printf("Patient with %d id not found\n",id);
        return;
    }
    prev->next=temp->next;
    free(temp);
    printf("Patient with ID %d deleted.\n", id);
}
void displayPatients(struct patient *head) //display patient queue
{
    if(head==NULL)
    {
        printf("Patient Queue is empty\n");
        return;
    }
    struct patient *temp=head;
    while(temp!=NULL)
    {
        printf("ID: %d   Name: %s   Priority: %d\n",temp->id,temp->name,temp->priority);
        temp=temp->next;
    }
}
void swapWithFront()
{
    struct patient *curr,*prev,*highest,*highestprev;
    if(head==NULL||head->next==NULL)// null or only one element
    {
        return;
    }
    highest=head;
    curr=head;
    highestprev=NULL;
    prev=NULL;
    while(curr!=NULL)
    {
```

```
        if(curr->priority<highest->priority)
        {
           highest=curr;
           highestprev=prev;
        }
        prev=curr;
        curr=curr->next;
    }
    if(highest==head)//head has highest priority
        return;
    if(highest->priority<head->priority)//head has lesser priority then move it to front
    {
        highestprev->next=highest->next;
        highest->next=head;
        head=highest;
    }
}
struct patient* reverseInGroups(struct patient *head,int k)
{
    if(head==NULL)
    {
        printf("Patient Queue is empty\n");
        return NULL;
    }
    int count=0;
    struct patient *curr=head,*prev=NULL,*nextnode=NULL;
    while(curr!=NULL&&count<k)
    {
        nextnode=curr->next;
        curr->next=prev;
        prev=curr; //after last loop,prev is new head of the reversed segment
        curr=nextnode;//after last loop, current is the head of the remaining unreversed part
        count++;
    }
    if(nextnode!=NULL)
        head->next=reverseInGroups(nextnode,k);
    return prev;
}
struct patient* mergeSortedQueues()
{
    int n1,n2;
    struct patient *head1=NULL,*head2=NULL,*temp;
    printf("General Department\n");
    printf("Enter the no. of patients in general queue: ");
    scanf("%d", &n1);
```

```
for(int i=1; i<=n1; i++)
{
   struct patient *newnode = (struct patient *)malloc(sizeof(struct patient));
   if(newnode==NULL)
   {
      printf("Can't allocate memory");
      break;
   }
   printf("Enter the name of patient %d: ", i);
   getchar();
   scanf("%[^\n]", newnode->name);
   printf("Enter the ID: ");
   scanf("%d",&newnode->id );
   printf("Enter the priority number: ");
   scanf("%d", &newnode->priority);
   newnode->next=NULL;
   if(head1==NULL)
   {
      head1=newnode;
   }
   else
   {
      temp=head1;
      while(temp->next!=NULL)
      {
         temp=temp->next;
      }
      temp->next=newnode;
   }
}
printf("Emergency Department\n");
printf("Enter the no. of patients in emergency queue: ");
scanf("%d", &n2);
for(int i=1; i<=n2; i++)
{
   struct patient *newnode = (struct patient *)malloc(sizeof(struct patient));
   if(newnode==NULL)
   {
      printf("Can't allocate memory");
      break;
   }
   printf("Enter the name of patient %d: ", i);
   getchar();
   scanf("%[^\n]", newnode->name);
   printf("Enter the ID: ");
   scanf("%d",&newnode->id );
```

```
    printf("Enter the priority number: ");
    scanf("%d", &newnode->priority);
    newnode->next=NULL;
    if(head2==NULL)
    {
       head2=newnode;
    }
    else
    {
       temp=head2;
       while(temp->next!=NULL)
       {
          temp=temp->next;
       }
       temp->next=newnode;
    }
}
if(head1==NULL)
    return head2;
if(head2==NULL)
    return head1;
struct patient *merged=NULL,*temp1=head1,*temp2=head2,*tail=NULL;
if(temp1->priority<=temp2->priority)
{
    merged =temp1;
    temp1= temp1->next;
}
else
{
    merged = temp2;
    temp2 = temp2->next;
}
tail=merged;
while(temp1!=NULL||temp2!=NULL)
{
    if(temp1 == NULL)
    {
       tail->next = temp2;
       break;
    }
    else if(temp2 == NULL)
    {
       tail->next = temp1;
       break;
    }
    else if(temp1->priority<=temp2->priority)
```

```
        {
            tail->next =temp1;
            temp1= temp1->next;
        }
        else
        {
            tail->next = temp2;
            temp2 = temp2->next;
        }
        tail=tail->next;
    }
    return merged;
}
void searchAndMoveToFront(int id)
{
    struct patient *temp=head,*prev=NULL;
    if(head==NULL)
    {
        printf("Patient Queue is empty.\n");
        return;
    }
    while(temp!=NULL&&temp->id!=id)
    {
        prev=temp;
        temp=temp->next;
    }
    if(temp==NULL)
    {
        printf("Patient with %d id not found\n",id);
        return;
    }
    if(prev == NULL)//already at front
    {
        printf("Patient with ID %d is already at the front.\n", id);
        return;
    }
    prev->next=temp->next;
    temp->next=head;
    head=temp;
    printf("Patient with %d id found and moved to front\n",id);
}
int main() {
    int choice, id, k;
    struct patient *mergedQueue;

    do {
```

```
        printf("\n=== Hospital Patient Queue Menu ===\n");
        printf("1. Insert Patient at End\n");
        printf("2. Delete Patient by ID\n");
        printf("3. Display Queue\n");
        printf("4. Swap Highest Priority to Front\n");
        printf("5. Reverse Queue in Groups\n");
        printf("6. Merge Department Queues\n");
        printf("7. Search and Move Patient to Front\n");
        printf("0. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1: insertAtEnd(); break;
            case 2:
                printf("Enter ID to delete: ");
                scanf("%d", &id);
                deletePatientById(id);
                break;
            case 3: displayPatients(head); break;
            case 4: swapWithFront(); break;
            case 5:
                printf("Enter group size k: ");
                scanf("%d", &k);
                head = reverseInGroups(head, k);
                break;
            case 6:
                mergedQueue = mergeSortedQueues();
                printf("Merged Queue:\n");
                displayPatients(mergedQueue);
                //head = mergedQueue; // update global head
                break;
            case 7:
                printf("Enter ID to search and move to front: ");
                scanf("%d", &id);
                searchAndMoveToFront(id);
                break;
            case 0: printf("Exit\n"); break;
            default: printf("Invalid choice!\n");
        }
    } while(choice != 0);

    return 0;
}
```

```
//7)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct node
{
   char name[50];
   struct node *next;
   struct node *prev;
};
struct node *head=NULL,*tail=NULL;

void insert(int pos)
{
   struct node *newnode=(struct node*)malloc(sizeof(struct node));
   if(newnode==NULL)
   {
      printf("Cannot allocate memory\n");
      return;
   }
   printf("Input station name: ");
   getchar();
   scanf("%[^\n]",newnode->name);
   newnode->next=NULL;
   newnode->prev=NULL;
   if(head==NULL)
   {
      head=newnode;
      tail=newnode;
      return;
   }
   if(pos==1)
   {
      newnode->next=head;
      head->prev=newnode;
```

```
        head=newnode;
        return;
    }
    struct node *temp=head; int i=1;
    while(temp->next!=NULL&&i<pos-1)
    {
        temp=temp->next;
        i++;
    }
    if(temp->next==NULL)
    {
        temp->next=newnode;
        newnode->prev=temp;
        tail=newnode;
    }
    else
    {
        newnode->next=temp->next;
        newnode->prev=temp;
        temp->next->prev=newnode;
        temp->next=newnode;
    }
}
void deleteByName(char name[]) //to delete a station by name
{
    if(head==NULL)
    {
        printf("No stations in the route.\n");
        return;
    }
    struct node *temp=head;
    while(temp!=NULL&&strcmp(temp->name,name)!=0)
    {
        temp=temp->next;
```

```
    }
    if(temp==NULL)
    {
        printf("Station %s not found\n",name);
        return;
    }
    if(temp->prev != NULL)
        temp->prev->next = temp->next;
    else
        head = temp->next; // deleting head

    if(temp->next != NULL)
        temp->next->prev = temp->prev;
    else
        tail = temp->prev; // deleting tail
    free(temp);
    printf("Station %s deleted\n",name);
}
void traverseForward() //traverse in forward directon
{
    struct node *temp=head;
    if(head==NULL)
    {
        printf("No stations found\n");
        return;
    }
    while(temp!=NULL)
    {

        printf("%s",temp->name);
        if(temp->next!=NULL)
            printf(" -> ");
        temp=temp->next;
    }
```

```
    printf("\n");
}
void traverseBackward() //traverse in backward directon
{
    struct node *temp=tail;
    if(tail==NULL)
    {
        printf("No stations found.\n");
        return;
    }
    while(temp!=NULL)
    {
        printf("%s",temp->name);
        if(temp->prev!=NULL)
            printf(" -> ");
        temp=temp->prev;
    }
    printf("\n");
}
void modifyStation(char name[]) //to modify station name
{
    if(head==NULL)
    {
        printf("No stations in the route.\n");
        return;
    }
    struct node *temp=head;
    while(temp!=NULL&&strcmp(temp->name,name)!=0)
    {
        temp=temp->next;
    }
    if(temp==NULL)
    {
        printf("Station %s not found\n",name);
```

```
        return;
    }
    printf("Input new station name: ");
    char n[50];
    getchar();
    scanf("%[^\n]",n);
    printf("Station name changed from %s to %s\n",temp->name,n);
    strcpy(temp->name, n);
}
void displayRoute(char start[],char end[]) //to display route between 2 given stations
{
    if(head==NULL)
    {
        printf("No stations found.\n");
        return;
    }
    int found1=0,found2=0;
    struct node *temp1=head,*temp2=head;
    while(temp1!=NULL)
    {
        if(strcmp(temp1->name, start) == 0)
        {
            found1 = 1;
            break;
        }
        temp1=temp1->next;
    }
    while(temp2!=NULL)
    {
        if(strcmp(temp2->name, end) == 0)
        {
            found2 = 1;
            break;
        }
```

```
    temp2=temp2->next;
}
if(!found1||!found2)
{
    printf("One or more stations not found\n");
    return;
}
struct node *temp=temp1;
int forward=0;
while(temp!=NULL)
{
    if(temp==temp2)
    {
        forward=1;
        break;
    }
    temp=temp->next;
}
if(forward)
{
    temp=temp1;
    while(temp!=NULL)
    {
        printf("%s",temp->name);
        if(temp==temp2)
            break;
        printf(" -> ");
        temp=temp->next;
    }
}
else
{
    temp=temp1;
    while(temp!=NULL)
```

```
        {
            printf("%s",temp->name);
            if(temp==temp2)
                break;
            printf(" -> ");
            temp=temp->prev;
        }
    }
    printf("\n");
}
int main()
{
    int choice, pos;
    char name[50], start[50], end[50];
    do
    {
        printf("\n=== Smart Metro Navigation Menu ===\n");
        printf("1. Insert Station\n");
        printf("2. Delete Station by Name\n");
        printf("3. Traverse Forward\n");
        printf("4. Traverse Backward\n");
        printf("5. Modify Station Name\n");
        printf("6. Display Route Between Two Stations\n");
        printf("0. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter position to insert station: ");
                scanf("%d", &pos);
                insert(pos);
                break;
            case 2:
```

```
            printf("Enter station name to delete: ");
            getchar();
            scanf("%[^\n]", name);
            deleteByName(name);
            break;
        case 3:
            printf("Route forward: \n");
            traverseForward();
            break;
        case 4:
            printf("Route backward: \n");
            traverseBackward();
            break;
        case 5:
            printf("Enter station name to modify: ");
            getchar();
            scanf("%[^\n]", name);
            modifyStation(name);
            break;
        case 6:
            printf("Enter start station: ");
            getchar();
            scanf("%[^\n]", start);
            printf("Enter end station: ");
            getchar();
            scanf("%[^\n]", end);
            displayRoute(start, end);
            break;
        case 0:
            printf("Exit.\n");
            break;
        default:
            printf("Invalid choice!\n");
    }
```

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

    return 0;
}
```

```
//8)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Task {
    char name[50];
    struct Task *next;
};
struct Task *head = NULL;
struct Task* createTask(char name[]) {
    struct Task *newNode = (struct Task*)malloc(sizeof(struct Task));
    strcpy(newNode->name, name);
    newNode->next = NULL;
    return newNode;
}
void insertTask(char name[]) {
    struct Task *newNode = createTask(name);

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
        return;
    }

    struct Task *temp = head;
    while (temp->next != head)
        temp = temp->next;
    temp->next = newNode;
    newNode->next = head;
}

void deleteTask(char name[]) {
    if (head == NULL) {
        printf("No tasks to delete!\n");
        return;
    }

    struct Task *curr = head, *prev = NULL;


    if (strcmp(head->name, name) == 0) {
        struct Task *last = head;
        while (last->next != head)
            last = last->next;
```

```
        if (head->next == head) {
            free(head);
            head = NULL;
        } else {
            last->next = head->next;
            free(head);
            head = last->next;
        }
        printf("Task '%s' deleted.\n", name);
        return;
    }

    do {
        prev = curr;
        curr = curr->next;
        if (strcmp(curr->name, name) == 0) {
            prev->next = curr->next;
            free(curr);
            printf("Task '%s' deleted.\n", name);
            return;
        }
    } while (curr != head);

    printf("Task '%s' not found!\n", name);
}

void displayTasks(struct Task *start) {
    if (start == NULL) {
        printf("No tasks!\n");
        return;
    }
    struct Task *temp = start;
    do {
        printf("%s", temp->name);
        temp = temp->next;
        if (temp != start) printf(" -> ");
    } while (temp != start);
    printf(" -> (back to %s)\n", start->name);
}


void splitList(struct Task *head, struct Task **head1, struct Task **head2) {
    if (head == NULL || head->next == head) {
        *head1 = head;
        *head2 = NULL;
        return;
```

```
    }

    struct Task *slow = head, *fast = head;

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

    if (fast->next->next == head)
        fast = fast->next;

    *head1 = head;
    *head2 = slow->next;


    slow->next = *head1;
    fast->next = *head2;
}


int main()
{
    int choice;
    char name[50];
    struct Task *P1 = NULL, *P2 = NULL;
    char ch;
    do
    {
        printf("\n--- Round Robin Task Scheduler ---\n");
        printf("1. Insert Task\n");
        printf("2. Delete Task\n");
        printf("3. Display All Tasks\n");
        printf("4. Split Task List\n");
        printf("5. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter task name: ");
                scanf("%s", name);
                insertTask(name);
                break;
            case 2:
                printf("Enter task name to delete: ");
```

```
            scanf("%s", name);
            deleteTask(name);
            break;
        case 3:
            displayTasks(head);
            break;
        case 4:
            splitList(head, &P1, &P2);
            printf("List P1: ");
            displayTasks(P1);
            printf("List P2: ");
            displayTasks(P2);
            break;
        case 5:
            printf("Exit\n");
            break;
        default:
            printf("Invalid choice!\n");
    }
  printf("Do you wish to continue(y/n):\n");
  scanf(" %c",&ch);
  }
  while(ch=='y'||ch=='Y');

  return 0;
}
```