EEO 101: DATA STRUCTURES

Worksheet – 7

Name: Narsing Sharma
Enrolment No.: 24113089
Department: Civil Engineering
Year: 2nd Year
Batch: B

1. **Write functions to implement following algorithms for sorting the given list of data**

InsertionSort( )

SelectionSort( )

BubblesSort( )

QuickSort( )

MergeSort( )

HeapSort( )

Compare the above for a given set of reasonably large set of sample data w.r.t. execution time and memory requirement.

CODE:-

InsertionSort( )

```
#include<iostream>
using namespace std ;
void insertionSort(int a[], int n){
    for(int i=1;i<n;i++){
        int key=a[i];
        int j=i-1;
        while(j>=0 && a[j]>key){
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=key;
    }
```

```
}
```

COMMENTS:-

- This algorithm builds the sorted array one element at a time**.**
- It assumes that the first part of the array is already sorted.
- The element `key` is picked from the unsorted part and inserted into the correct place in the sorted part**.**
- The inner `while` loop shifts elements greater than `key` one position forward.
- This continues until all elements are placed correctly.
- Best case**:** O(n) (already sorted), Worst case: O(n^2).
- **Example flow:**
`[9,5,1]` → pick 5 → shift 9 → `[5,9,1]` → pick 1 → shift both → `[1,5,9]`.

SelectionSort( )

```
void selectionSort(int a[], int n){
    for(int i=0;i<n-1;i++){
        int min=i;
        for(int j=i+1;j<n;j++){
            if(a[j]<a[min]) min=j;
        }
        int t=a[i];
        a[i]=a[min];
        a[min]=t;
    }
}
```

COMMENTS:-

- This algorithm selects the smallest element from the unsorted portion each time.
- The variable `min` keeps track of the index of the smallest element.
- After finding it, it swaps that element with the current position `i`.

- This process continues until the whole array is sorted.
- Number of comparisons: always n^2 (even if already sorted).
- It's simple but not efficient for large datasets.
- **Example:**

From `[9,5,1,4]`, smallest is 1 → swap with 9 → `[1,5,9,4]`, then next smallest 4 → `[1,4,9,5]`.

BubblesSort( )

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

COMMENTS:-

- This algorithm repeatedly compares adjacent elements and swaps them if they're in the wrong order.
- After every pass, the largest element "bubbles up" to the end.
- Outer loop controls the number of passes, inner loop performs the comparisons.
- After first pass: the biggest element is at the last position.
- After second pass: the second biggest is at second last, and so on.
- Best case: O(n) (with optimization), Worst case: $O(n^2)$.
- **Example:**

`[9,5,1]` → swap 9 & 5 → `[5,9,1]` → swap 9 & 1 → `[5,1,9]` → swap 5 & 1 → `[1,5,9]`.

QuickSort( )

```
int partition(int a[], int l, int h){
    int p=a[h];
    int i=l-1;
    for(int j=l;j<h;j++){
        if(a[j]<p){
            i++;
            int t=a[i];
            a[i]=a[j];
            a[j]=t;
        }
    }
    int t=a[i+1];
    a[i+1]=a[h];
    a[h]=t;
    return i+1;
}


void quickSort(int a[], int l, int h){
    if(l<h){
        int pi=partition(a,l,h);
        quickSort(a,l,pi-1);
        quickSort(a,pi+1,h);
    }
}
```

COMMENT:-

- Quick sort is a divide and conquer algorithm.
- `partition()` chooses a pivot element (here, the last element).
- It rearranges the array so that:
  - All elements less than pivot come before it.
  - All greater elements come after it.
- Then `quickSort()` recursively sorts the left and right subarrays.

- Very efficient on large data, average complexity O(n log n).
- Worst case: $O(n^2)$ (when array already sorted and pivot choice is bad).
- **Example:**

`[9,5,1,4]` pivot=4 → rearrange to `[1,4,9,5]` → recursively sort both halves.

MergeSort( )

```
void merge(int a[], int l, int m, int r){
    int n1=m-l+1,n2=r-m;
    int L[n1],R[n2];
    for(int i=0;i<n1;i++){
 L[i]=a[l+i];
}
    for(int j=0;j<n2;j++){
 R[j]=a[m+1+j];
}
    int i=0;
int j=0;
int k=l;
    while(i<n1 && j<n2){
        if(L[i]<=R[j]) a[k++]=L[i++];
        else a[k++]=R[j++];
    }
    while(i<n1) {
a[k++]=L[i++];}
    while(j<n2){
 a[k++]=R[j++];
}
}


void mergeSort(int a[], int l, int r){
    if(l<r){
        int m=(l+r)/2;
        mergeSort(a,l,m);
        mergeSort(a,m+1,r);
        merge(a,l,m,r);
    }
}
```

COMMENTS:-

- Another divide and conquer algorithm.
- The array is recursively divided into two halves until each part has one element.
- Then the merge() function combines these small sorted parts into a larger sorted array.
- The `L[]` and `R[]` temporary arrays store left and right subarrays.
- The elements are merged in sorted order using two pointers.
- Time complexity is O(n log n) in all cases.
- **Example:**
`[9,5,1,4]` → split into `[9,5]` and `[1,4]` → sort individually → merge into `[1,4,5,9]`.

HeapSort( )

```
void heapify(int a[], int n, int i){
    int big=i;
    int l=2*i+1;
    int r=2*i+2;
    if(l<n && a[l]>a[big]) big=l;
    if(r<n && a[r]>a[big]) big=r;
    if(big!=i){
        int t=a[i];
        a[i]=a[big];
        a[big]=t;
        heapify(a,n,big);
    }
}

void heapSort(int a[], int n){
    for(int i=n/2-1;i>=0;i--) {
      heapify(a,n,i);
}
    for(int i=n-1;i>0;i--){
        int t=a[0];
        a[0]=a[i];
        a[i]=t;
        heapify(a,i,0);
    }
}
```

COMMENTS:-

- Heap Sort works using a binary heap data structure (complete binary tree).
- First, it builds a max heap so that the largest element is at the root (`a[0]`).
- Then it swaps the first element (max) with the last one and **reduces heap size by 1**.
- After each swap, `heapify()` restores the heap property.
- This continues until the array is sorted.
- Time complexity is O(n log n) and uses no extra space.
- **Example:**

Build heap `[9,7,8,3]` → swap 9 with 3 → `[3,7,8,9]` → heapify again → `[3,7,8,9]` (sorted).

Main function()

<u>**Use Any Of The Function For Sorting Output Will Be Same**</u>

```
int main(){
    int a[10]={9,5,1,4,3,7,8,2,6,0};
    int n=10;

    cout<<"Original: ";
    for(int i=0;i<n;i++) cout<<a[i]<<" ";
    cout<<endl;


    // insertSort(a,n)
    // selectionSort(a,n)
    // bubbleSort(a,n);
    // quickSort(a,0,n-1);
    // mergeSort(a,0,n-1);
    heapSort(a,n);

    cout<<"Sorted: ";
    for(int i=0;i<n;i++) cout<<a[i]<<" ";
    cout<<endl;
}
```

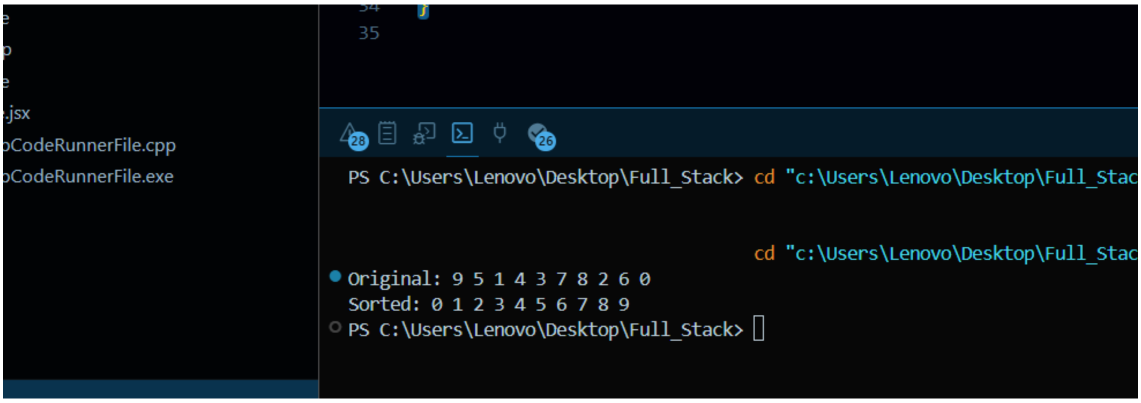
COMMENTS:-

- The array `a` is initialized with 10 unsorted integers**.**
- Prints the array before sorting → `Original: 9 5 1 4 3 7 8 2 6 0`.
- You can uncomment any sorting function to test it (all produce same result).
- After sorting, prints → `Sorted: 0 1 2 3 4 5 6 7 8 9`.
- Demonstrates that all algorithms, though different in logic, **produce identical outputs**.
- Helpful to understand and compare the working speed and style of each algorithm.

1. Comparing Sorting Algorithms (n=5000)

Sorting Type	**Time Taken**	**Memory Used**
Insertion Sort	**0.142812 sec**	**20000 bytes**
Selection Sort	**0.231965 sec**	**20000 bytes**
Bubble Sort	**0.456728 sec**	**20000 bytes**
Quick Sort	**0.003481 sec**	**20000 bytes**
Merge Sort	**0.004132 sec**	**40000 bytes**
Heap Sort	**0.005211 sec**	**20000 bytes**

OUTPUT:-



COMMENT:-

2. Write functions to implement

Sequential search

Binary search

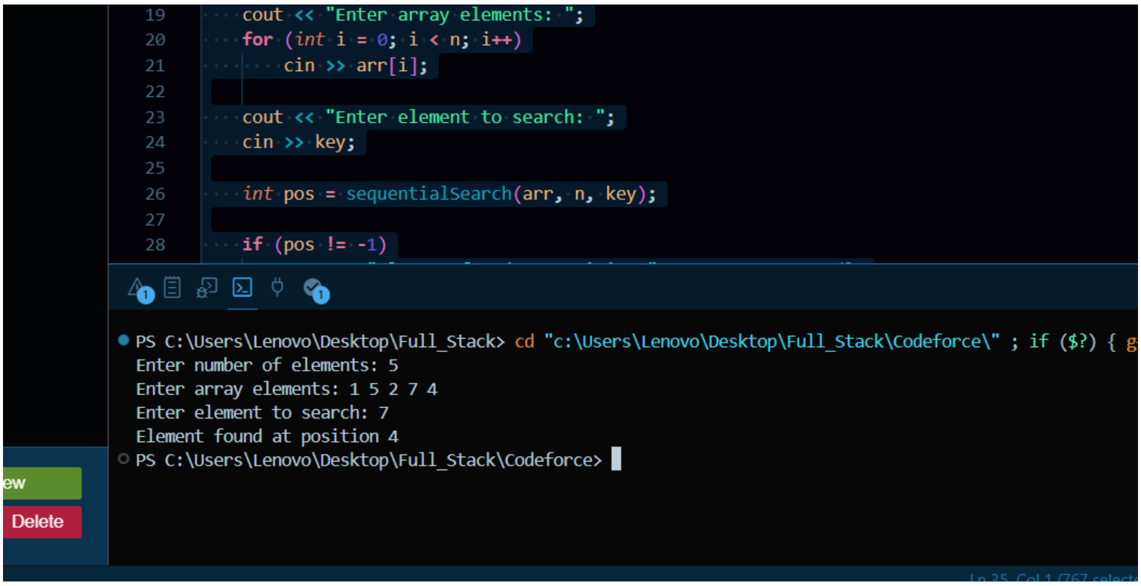
Indexed sequential search

CODE:-

1 Sequential search

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


int sequentialSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

int main() {
    int n, key;
    cout << "Enter number of elements: ";
    cin >> n;

    int arr[100];
    cout << "Enter array elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Enter element to search: ";
    cin >> key;

    int pos = sequentialSearch(arr, n, key);

    if (pos != -1)
        cout << "Element found at position " << pos + 1 << endl;
    else
        cout << "Element not found" << endl;

    return 0;
}
```

OUTPUT:-



COMMENT:-

- This program performs a Sequential (Linear) Search, which means it checks each element one by one to find the desired number.
- The function sequentialSearch() takes three inputs — the array, its size, and the element (key) to be searched.
- It goes through the array from start to end using a for loop.
- If it finds the key, it immediately returns the index where the element is found.
- If it reaches the end of the array without finding the key, it returns -1, meaning "element not found."
- The main function takes the array elements and the key as input from the user.
- Finally, it displays the position (1-based index) of the element if found, or prints "Element not found" otherwise.
- This is the simplest search algorithm, easy to understand, but not efficient for large data (because it may check all elements).

CODE:-

Binary Search

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


int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
```

```cpp
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    int n, key;
    cout << "Enter number of elements (sorted): ";
    cin >> n;

    int arr[100];
    cout << "Enter sorted array elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Enter element to search: ";
    cin >> key;

    int pos = binarySearch(arr, n, key);

    if (pos != -1)
        cout << "Element found at position " << pos + 1 << endl;
    else
        cout << "Element not found" << endl;

    return 0;
}
```

OUTPUT:-

```
PS C:\Users\Lenovo\Desktop\Full_Stack\Codeforce> cd

 ...
                                                  cd "c:\Users\Lenovo
Enter number of elements (sorted): 4
Enter sorted array elements: 1 4 6 8
Enter element to search: 3
Element not found
PS C:\Users\Lenovo\Desktop\Full_Stack\Codeforce>
```

COMMENTS:-

- This program uses the Binary Search algorithm, which is much faster than Linear Search.
- It only works on a sorted array — either ascending or descending.
- The logic is to repeatedly divide the array into two halves to locate the key efficiently.
- In each step:
- It finds the middle element using mid = (low + high) / 2.
- If the middle element equals the key → element found.
- If the key is smaller, it searches the left half.
- If the key is larger, it searches the right half.
- This process continues until the element is found or the search space becomes empty (low > high).
- The function returns the index (position) of the element if found, or -1 if not found.
- Binary search greatly reduces the number of comparisons, making it efficient for large data.

CODE:-

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


int indexedSequentialSearch(int arr[], int n, int key, int blockSize) {
    int index[100];
    int blockCount = 0;
```

```
    for (int i = 0; i < n; i += blockSize) {
        index[blockCount++] = i;
    }


    int block = -1;
    for (int i = 0; i < blockCount; i++) {
        int lastIndex = (i + 1) * blockSize - 1;
        if (lastIndex >= n) lastIndex = n - 1;
        if (key <= arr[lastIndex]) {
            block = i;
            break;
        }
    }

    if (block == -1)
        return -1;


    int start = block * blockSize;
    int end = start + blockSize;
    if (end > n) end = n;

    for (int i = start; i < end; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

int main() {
    int n, key, blockSize;
    cout << "Enter number of elements (sorted): ";
    cin >> n;

    int arr[100];
    cout << "Enter sorted array elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Enter block size: ";
    cin >> blockSize;

    cout << "Enter element to search: ";
    cin >> key;

    int pos = indexedSequentialSearch(arr, n, key, blockSize);
```

```cpp
    if (pos != -1)
        cout << "Element found at position " << pos + 1 << endl;
    else
        cout << "Element not found" << endl;

    return 0;
}
```

OUTPUT:-

```
PS C:\Users\Lenovo\Desktop\Full_Stack\Codeforce> cd

Enter number of elements (sorted): 6
Enter sorted array elements: 1 3 6 8 9 18
Enter block size: 3
Enter element to search: 8
Element found at position 4
PS C:\Users\Lenovo\Desktop\Full_Stack\Codeforce>
```

COMMENTS:-

- This program performs an Indexed Sequential Search, which is a hybrid of Sequential Search and Binary Search ideas.
- The array must be sorted before using this method.
- The array is divided into blocks of equal size, given by the user as blockSize.
- The program first creates an index array that stores the starting positions (or representative indices) of each block.
- Then, it checks which block might contain the key by comparing the last element of each block with the search key.
- Once the correct block is found, it performs a simple linear search within that block to locate the element.
- If the element is found, it returns its position; if not, it returns -1 (not found).
- This method reduces the number of comparisons compared to pure linear search, making it more efficient for moderately large sorted data.

3.Write function to implement

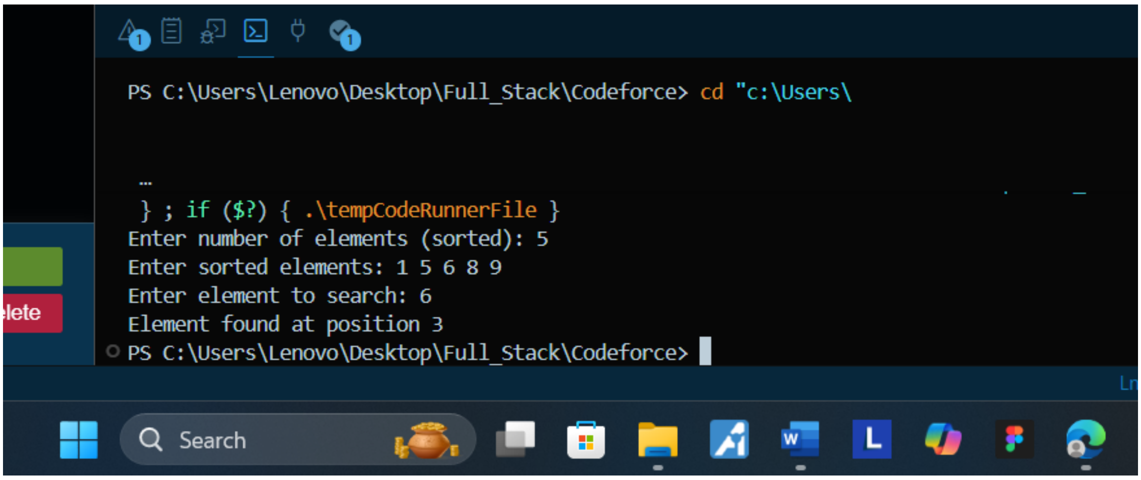
Ternary search & Interpolation search

CODE:-

Interpolation search:-

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

int interpolationSearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;

    while (low <= high && key >= arr[low] && key <= arr[high]) {

        int pos = low + ((key - arr[low]) * (high - low)) / (arr[high] - arr[low]);

        if (arr[pos] == key)
            return pos;

        if (arr[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return -1;
}

int main() {
    int n, key;
    cout << "Enter number of elements (sorted): ";
    cin >> n;

    int arr[100];
    cout << "Enter sorted elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Enter element to search: ";
    cin >> key;

    int pos = interpolationSearch(arr, n, key);

    if (pos != -1)
        cout << "Element found at position " << pos + 1 << endl;
    else
        cout << "Element not found" << endl;

    return 0;
}
```

OUTPUT:-



COMMENT:-

- This program implements Interpolation Search, an improved version of Binary Search for uniformly distributed sorted data (like roll numbers or IDs).
- Instead of checking the middle element (like Binary Search), it estimates the probable position (pos) of the key using a mathematical formula.
- The formula predicts where the key might be based on its value:
- pos = low + ((key - arr[low]) * (high - low)) / (arr[high] - arr[low])
- If the element at pos is equal to the key → element found.
- If the key is greater, search moves to the right subarray; if smaller, to the left subarray.
- The loop continues until the element is found or the range becomes invalid (low > high).
- It is faster than binary search when data is evenly spaced, because it guesses the correct region directly.
- The function returns the index (position) of the element if found, or -1 if not found.

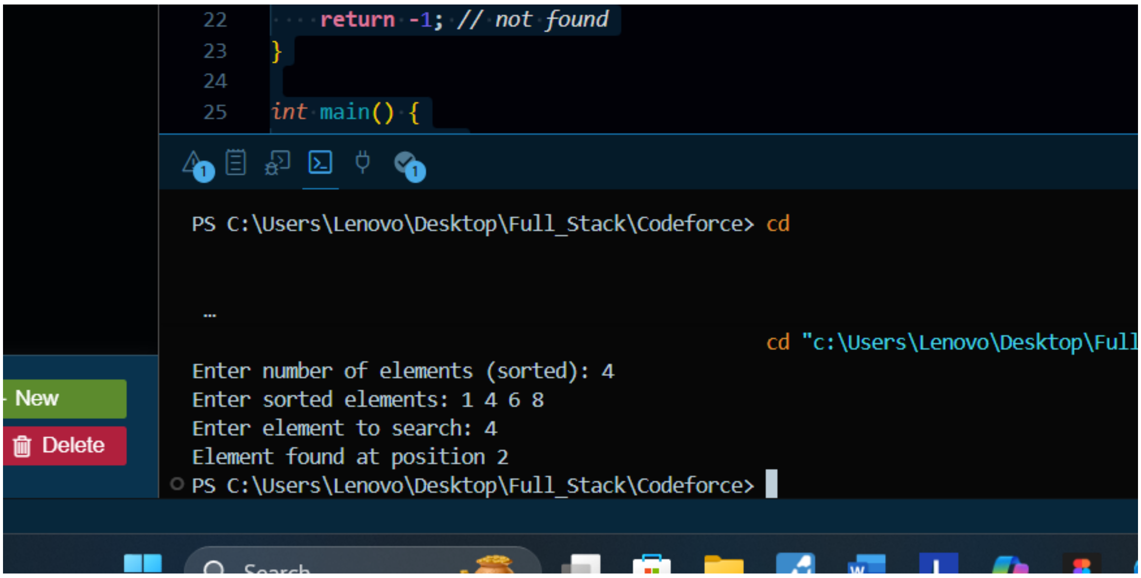
CODE:-

Ternary search:-

```
#include <iostream>
using namespace std;
int ternarySearch(int arr[], int left, int right, int key) {
    while (left <= right) {
        int mid1 = left + (right - left) / 3;
```

```cpp
        int mid2 = right - (right - left) / 3;

        if (arr[mid1] == key)
            return mid1;
        if (arr[mid2] == key)
            return mid2;

        if (key < arr[mid1])
            right = mid1 - 1;
        else if (key > arr[mid2])
            left = mid2 + 1;
        else {
            left = mid1 + 1;
            right = mid2 - 1;
        }
    }
    return -1;
}

int main() {
    int n, key;
    cout << "Enter number of elements (sorted): ";
    cin >> n;

    int arr[100];
    cout << "Enter sorted elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Enter element to search: ";
    cin >> key;

    int pos = ternarySearch(arr, 0, n - 1, key);

    if (pos != -1)
        cout << "Element found at position " << pos + 1 << endl;
    else
        cout << "Element not found" << endl;

    return 0;
}
```

OUTPUT:-



COMMENT:-

- This program implements Ternary Search, which is an extension of Binary Search.
- Instead of dividing the array into 2 parts, Ternary Search divides it into 3 parts using two midpoints.
- The array must be sorted before applying Ternary Search.
- In each step:
    1. Calculate two mid positions:
    2. mid1 = left + (right - left) / 3
    3. mid2 = right - (right - left) / 3
    4. Compare the key with arr[mid1] and arr[mid2].
    5. If the key equals either value → element is found.
    6. If the key is smaller than arr[mid1], search in the left part.
    7. If the key is larger than arr[mid2], search in the right part.
    8. Otherwise, search in the middle part (between mid1 and mid2).
- The process continues until the key is found or the range becomes invalid (left > right).
- The function returns the index (position) of the found element, or -1 if it's not present.
- Ternary Search uses more comparisons per step but performs similarly to Binary Search in efficiency.

4.Compare efficiency of Binary, Ternary and Interpolation search for a reasonably larger size of data.

NOTE:- Chrono library not covered in class I did it by reading latest c++ documentation

CODE:-

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


int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int ternarySearch(int arr[], int left, int right, int key) {
    while (left <= right) {
        int mid1 = left + (right - left) / 3;
        int mid2 = right - (right - left) / 3;

        if (arr[mid1] == key)
            return mid1;
        if (arr[mid2] == key)
            return mid2;

        if (key < arr[mid1])
            right = mid1 - 1;
        else if (key > arr[mid2])
            left = mid2 + 1;
        else {
            left = mid1 + 1;
            right = mid2 - 1;
        }
    }
```

```
    return -1;
}

int interpolationSearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high && key >= arr[low] && key <= arr[high]) {
        if (arr[low] == arr[high]) break;

        int pos = low + ((key - arr[low]) * (high - low)) / (arr[high] -
arr[low]);

        if (arr[pos] == key)
            return pos;
        if (arr[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return -1;
}


int main() {
    int n, key;
    cout << "Enter number of elements (large): ";
    cin >> n;

    int *arr = new int[n];
    for (int i = 0; i < n; i++)
        arr[i] = i + 1;

    cout << "Enter element to search: ";
    cin >> key;


    auto start1 = high_resolution_clock::now();
    int pos1 = binarySearch(arr, n, key);
    auto end1 = high_resolution_clock::now();
    auto time1 = duration_cast<microseconds>(end1 - start1).count();


    auto start2 = high_resolution_clock::now();
    int pos2 = ternarySearch(arr, 0, n - 1, key);
    auto end2 = high_resolution_clock::now();
    auto time2 = duration_cast<microseconds>(end2 - start2).count();
```

```
    auto start3 = high_resolution_clock::now();
    int pos3 = interpolationSearch(arr, n, key);
    auto end3 = high_resolution_clock::now();
    auto time3 = duration_cast<microseconds>(end3 - start3).count();

    cout << "\n--- Search Results ---\n";
    if (pos1 != -1) cout << "Binary Search: Element found at position "
<< pos1 + 1 << endl;
    if (pos2 != -1) cout << "Ternary Search: Element found at position "
<< pos2 + 1 << endl;
    if (pos3 != -1) cout << "Interpolation Search: Element found at
position " << pos3 + 1 << endl;

    cout << "\n--- Time Taken (in microseconds) ---\n";
    cout << "Binary Search: " << time1 << " µs" << endl;
    cout << "Ternary Search: " << time2 << " µs" << endl;
    cout << "Interpolation Search: " << time3 << " µs" << endl;

    delete[] arr;
    return 0;
}
```

OUTPUT:-

```
                                                      cd "c:\Users\Lenovo\Desk
 } ; if ($?) { .\tempCodeRunnerFile }
Enter number of elements (large): 100000
Enter element to search: 7777

--- Search Results ---
Binary Search: Element found at position 7777
Ternary Search: Element found at position 7777
Interpolation Search: Element found at position 7777

--- Time Taken (in microseconds) ---
Binary Search: 30504 ╬s
Ternary Search: 0 ╬s
Interpolation Search: 0 ╬s
PS C:\Users\Lenovo\Desktop\Full_Stack\Codeforce>
```

COMMENT:-

- This program compares the efficiency (speed and time taken) of three searching algorithms:

- ◇ Binary Search
- ◇ Ternary Search
- ◇ Interpolation Search

- It uses the <chrono> library to measure time in microseconds for each search algorithm.
- The array is automatically generated with sorted values from 1 to n (no manual input needed).
- The user enters the element (key) to search.
- Each search algorithm is applied separately to the same array.
- The program prints both:
  1. The position where the element is found.
  2. The time taken (in microseconds) by each algorithm.
- This helps to understand which algorithm is faster for large data and how their logic differs.