SORTING AND SEARCHING

Different sorting techniques:

- Bubble Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Insertion Sort
- Bucket Sort

BUBBLE SORT:

We compare two adjacent elements and swap them in the order we want. It's just like an air bubble rising to the water surface.

```c
#include <stdio.h>
//Function for the swapping elements using a temporary var
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
//Bubble sort implementation
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])
                swap(&arr[j], &arr[j + 1]);
    }
}
//Function to print sorted array
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
//Main Method
```

```c
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

QUICK SORT:

- Based on divide and conquer algorithm.
- Select a pivot element and divide array in two subparts such that elements on left of pivot are smaller than the pivot element and greater on right.
- Similar approach is used to divide the subarrays until each element is an independent subarray.
- On combining them they form sorted array.

MERGE SORT:

- It is also based on the principle of divide and conquer algorithm.
- The array is divided into two subparts somewhere from the middle element.
- Sort the two sub array (This step is done recursively until the sub array is left out with a single element i.e. base condition).
- Sorted subarrays are merged into single array.

Implementation of Merge sort and Quick sort:

```c
#include <stdio.h>
//Entering elements in the array
int read(int *A, int size)
{
    int l = 0;
    while (l < size)
    {
        scanf("%d", &A[l]);
        l++;
    }
}
// Function to swap position of elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Function to partition the Array on the basis of pivot element
int partition(int A[], int lb, int ub)
{
    int pivot = A[lb]; // Select the pivot element
    int start = lb, end = ub;
    while (start < end)
    {
        while (A[start] <= pivot)
            start++;
        while (A[end] > pivot)
            end--;
        if (start < end)
            swap(&A[start], &A[end]);
    }
    swap(&A[lb], &A[end]);
    return end;
}
//quicksort algorithm implementation
void quickSort(int A[], int lb, int ub)
{
    if (lb < ub)
    {
        int loc = partition(A, lb, ub); // Select pivot position and put all
the elements smaller than pivot on left and greater than pivot on right
        quickSort(A, lb, loc - 1); // Sort the elements on the left of pivot
        quickSort(A, loc + 1, ub); // Sort the elements on the right of pivot
    }
}
```

```c
void Merge(int A[], int lb, int mid, int ub)
{
    int i = lb, j = mid + 1, k = 0, B[ub - lb + 1], m;
    while (i <= mid && j <= ub)
    {
        if (A[i] <= A[j])
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
    }
    while (i <= mid)
    {
        B[k++] = A[i++];
    }
    while (j <= ub)
    {
        B[k++] = A[j++];
    }
    m = 0;
    for (i = lb; i <= ub; i++)
        A[i] = B[m++];
}
//mergesort algorithm implementation(step 2)
void MergeSort(int A[], int lb, int ub)
{
    if (lb < ub)
    {
        int mid = (lb + ub) / 2;
        MergeSort(A, lb, mid);
        MergeSort(A, mid + 1, ub);
        Merge(A, lb, mid, ub);
    }
}
// Function to print sorted array
void printArray(int A[], int size)
{
    for (int i = 0; i < size; ++i)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
// Driver code
int main()
{
    printf("\nPress 1 to sort array using QuickSort Technique.");
    printf("\nPress 2 to sort array using MergeSort Technique.");
    printf("\nPress 0 to exit.\n");
```

```
    while (1)
    {
        int ch, n;
        printf("\nEnter your choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 0:
        {
            exit(ch);
        }
        case 1:
        {
            printf("Enter the size of the array : ");
            scanf("%d", &n);
            int A[n];
            printf("-- Enter %d values: ", n);
            read(A, n);
            quickSort(A, 0, n - 1);
            printf("\nSorted Array after applying quick sort: ");
            printArray(A, n);
            break;
        }
        case 2:
        {
            printf("Enter the size of the array : ");
            scanf("%d", &n);
            int A[n];
            printf("-- Enter %d values: ", n);
            read(A, n);
            MergeSort(A, 0, n - 1);
            printf("\nSorted Array after applying Merge sort: ");
            printArray(A, n);
            break;
        }
        default:
        {
            printf("\nPress 1 or 2, or 0 to exit.");
            break;
        }
        }
    }
}
```

INSERTION SORT:

It is used to place an element in an already sorted array A.

Just as we insert a card in an already arranged deck of cards.

Implementation

```
void InsertionSort(int A[], int size)
{
    int i, j, item;
    for (i = 1; i < size; i++)
    { /* Insert the element in A[i] */
        item = A[i];
        for (j = i - 1; j >= 0; j--)
            if (item > A[j])
            { /* push elements down*/
                A[j + 1] = A[j];
                A[j] = item; /* can do this once finally also */
            }
            else
                break; /*inserted, exit loop */
    }
}
```

SEARCHING:

- Linear Search- It is a very simple searching algorithm that checks every element of the array starting from the front end until the desired element is found.

Implementation:

```c
#include <stdio.h>
int search(int array[], int n, int x) //linear search algorithm
{
    for (int i = 0; i < n; i++)
        if (array[i] == x)
            return i;
    return -1;
}
//main method
int main()
{
    int array[] = {2, 4, 0, 1, 9};
    int x = 1;
    int n = sizeof(array) / sizeof(array[0]);
    int val = search(array, n, x);
    if (val == -1)
    {
        printf("Element not found");
    }
    else
    {
        printf("Element  found");
    }
}
```

- Binary Search- It works on the principle of divide and conquer on a sorted array. We use two pointers High and Low and find the middle element. Then we look for the element on either right or left of the middle element using iterative or recursive technique.

Implementation:

```
//Binary search using recursive technique
int binarySearch(int data[], int low, int high, int num)
{
    if (low == high)
    {
        if (data[low] == num)
            return low;
        else
            return -1;
    }
    if (low < high)
    {
        int m = (low + high) / 2;
        if (data[m] >= num)
            return binarySearch(data, low, m, num);
        else
            return binarySearch(data, m + 1, high, num);
    }
}
```