1.Madhavi wants to generate an ordered sequence of numbers for creating a new PIN number for her debit card. So, she developed Program such a way that, she will enter some random numbers and arrange them in an order by using divide and conquer technique. Help the students to write a C program.

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

// Function to merge two subarrays
void merge(int arr[], int l, int m, int r) {
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;

   // Create temporary arrays
   int L[n1], R[n2];

   // Copy data to temporary arrays L[] and R[]
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];

   // Merge the temporary arrays back into arr[l..r]
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = l; // Initial index of merged subarray
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      } else {
         arr[k] = R[j];
         j++;
      }
      k++;
   }

   // Copy the remaining elements of L[], if there are any
   while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
   }

   // Copy the remaining elements of R[], if there are any
   while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
   }
}

// Main function to perform merge sort
void mergeSort(int arr[], int l, int r) {
   if (l < r) {
```

```
        // Same as (l+r)/2, but avoids overflow for large l and r
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

// Function to print an array
void printArray(int A[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

// Driver program to test the functions
int main() {
    int n;

    // Get the number of elements from the user
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));

    // Get the elements from the user
    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    // Perform merge sort
    mergeSort(arr, 0, n - 1);

    // Display the sorted array
    printf("Sorted array: ");
    printArray(arr, n);

    // Free the allocated memory
    free(arr);

    return 0;
}
```

2.Tribhuvana wants to search for a key from the key list in the basket. Help Tribhuvana to write a c program to search the key.

```
#include <stdio.h>

// Function to perform linear search
int searchKey(int keyList[], int n, int key) {
```

```
    for (int i = 0; i < n; i++) {
        if (keyList[i] == key) {
            return i;  // Key found, return its index
        }
    }
    return -1;  // Key not found
}

int main() {
    // Example key list
    int keyList[] = {2, 5, 8, 12, 16, 23, 38, 42, 55, 72};
    int n = sizeof(keyList) / sizeof(keyList[0]);

    // Key to search
    int keyToSearch;

    // Get the key from the user
    printf("Enter the key to search: ");
    scanf("%d", &keyToSearch);

    // Perform the search
    int index = searchKey(keyList, n, keyToSearch);

    // Display the result
    if (index != -1) {
        printf("Key %d found at index %d.\n", keyToSearch, index);
    } else {
        printf("Key %d not found in the list.\n", keyToSearch);
    }

    return 0;
}
```

(Or)

```
#include <stdio.h>

// Function to perform linear search
int linearSearch(int key, int arr[], int size) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // Return the index if key is found
        }
    }
    return -1; // Return -1 if key is not found
}

int main() {
    int key;

    // Sample key list
    int keyList[] = {10, 23, 5, 17, 8, 12, 15, 21};
    int size = sizeof(keyList) / sizeof(keyList[0]);

    // Ask Tribhuvana to enter the key to search
    printf("Enter the key to search: ");
```

```
    scanf("%d", &key);

    // Perform linear search
    int index = linearSearch(key, keyList, size);

    // Display the result
    if (index != -1) {
        printf("Key found at index %d\n", index);
    } else {
        printf("Key not found in the list\n");
    }

    return 0;
}
#include <stdio.h>

// Function to perform linear search
int linearSearch(int key, int arr[], int size) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // Return the index if key is found
        }
    }
    return -1; // Return -1 if key is not found
}

int main() {
    int key;

    // Sample key list
    int keyList[] = {10, 23, 5, 17, 8, 12, 15, 21};
    int size = sizeof(keyList) / sizeof(keyList[0]);

    // Ask Tribhuvana to enter the key to search
    printf("Enter the key to search: ");
    scanf("%d", &key);

    // Perform linear search
    int index = linearSearch(key, keyList, size);

    // Display the result
    if (index != -1) {
        printf("Key found at index %d\n", index);
    } else {
        printf("Key not found in the list\n");
    }

    return 0;
}
```

3.Divya wants to arrange the set of keys (5,4,6,7,1,2,3,8,9) in order by using **INSERTION SORT** TECHNIQUE. Help the students to write a C program.

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
```

```
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;

      // Move elements of arr[0..i-1] that are greater than key to one position ahead of their current
position
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}

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

int main() {
   int keys[] = {5, 4, 6, 7, 1, 2, 3, 8, 9};
   int n = sizeof(keys) / sizeof(keys[0]);

   printf("Original array: ");
   printArray(keys, n);

   insertionSort(keys, n);

   printf("Sorted array: ");
   printArray(keys, n);

   return 0;
}
```

(or)

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;

      // Move elements of arr[0..i-1] that are greater than key
      // to one position ahead of their current position
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}
```

```
void printArray(int arr[], int size) {
   int i;
   for (i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

int main() {
   int arr[] = {5, 4, 6, 7, 1, 2, 3, 8, 9};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Original array: ");
   printArray(arr, n);

   insertionSort(arr, n);

   printf("Sorted array: ");
   printArray(arr, n);

   return 0;
}
```

4.The list of 'n' numbers is given to the student of class X by the Maths teacher. And another 'n' numbers are given to another student. Now they both have to merge the given numbers which were given by the Maths teacher. After merging the two set of list elements, then they have to write those merge elements in the sorted order. Help the students to write a C program.

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

// Function to merge two arrays and return a new array
int* mergeArrays(int arr1[], int arr2[], int n) {
   int* mergedArray = (int*)malloc(2 * n * sizeof(int));
   int i = 0, j = 0, k = 0;

   // Merge arrays until one of them is exhausted
   while (i < n && j < n) {
      if (arr1[i] < arr2[j]) {
         mergedArray[k++] = arr1[i++];
      } else {
         mergedArray[k++] = arr2[j++];
      }
   }

   // Copy the remaining elements from both arrays, if any
   while (i < n) {
      mergedArray[k++] = arr1[i++];
   }

   while (j < n) {
      mergedArray[k++] = arr2[j++];
   }

   return mergedArray;
```

```
}

// Function to print an array
void printArray(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}

int main() {
   int n;

   // Input the size of the arrays
   printf("Enter the size of the arrays: ");
   scanf("%d", &n);

   int arr1[n], arr2[n];

   // Input elements for the first array
   printf("Enter %d elements for the first array:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr1[i]);
   }

   // Input elements for the second array
   printf("Enter %d elements for the second array:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr2[i]);
   }

   // Merge the arrays
   int* mergedArray = mergeArrays(arr1, arr2, n);

   // Sort the merged array
   for (int i = 0; i < 2 * n - 1; i++) {
      for (int j = 0; j < 2 * n - i - 1; j++) {
         if (mergedArray[j] > mergedArray[j + 1]) {
            // Swap elements if they are in the wrong order
            int temp = mergedArray[j];
            mergedArray[j] = mergedArray[j + 1];
            mergedArray[j + 1] = temp;
         }
      }
   }

   // Print the sorted merged array
   printf("Sorted Merged Array: ");
   printArray(mergedArray, 2 * n);

   // Free dynamically allocated memory
   free(mergedArray);

   return 0;
}
```

(or)

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

// Function to merge two arrays and return the merged array
int* mergeArrays(int arr1[], int arr2[], int n1, int n2, int* mergedSize) {
   *mergedSize = n1 + n2; // Size of the merged array
   int* mergedArr = (int*)malloc(*mergedSize * sizeof(int));

   int i = 0, j = 0, k = 0;

   // Merge until one of the arrays is exhausted
   while (i < n1 && j < n2) {
      if (arr1[i] < arr2[j]) {
         mergedArr[k++] = arr1[i++];
      } else {
         mergedArr[k++] = arr2[j++];
      }
   }

   // Copy the remaining elements from the first array, if any
   while (i < n1) {
      mergedArr[k++] = arr1[i++];
   }

   // Copy the remaining elements from the second array, if any
   while (j < n2) {
      mergedArr[k++] = arr2[j++];
   }

   return mergedArr;
}

// Function to perform a bubble sort on an array
void bubbleSort(int arr[], int size) {
   for (int i = 0; i < size - 1; i++) {
      for (int j = 0; j < size - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            // Swap the elements if they are in the wrong order
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}

int main() {
   int n;

   // Input the size of the arrays
   printf("Enter the size of the arrays: ");
   scanf("%d", &n);

   int arr1[n], arr2[n];

   // Input the elements of the first array
   printf("Enter %d elements for the first array:\n", n);
```

```
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr1[i]);
    }

    // Input the elements of the second array
    printf("Enter %d elements for the second array:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr2[i]);
    }

    int mergedSize;
    int* mergedArr = mergeArrays(arr1, arr2, n, n, &mergedSize);

    // Sort the merged array
    bubbleSort(mergedArr, mergedSize);

    // Display the sorted merged array
    printf("Sorted Merged Array:\n");
    for (int i = 0; i < mergedSize; i++) {
        printf("%d ", mergedArr[i]);
    }

    // Free dynamically allocated memory
    free(mergedArr);

    return 0;
}
```

5.The list of 'n' keys is given to the student of class 7th. The student wants to sort the set of keys by placing them in a separate bucket. Help the students to write a C program.

```
#include <stdio.h>

// Function to perform bucket sort
void bucketSort(int arr[], int n) {
    // Find the maximum and minimum values in the array
    int max_val = arr[0];
    int min_val = arr[0];

    for (int i = 1; i < n; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
        if (arr[i] < min_val) {
            min_val = arr[i];
        }
    }

    // Calculate the range of values
    int range = max_val - min_val + 1;

    // Create buckets
    int buckets[range];
    for (int i = 0; i < range; i++) {
        buckets[i] = 0;
```

```
    }

    // Count the number of occurrences of each value in the input array
    for (int i = 0; i < n; i++) {
        buckets[arr[i] - min_val]++;
    }

    // Place the values back into the array in sorted order
    int index = 0;
    for (int i = 0; i < range; i++) {
        while (buckets[i] > 0) {
            arr[index++] = i + min_val;
            buckets[i]--;
        }
    }
}

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

// Driver program
int main() {
    int n;

    // Input the number of keys
    printf("Enter the number of keys: ");
    scanf("%d", &n);

    int keys[n];

    // Input the keys
    printf("Enter the keys:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &keys[i]);
    }

    // Perform bucket sort
    bucketSort(keys, n);

    // Print the sorted array
    printf("Sorted keys: ");
    printArray(keys, n);

    return 0;
}
```

(or)

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

// Function to perform bucket sort
void bucketSort(int keys[], int n) {
```

```
    // Find the maximum and minimum values in the array
    int max = keys[0], min = keys[0];
    for (int i = 1; i < n; i++) {
        if (keys[i] > max) {
            max = keys[i];
        }
        if (keys[i] < min) {
            min = keys[i];
        }
    }

    // Create buckets
    int range = max - min + 1;
    int* buckets = (int*)malloc(range * sizeof(int));

    // Initialize buckets
    for (int i = 0; i < range; i++) {
        buckets[i] = 0;
    }

    // Count the occurrences of each key in the array
    for (int i = 0; i < n; i++) {
        buckets[keys[i] - min]++;
    }

    // Place the keys back into the array in sorted order
    int index = 0;
    for (int i = 0; i < range; i++) {
        while (buckets[i] > 0) {
            keys[index++] = i + min;
            buckets[i]--;
        }
    }

    // Free dynamically allocated memory
    free(buckets);
}

// Function to print the sorted keys
void printSortedKeys(int keys[], int n) {
    printf("Sorted keys: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", keys[i]);
    }
    printf("\n");
}

int main() {
    int n;

    // Get the number of keys
    printf("Enter the number of keys: ");
    scanf("%d", &n);

    // Allocate memory for the keys
    int* keys = (int*)malloc(n * sizeof(int));
```

```
    // Get the keys from the user
    printf("Enter the keys: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &keys[i]);
    }

    // Perform bucket sort
    bucketSort(keys, n);

    // Print the sorted keys
    printSortedKeys(keys, n);

    // Free dynamically allocated memory
    free(keys);

    return 0;
}
```

6.Ravi wants to create a data structure to store elements in a particular order. If the seven elements A, B, C, D, E, F and G are pushed into a stack in reverse order, i.e., starting from G. The stack is popped five times, what is the current stack status (using Arrays). Help the Ravi to write a C program.

```
#include <stdio.h>

#define MAX_SIZE 7

// Stack structure
struct Stack {
    int arr[MAX_SIZE];
    int top;
};

// Function to initialize the stack
void initialize(struct Stack* stack) {
    stack->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack* stack) {
    return stack->top == MAX_SIZE - 1;
}

// Function to push an element onto the stack
void push(struct Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack Overflow\n");
        return;
    }
    stack->arr[++stack->top] = value;
}
```

```
// Function to pop an element from the stack
int pop(struct Stack* stack) {
   if (isEmpty(stack)) {
      printf("Stack Underflow\n");
      return -1; // Return a special value indicating underflow
   }
   return stack->arr[stack->top--];
}

// Function to display the current stack status
void display(struct Stack* stack) {
   printf("Current Stack Status: ");
   for (int i = 0; i <= stack->top; ++i) {
      printf("%c ", stack->arr[i]);
   }
   printf("\n");
}

int main() {
   struct Stack stack;
   initialize(&stack);

   // Pushing elements in reverse order
   push(&stack, 'G');
   push(&stack, 'F');
   push(&stack, 'E');
   push(&stack, 'D');
   push(&stack, 'C');
   push(&stack, 'B');
   push(&stack, 'A');

   // Popping five elements and displaying the stack status after each pop
   for (int i = 0; i < 5; ++i) {
      int popped = pop(&stack);
      if (popped != -1) {
         printf("Popped: %c\n", popped);
         display(&stack);
      }
   }

   return 0;
}
```

(or)

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

#define MAX_SIZE 7

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

// Function to initialize the stack
void initialize(struct Stack *s) {
   s->top = -1;
```

```
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
   return s->top == -1;
}

// Function to push an element onto the stack
void push(struct Stack *s, char item) {
   if (s->top == MAX_SIZE - 1) {
      printf("Stack overflow\n");
      exit(EXIT_FAILURE);
   }
   s->items[++(s->top)] = item;
}

// Function to pop an element from the stack
char pop(struct Stack *s) {
   if (isEmpty(s)) {
      printf("Stack underflow\n");
      exit(EXIT_FAILURE);
   }
   return s->items[(s->top)--];
}

int main() {
   struct Stack stack;
   initialize(&stack);

   // Pushing elements into the stack in reverse order
   push(&stack, 'G');
   push(&stack, 'F');
   push(&stack, 'E');
   push(&stack, 'D');
   push(&stack, 'C');
   push(&stack, 'B');
   push(&stack, 'A');

   // Popping five elements from the stack
   for (int i = 0; i < 5; i++) {
      pop(&stack);
   }

   // Displaying the current status of the stack
   printf("Current stack status after popping five elements:\n");
   while (!isEmpty(&stack)) {
      printf("%c ", pop(&stack));
   }

   return 0;
}
```

7.Mrs. Aruna is a math's teacher; she wants to teach how to evaluate an expression by using precedency table to her students. Write a C Program to convert an infix expression to postfix expression.

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

#define MAX_SIZE 100

// Stack structure
struct Stack {
   int top;
   unsigned capacity;
   char* array;
};

// Stack functions
struct Stack* createStack(unsigned capacity) {
   struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
   stack->top = -1;
   stack->capacity = capacity;
   stack->array = (char*)malloc(stack->capacity * sizeof(char));
   return stack;
}

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

char peek(struct Stack* stack) {
   return stack->array[stack->top];
}

void push(struct Stack* stack, char op) {
   stack->array[++stack->top] = op;
}

char pop(struct Stack* stack) {
   if (!isEmpty(stack)) {
      return stack->array[stack->top--];
   }
   return '$'; // Dummy value for empty stack
}

// Function to check if a character is an operator
int isOperator(char ch) {
   return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

// Function to get the precedence of an operator
int getPrecedence(char op) {
   if (op == '+' || op == '-') {
      return 1;
   } else if (op == '*' || op == '/') {
      return 2;
   }
   return 0;
}

// Function to convert infix expression to postfix expression
```

```
void infixToPostfix(char* infix) {
   struct Stack* stack = createStack(MAX_SIZE);
   int i, j;

   for (i = 0, j = -1; infix[i]; ++i) {
      if (isalnum(infix[i])) {
         // If the current character is an operand, add it to the output
         infix[++j] = infix[i];
      } else if (infix[i] == '(') {
         // If the current character is an open parenthesis, push it onto the stack
         push(stack, infix[i]);
      } else if (infix[i] == ')') {
         // If the current character is a close parenthesis, pop and output from the stack until an open
parenthesis is encountered
         while (!isEmpty(stack) && peek(stack) != '(') {
            infix[++j] = pop(stack);
         }
         pop(stack); // Pop the open parenthesis from the stack
      } else {
         // If the current character is an operator, pop and output operators from the stack until an
operator with lower precedence is encountered
         while (!isEmpty(stack) && getPrecedence(infix[i]) <= getPrecedence(peek(stack))) {
            infix[++j] = pop(stack);
         }
         push(stack, infix[i]); // Push the current operator onto the stack
      }
   }

   // Pop and output the remaining operators from the stack
   while (!isEmpty(stack)) {
      infix[++j] = pop(stack);
   }

   infix[++j] = '\0'; // Null-terminate the postfix expression
}

int main() {
   char infix[MAX_SIZE];

   // Get the infix expression from the user
   printf("Enter infix expression: ");
   fgets(infix, sizeof(infix), stdin);
   infix[strcspn(infix, "\n")] = '\0'; // Remove the newline character from the input

   // Convert infix to postfix
   infixToPostfix(infix);

   // Display the postfix expression
   printf("Postfix expression: %s\n", infix);

   return 0;
}
```

(or)

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

```
#define MAX_SIZE 100

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

// Function prototypes
void initialize(Stack *s);
int isEmpty(Stack *s);
int isFull(Stack *s);
void push(Stack *s, char value);
char pop(Stack *s);
char peek(Stack *s);
int isOperand(char ch);
int getPrecedence(char ch);

// Function to convert infix expression to postfix expression
void infixToPostfix(char infix[], char postfix[]);

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

   printf("Enter the infix expression: ");
   scanf("%s", infix);

   infixToPostfix(infix, postfix);

   printf("Postfix expression: %s\n", postfix);

   return 0;
}

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 value) {
   if (isFull(s)) {
      printf("Stack overflow\n");
      exit(EXIT_FAILURE);
   }
   s->items[++s->top] = value;
}

char pop(Stack *s) {
```

```
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    }
    return s->items[s->top--];
}

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

int isOperand(char ch) {
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

int getPrecedence(char ch) {
    switch (ch) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return -1;
    }
}

void infixToPostfix(char infix[], char postfix[]) {
    Stack operatorStack;
    initialize(&operatorStack);

    int i, j;
    i = j = 0;

    while (infix[i] != '\0') {
        char currentChar = infix[i];

        if (isOperand(currentChar)) {
            postfix[j++] = currentChar;
        } else if (currentChar == '(') {
            push(&operatorStack, currentChar);
        } else if (currentChar == ')') {
            while (!isEmpty(&operatorStack) && peek(&operatorStack) != '(') {
                postfix[j++] = pop(&operatorStack);
            }
            pop(&operatorStack); // Pop '('
        } else {
            while (!isEmpty(&operatorStack) && getPrecedence(currentChar) <=
getPrecedence(peek(&operatorStack))) {
                postfix[j++] = pop(&operatorStack);
```

```
            }
            push(&operatorStack, currentChar);
        }

        i++;
    }

    // Pop remaining operators from the stack
    while (!isEmpty(&operatorStack)) {
        postfix[j++] = pop(&operatorStack);
    }

    postfix[j] = '\0'; // Null-terminate the postfix expression
}
```

8.Mr. John want to create an audio play list in his mobile. Now, he wishes to add N songs into the list with song numbers from 1 to N in sequential order. Write a C program to implement above scenario using queue (Arrays)?

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

#define MAX_SIZE 100

// Structure to represent a queue
struct Queue {
    int front, rear, size;
    unsigned capacity;
    int* array;
};

// Function to create a new queue
struct Queue* createQueue(unsigned capacity) {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;
    queue->array = (int*)malloc(queue->capacity * sizeof(int));
    return queue;
}

// Function to check if the queue is full
int isFull(struct Queue* queue) {
    return (queue->size == queue->capacity);
}

// Function to check if the queue is empty
int isEmpty(struct Queue* queue) {
    return (queue->size == 0);
}

// Function to add a song to the playlist
void enqueue(struct Queue* queue, int song) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot add more songs.\n");
        return;
    }
```

```
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = song;
    queue->size++;
    printf("Song %d added to the playlist.\n", song);
}

// Function to remove a song from the playlist
int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot remove a song.\n");
        return -1; // indicating failure
    }
    int song = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size--;
    return song;
}

// Function to display the current playlist
void display(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Playlist is empty.\n");
        return;
    }
    printf("Current Playlist: ");
    int i;
    for (i = 0; i < queue->size; i++) {
        printf("%d ", queue->array[(queue->front + i) % queue->capacity]);
    }
    printf("\n");
}

int main() {
    int N, i;
    printf("Enter the number of songs (N): ");
    scanf("%d", &N);

    struct Queue* playlist = createQueue(MAX_SIZE);

    // Adding songs to the playlist
    for (i = 1; i <= N; i++) {
        enqueue(playlist, i);
    }

    // Displaying the initial playlist
    display(playlist);

    // Removing songs from the playlist (dequeue)
    int removedSong = dequeue(playlist);
    if (removedSong != -1) {
        printf("Removed song from the playlist: %d\n", removedSong);
    }

    // Displaying the updated playlist
    display(playlist);

    // Freeing allocated memory
```

```
    free(playlist->array);
    free(playlist);

    return 0;
}
```

(or)

```
#include <stdio.h>

#define MAX_SIZE 100

// Structure to represent a queue
struct Queue {
    int front, rear, size;
    unsigned capacity;
    int* array;
};

// Function to create a queue with a given capacity
struct Queue* createQueue(unsigned capacity) {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;
    queue->array = (int*)malloc(queue->capacity * sizeof(int));
    return queue;
}

// Function to check if the queue is full
int isFull(struct Queue* queue) {
    return (queue->size == queue->capacity);
}

// Function to check if the queue is empty
int isEmpty(struct Queue* queue) {
    return (queue->size == 0);
}

// Function to add an item to the queue
void enqueue(struct Queue* queue, int item) {
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
}

// Function to print the contents of the queue
void printQueue(struct Queue* queue) {
    int i;
    for (i = 0; i < queue->size; i++)
        printf("%d ", queue->array[i]);
    printf("\n");
}

int main() {
    int N;
```

```
    printf("Enter the number of songs (N): ");
    scanf("%d", &N);

    struct Queue* playlist = createQueue(MAX_SIZE);

    // Enqueue songs from 1 to N in sequential order
    for (int i = 1; i <= N; i++) {
        enqueue(playlist, i);
    }

    // Display the playlist
    printf("The playlist is: ");
    printQueue(playlist);

    // Free allocated memory
    free(playlist->array);
    free(playlist);

    return 0;
}
```

9.Mr. Rajesh is CEO of an organization 'Software Solutions'. He wants to maintain the personal information of his employee's like Eage (Employee Age), Eno (Employee Number), and Salary(Employee Salary) using singly linked list. So, help him to store and organize the data.

```
#include <iostream>
using namespace std;

// Define the structure for an employee
struct Employee {
    int Eage;   // Employee Age
    int Eno;    // Employee Number
    float Salary; // Employee Salary
    Employee* next; // Pointer to the next employee in the list
};

// Function to add a new employee to the linked list
void addEmployee(Employee*& head, int Eage, int Eno, float Salary) {
    Employee* newEmployee = new Employee;
    newEmployee->Eage = Eage;
    newEmployee->Eno = Eno;
    newEmployee->Salary = Salary;
    newEmployee->next = head;
    head = newEmployee;
}

// Function to display all employees in the linked list
void displayEmployees(Employee* head) {
    cout << "Employee Information:\n";
    cout << "---------------------\n";
    Employee* current = head;
    while (current != nullptr) {
        cout << "Employee Age: " << current->Eage << ", Employee Number: " << current->Eno << ",
Salary: " << current->Salary << endl;
        current = current->next;
```

```
    }
    cout << endl;
}

// Function to delete all employees in the linked list (free memory)
void deleteEmployees(Employee*& head) {
    while (head != nullptr) {
        Employee* temp = head;
        head = head->next;
        delete temp;
    }
}

int main() {
    // Initialize an empty linked list
    Employee* head = nullptr;

    // Adding employees to the linked list
    addEmployee(head, 25, 101, 50000.0);
    addEmployee(head, 30, 102, 60000.0);
    addEmployee(head, 28, 103, 55000.0);

    // Displaying the employee information
    displayEmployees(head);

    // Deleting all employees and free memory
    deleteEmployees(head);

    return 0;
}
```

10.Mr.Kiran is CEO of an organization 'Software Solutions'. He wants to maintain the personal information of his employee's like Eage (Employee Age), Eno (Employee Number), and Salary(Employee Salary) using doubly linked list. So, help him to store and organize the data.

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

// Structure to represent an employee
struct Employee {
    int Eage;
    int Eno;
    float Salary;
    struct Employee *prev;
    struct Employee *next;
};

// Function to create a new employee node
struct Employee *createEmployee(int age, int eno, float salary) {
    struct Employee *newEmployee = (struct Employee *)malloc(sizeof(struct
Employee));
    newEmployee->Eage = age;
    newEmployee->Eno = eno;
    newEmployee->Salary = salary;
```

```
    newEmployee->prev = NULL;
    newEmployee->next = NULL;
    return newEmployee;
}

// Function to insert a new employee at the end of the list
void insertEmployeeAtEnd(struct Employee **head, int age, int eno, float salary) {
    struct Employee *newEmployee = createEmployee(age, eno, salary);

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

// Function to display all employees in the list
void displayEmployees(struct Employee *head) {
    printf("Employee Information:\n");
    printf("Eage\tEno\tSalary\n");
    while (head != NULL) {
        printf("%d\t%d\t%.2f\n", head->Eage, head->Eno, head->Salary);
        head = head->next;
    }
    printf("\n");
}

// Function to delete an employee with a specific Employee Number
void deleteEmployeeByNumber(struct Employee **head, int eno) {
    struct Employee *current = *head;
    while (current != NULL) {
        if (current->Eno == eno) {
            if (current->prev != NULL) {
                current->prev->next = current->next;
            } else {
                *head = current->next;
            }

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

            free(current);
            printf("Employee with Employee Number %d deleted.\n", eno);
            return;
```

```
        }
        current = current->next;
    }

    printf("Employee with Employee Number %d not found.\n", eno);
}

// Function to free the memory allocated for the list
void freeList(struct Employee **head) {
    struct Employee *current = *head;
    while (current != NULL) {
        struct Employee *temp = current;
        current = current->next;
        free(temp);
    }
    *head = NULL;
}

int main() {
    struct Employee *employeeList = NULL;

    // Insert employees
    insertEmployeeAtEnd(&employeeList, 25, 101, 50000.0);
    insertEmployeeAtEnd(&employeeList, 30, 102, 60000.0);
    insertEmployeeAtEnd(&employeeList, 28, 103, 55000.0);

    // Display all employees
    displayEmployees(employeeList);

    // Delete employee with Employee Number 102
    deleteEmployeeByNumber(&employeeList, 102);

    // Display employees after deletion
    displayEmployees(employeeList);

    // Free the memory allocated for the list
    freeList(&employeeList);

    return 0;
}
```

11.Mr. Ishan is very much interested to know about his ancestors and he want to store their data in a hierarchical manner. Help Ishan to create a data structure to store the data in hierarchical manner and traverse it in preorder, postorder and inorder.

```
class TreeNode:
   def __init__(self, data):
      self.data = data
      self.children = []
```

```
def preorder_traversal(node):
    if node is not None:
        print(node.data, end=" ")
        for child in node.children:
            preorder_traversal(child)

def postorder_traversal(node):
    if node is not None:
        for child in node.children:
            postorder_traversal(child)
        print(node.data, end=" ")

def inorder_traversal(node):
    if node is not None:
        if len(node.children) > 0:
            inorder_traversal(node.children[0])
        print(node.data, end=" ")
        if len(node.children) > 1:
            inorder_traversal(node.children[1])

# Example usage:
root = TreeNode("Ishan")

# Adding ancestors
root.children.append(TreeNode("Parent1"))
root.children[0].children.append(TreeNode("Grandparent1"))
root.children[0].children.append(TreeNode("Grandparent2"))
root.children.append(TreeNode("Parent2"))

# Performing traversals
print("Preorder Traversal:")
preorder_traversal(root)
print("\n")

print("Postorder Traversal:")
postorder_traversal(root)
print("\n")

print("Inorder Traversal:")
inorder_traversal(root)
```

12.An Adjacency Matrix is used for representing a graph. Given the adjacency Matrix and number of vertices and edges of a graph, the task is to visit the graph using Breath First Traversal

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

#define MAX_VERTICES 100

// Function to perform Breadth-First Traversal
```

```
void bfs(int adjacencyMatrix[MAX_VERTICES][MAX_VERTICES], int visited[MAX_VERTICES], int
vertices, int start) {
   // Queue for BFS
   int queue[MAX_VERTICES];
   int front = -1, rear = -1;

   // Enqueue the starting vertex
   queue[++rear] = start;
   visited[start] = 1;

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

      // Visit all adjacent vertices of the current vertex
      for (int i = 0; i < vertices; i++) {
         if (adjacencyMatrix[currentVertex][i] == 1 && !visited[i]) {
            queue[++rear] = i;
            visited[i] = 1;
         }
      }
   }
}

int main() {
   int vertices, edges;

   // Input: Number of vertices and edges
   printf("Enter the number of vertices: ");
   scanf("%d", &vertices);

   printf("Enter the number of edges: ");
   scanf("%d", &edges);

   // Input: Adjacency Matrix
   int adjacencyMatrix[MAX_VERTICES][MAX_VERTICES];
   printf("Enter the adjacency matrix:\n");

   for (int i = 0; i < vertices; i++) {
      for (int j = 0; j < vertices; j++) {
         scanf("%d", &adjacencyMatrix[i][j]);
      }
   }

   // Visited array to keep track of visited vertices
   int visited[MAX_VERTICES] = {0};

   // Input: Starting vertex for BFS
   int startVertex;
   printf("Enter the starting vertex for BFS: ");
   scanf("%d", &startVertex);

   // Perform BFS and print the traversal
   printf("BFS Traversal starting from vertex %d: ", startVertex);
   bfs(adjacencyMatrix, visited, vertices, startVertex);
```

```
    return 0;
}
```

13.An Adjacency Matrix is used for representing a graph. Given the adjacency Matrix and number of vertices and edges of a graph, the task is to visit the graph using Depth First Traversal.

```
#include <stdio.h>

#define MAX_VERTICES 100

int visited[MAX_VERTICES];

void dfs(int graph[MAX_VERTICES][MAX_VERTICES], int vertex, int numVertices) {
    printf("%d ", vertex);
    visited[vertex] = 1;

    for (int i = 0; i < numVertices; ++i) {
        if (graph[vertex][i] == 1 && !visited[i]) {
            dfs(graph, i, numVertices);
        }
    }
}

int main() {
    int numVertices, numEdges;

    printf("Enter the number of vertices and edges: ");
    scanf("%d %d", &numVertices, &numEdges);

    int graph[MAX_VERTICES][MAX_VERTICES];

    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {
            scanf("%d", &graph[i][j]);
        }
    }

    for (int i = 0; i < numVertices; ++i) {
        visited[i] = 0;
    }

    printf("Depth First Traversal starting from vertex 0: ");
    dfs(graph, 0, numVertices);

    return 0;
}
```

14.Mr. Rakesh, Professor stores the students grade list for a subject ‘Y’ which contains student_id, grade based on sorted order of student_id.
Mr. Rakesh plans to write an optimal search application to search the grade for a student having an ID as s_id and print it onto the console, if s_id does not exist print “RECORD NOT FOUND”. How did he achieve it?

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

```
// Function to perform binary search
int binarySearch(int student_ids[], int grades[], int n, int s_id) {
   int low = 0, high = n - 1;

   while (low <= high) {
      int mid = low + (high - low) / 2;

      if (student_ids[mid] == s_id) {
         // Student ID found, return the corresponding grade
         return grades[mid];
      } else if (student_ids[mid] < s_id) {
         // If s_id is greater, ignore the left half
         low = mid + 1;
      } else {
         // If s_id is smaller, ignore the right half
         high = mid - 1;
      }
   }

   // If control reaches here, s_id is not in the list
   return -1;  // Return -1 to indicate "RECORD NOT FOUND"
}

int main() {
   // Assuming the list is pre-sorted
   int student_ids[] = {101, 102, 103, 104, 105};
   int grades[] = {90, 85, 92, 88, 95};
   int n = sizeof(student_ids) / sizeof(student_ids[0]);

   // Example student ID to search
   int s_id = 103;

   // Perform binary search
   int grade = binarySearch(student_ids, grades, n, s_id);

   // Display the result
   if (grade != -1) {
      printf("Grade for student ID %d: %d\n", s_id, grade);
   } else {
      printf("RECORD NOT FOUND\n");
   }

   return 0;
}
```

15.Ms.Vidhya designed an application uses a doubly linked list to store a list of integer data in which a node can be deleted from any position given that position ‘P’ is always less than or equal to the total number of nodes within the list. Also, the application offers search utility which returns the address of the node if searched otherwise NULL. Write a program to implement the deletion and search function over the list.

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

```
// Node structure for doubly linked list
struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
      printf("Memory allocation failed.\n");
      exit(EXIT_FAILURE);
   }
   newNode->data = data;
   newNode->prev = NULL;
   newNode->next = NULL;
   return newNode;
}

// Function to insert a node at the end of the list
void insertEnd(struct Node** head, int data) {
   struct Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode;
   } else {
      struct Node* temp = *head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
      newNode->prev = temp;
   }
}

// Function to delete a node at a given position
void deleteNode(struct Node** head, int position) {
   if (*head == NULL) {
      printf("List is empty.\n");
      return;
   }

   if (position <= 0) {
      printf("Invalid position.\n");
      return;
   }

   struct Node* current = *head;

   // Traverse to the node at the specified position
   for (int i = 1; i < position && current != NULL; i++) {
      current = current->next;
   }

   if (current == NULL) {
      printf("Position out of bounds.\n");
      return;
```

```
    }

    // Adjust pointers to skip the current node
    if (current->prev != NULL) {
        current->prev->next = current->next;
    } else {
        *head = current->next;
    }

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

    // Free the memory of the deleted node
    free(current);
}

// Function to search for a node with a given value
struct Node* searchNode(struct Node* head, int key) {
    while (head != NULL) {
        if (head->data == key) {
            return head; // Node found
        }
        head = head->next;
    }
    return NULL; // Node not found
}

// Function to display the doubly linked list
void displayList(struct Node* head) {
    while (head != NULL) {
        printf("%d <-> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

// Function to free the memory allocated for the list
void freeList(struct Node** head) {
    struct Node* current = *head;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }

    *head = NULL;
}

int main() {
    struct Node* head = NULL;

    insertEnd(&head, 10);
    insertEnd(&head, 20);
    insertEnd(&head, 30);
```

```
    insertEnd(&head, 40);

    printf("Original list: ");
    displayList(head);

    // Example of search
    int searchValue = 30;
    struct Node* searchResult = searchNode(head, searchValue);
    if (searchResult != NULL) {
        printf("Node with value %d found at address %p\n", searchValue, (void*)searchResult);
    } else {
        printf("Node with value %d not found\n", searchValue);
    }

    // Example of deletion at position
    int positionToDelete = 2;
    deleteNode(&head, positionToDelete);

    printf("List after deleting node at position %d: ", positionToDelete);
    displayList(head);

    // Free the memory allocated for the list
    freeList(&head);

    return 0;
}
```