Bubble Sort

```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Merge Sort :

```
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Insertion Sort

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

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;

        printf("\nArray after pass %d: \n", i); // Printing
array after each pass (iteration)
        printArray(arr, n);
    }
}
```

Selection Sort

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n - 1; i++)
    {
        min_idx = i;
        for (j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[min_idx])
            {
                min_idx = j;
            }
        }

        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;

        printf("\nArray after pass %d: \n", i + 1); // Printing
array after each pass (iteration
        printArray(arr, n);
    }
}
```

```
// 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);
    }
}
```

Infix to Prefix :

```
int precedence(char symbol)
{
    if (symbol == '*' || symbol == '/')
    {
        return 2;
    }
    else if (symbol == '+' || symbol == '-')
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

// Next Column —------>
```

```
void infixToPrefix(char infix[])
{
    int length = strlen(infix);
    char symbol;
    for (int i = length - 1; i >= 0; i--)
    {
        symbol = infix[i];
        if (isalnum(symbol))
        {
            printf("%c", symbol);
        }
        else if (symbol == ')')
        {
            push(symbol);
        }
        else if (symbol == '(')
        {
            while (stack[top] != ')')
            {
                printf("%c", pop());
            }
            pop();
        }
        else
        {
            while (precedence(symbol) <=
precedence(stack[top]) && top != -1)
            {
                printf("%c", pop());
            }
            push(symbol);
        }
    }
    while (top != -1)
    {
        printf("%c", pop());
    }
}
```

Quick Sort :

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

// Function to partition the array and return the pivot
index
int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // Pivot (last element)
    int i = (low - 1);      // Index of smaller element

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than or equal to
pivot
        if (arr[j] <= pivot)
        {
            i++; // Increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        // pi is partitioning index
        int pi = partition(arr, low, high);

        // Separately sort elements before partition and
after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Heap Sort :

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

// Function to heapify a subtree rooted with node i which
is an index in arr[]
void heapify(int arr[], int n, int i)
{
    int largest = i;   // Initialize largest as root
    int l = 2 * i + 1; // Left child
    int r = 2 * i + 2; // Right child

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to perform Heap Sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--)
    {
        // Move current root to end
        swap(&arr[0], &arr[i]);

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Prefix to Postfix :

```
#include <string.h>
#include <ctype.h>
#define MAX 100

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

void push(char item)
{
    if (top == (MAX - 1))
    {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = item;
}
char pop()
{
    if (top == -1)
    {
        printf("Stack Underflow\n");
        exit(1);
    }
    return stack[top--];
}
int isOperator(char symbol)
{
    if (symbol == '*' || symbol == '/' || symbol == '+' ||
symbol == '-')
    {
        return 1;
    }
    else
    {   return 0;   }

}
void prefixToPostfix(char prefix[])
{
    int length = strlen(prefix);
    char symbol, op1, op2;
    for (int i = length - 1; i >= 0; i--)
    {
        symbol = prefix[i];
        if (isOperator(symbol) == 0)
        {
            push(symbol);
        }
        else
        {
            op1 = pop();
            op2 = pop();
            char temp[3] = {op1, op2, symbol};
            char *tempPtr = temp;
            push(*tempPtr);
        }
    }
}
```

Infix to Postfix :

```
int precedence(char symbol)
{
    if (symbol == '*' || symbol == '/')
    {
        return 2;
    }
    else if (symbol == '+' || symbol == '-')
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void infixToPostfix(char infix[])
{
    int length = strlen(infix);
    char symbol;
    for (int i = 0; i < length; i++)
    {
        symbol = infix[i];
        if (isalnum(symbol))
        {
            printf("%c", symbol);
        }
        else if (symbol == '(')
        {
            push(symbol);
        }
        else if (symbol == ')')
        {
            while (stack[top] != '(')
            {
                printf("%c", pop());
            }
            pop();
        }
        else
        {
            while (precedence(stack[top]) >=
precedence(symbol))
            {
                printf("%c", pop());
            }
            push(symbol);
        }
    }
    while (top != -1)
    {
        printf("%c", pop());
    }
}
```

Binary Tree :

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

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

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

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

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

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

int search(struct Node *root, int data)
{
    if (root == NULL)
    {
        return 0;
    }
    else if (root->data == data)
    {
        return 1;
    }
    else if (data <= root->data)
    {
        return search(root->left, data);
    }
    else
    {
        return search(root->right, data);
    }
}

int findMin(struct Node *root)
{
    if (root == NULL)
    {
        printf("Error: Tree is empty\n");
        return -1;
    }
    else if (root->left == NULL)
    {
        return root->data;
    }
    return findMin(root->left);
}

int findMax(struct Node *root)
{
    if (root == NULL)
    {
        printf("Error: Tree is empty\n");
        return -1;
    }
    else if (root->right == NULL)
    {
        return root->data;
    }
    return findMax(root->right);
}
```

```
int findHeight(struct Node *root)
{
    if (root == NULL)
    {
        return -1;
    }
    int leftHeight = findHeight(root->left);
    int rightHeight = findHeight(root->right);
    return leftHeight > rightHeight ? leftHeight + 1 :
rightHeight + 1;
}
```

```
int findSize(struct Node *root)
{
    if (root == NULL)
    {
        return 0;
    }
    return findSize(root->left) + findSize(root->right) + 1;
}
```
