**Patuakhali Science and Technology University**



**Lab Problem 01**

Course Code: CCE 122

Course Title: Object Oriented Programming Sessional

**Submitted By:**

Mehadi Enam Nerjhor

ID: 2202050

REG: 11241

**Submitted To:**

Professor Dr. Md Samsuzzaman

Dept. of Computer and Communication Engineering

Faculty of Computer Science and Engineering

Date: 26 January 2025

Linear Search in Java

**Linear search**, also known as **sequential search**, is a simple and intuitive method for finding an element in a list. The algorithm works by checking each element one by one, starting from the beginning, until it either finds a match or reaches the end of the list.

This approach is particularly well-suited for small or unsorted datasets, where the overhead of implementing more complex search algorithms isn't justified. While it’s not the most efficient method for large datasets, its simplicity and ease of implementation make it a practical choice in certain scenarios.

**Algorithm:**

The algorithm follows these steps:

**Step 1:** Traverse over the array.

**Step 2:** Match the key element with array element.

**Step 3:** If key element is found, return the index position of the array element.

**Step 4:** If key element is not found, return -1.

Examples:

Example 1-

|  |
| --- |
| public class LinearSearchExample{  public static int linearSearch(int[] arr, int key){  for(int i=0;i<arr.length;i++){  if(arr[i] == key){  return i;  }  }  return -1;  }  public static void main(String a[]){  int[] a1= {10,20,30,50,70,90};  int key = 50;  System.out.println(key+" is found at index: "+linearSearch(a1, key));  }  } |

Output:

*50 is found at index: 3*

Example 2-

|  |
| --- |
| import java.util.Scanner;    class LinearSearchExample2  {  public static void main(String args[])  {  int c, n, search, array[];    Scanner in = new Scanner(System.in);  System.out.println("Enter number of elements");  n = in.nextInt();  array = new int[n];    System.out.println("Enter those " + n + " elements");    for (c = 0; c < n; c++)  array[c] = in.nextInt();    System.out.println("Enter value to find");  search = in.nextInt();    for (c = 0; c < n; c++)  {  if (array[c] == search) /\* Searching element is present \*/  {  System.out.println(search + " is present at location " + (c + 1) + ".");  break;  }  }  if (c == n) /\* Element to search isn't present \*/  System.out.println(search + " isn't present in array.");  }  } |

**Output:**

*Enter number of elements*

*5*

*Enter those 5 elements*

*10 20 30 40 50*

*Enter value to find*

*40*

*40 is present at location 4.*

Advantages and Disadvantages

Advantages

Simplicity: It is easy to understand and implement.

No Preprocessing: It does not require the data to be sorted.

Versatility: It can be used on any type of data structure (arrays, linked lists, etc.).

Disadvantages

Inefficiency: On average, it requires checking only the half of the elements (O(n/2)) and, in the worst case, all elements (O(n)).

Not Suitable for Large Datasets: It becomes impractical as the size of the dataset increases.

Binary Search in Java

Binary search is a highly efficient algorithm used to find the position of a target element within a **sorted array**. It works by repeatedly dividing the search interval in half, which significantly reduces the number of elements to be checked.

Binary Search Algorithm in Java

Define the method with an array and a target value as parameters.

Initialize left and right pointers to the start and end of the array, respectively.

Loop as long as left is less than or equal to right.

Find the middle of the current array segment (left + (right - left) / 2).

Compare the target with the middle element:

If equal, return the middle index (target found).

If the target is greater, search the right half (left = mid + 1).

If the target is less, search the left half (right = mid - 1).

If the loop ends without finding the target, return -1 (target not found).

Methods for Java Binary Search

There are three methods in Java to implement Binary Search in Java are mentioned below:

Iterative Method

Recursive Method

Inbuild Method

Examples:

Ex 1-

|  |
| --- |
| 1. class BinarySearchExample{ 2. **public** **static** **void** binarySearch(**int** arr[], **int** first, **int** last, **int** key){ 3. **int** mid = (first + last)/2; 4. **while**( first <= last ){ 5. **if** ( arr[mid] < key ){ 6. first = mid + 1; 7. }**else** **if** ( arr[mid] == key ){ 8. System.out.println("Element is found at index: " + mid); 9. **break**; 10. }**else**{ 11. last = mid - 1; 12. } 13. mid = (first + last)/2; 14. } 15. **if** ( first > last ){ 16. System.out.println("Element is not found!"); 17. } 18. } 19. **public** **static** **void** main(String args[]){ 20. **int** arr[] = {10,20,30,40,50}; 21. **int** key = 30; 22. **int** last=arr.length-1; 23. binarySearch(arr,0,last,key); 24. } 25. } |

**Output:**

*Element is found at index: 2*

Binary Search Example in Java using Iterative Method

|  |
| --- |
| 1. public **class** BinarySearchIterative { 3. // Method to perform binary search on a sorted array 4. **public** **static** **int** binarySearch(**int**[] arr, **int** target) { 5. **int** left = 0; 6. **int** right = arr.length - 1; 8. **while** (left <= right) { 9. **int** mid = left + (right - left) / 2; 11. // Check if target is present at mid 12. **if** (arr[mid] == target) { 13. **return** mid; // Target found 14. } 16. // If target greater than mid, ignore left half 17. **if** (arr[mid] < target) { 18. left = mid + 1; 19. } 20. // If target is smaller than mid, ignore right half 21. **else** { 22. right = mid - 1; 23. } 24. } 26. // Target not found 27. **return** -1; 28. } 30. // Main method to test the binarySearch method 31. **public** **static** **void** main(String[] args) { 32. **int**[] arr = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}; 33. **int** target = 23; 34. **int** result = binarySearch(arr, target); 36. **if** (result == -1) { 37. System.out.println("Element not present in the array."); 38. } **else** { 39. System.out.println("Element found at index: " + result); 40. } 41. } 42. } |

Output:

*Element found at index: 5*

Binary Search Example in Java using Recursion

|  |
| --- |
| 1. class BinarySearchExample1{ 2. **public** **static** **int** binarySearch(**int** arr[], **int** first, **int** last, **int** key){ 3. **if** (last>=first){ 4. **int** mid = first + (last - first)/2; 5. **if** (arr[mid] == key){ 6. **return** mid; 7. } 8. **if** (arr[mid] > key){ 9. **return** binarySearch(arr, first, mid-1, key);//search in left subarray 10. }**else**{ 11. **return** binarySearch(arr, mid+1, last, key);//search in right subarray 12. } 13. } 14. **return** -1; 15. } 16. **public** **static** **void** main(String args[]){ 17. **int** arr[] = {10,20,30,40,50}; 18. **int** key = 30; 19. **int** last=arr.length-1; 20. **int** result = binarySearch(arr,0,last,key); 21. **if** (result == -1) 22. System.out.println("Element is not found!"); 23. **else** 24. System.out.println("Element is found at index: "+result); 25. } 26. } |

**Output:**

*Element is found at index: 2*

Bubble Sort in Java

**Bubble Sort** is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Bubble Sort Algorithm

Start with the first element of the array.

Compare the current element with the next element.

If the current element is greater than the next element, swap them.

Move to the next element and repeat steps 2 and 3 until the end of the array.

Repeat steps 1-4 until no more swaps are needed, indicating the array is sorted.

Examples:

Ex 1-

|  |
| --- |
| 1. public **class** BubbleSortExample { 2. //function that sorts the array 3. **static** **void** bubbleSort(**int**[] arr) { 4. **int** n = arr.length; 5. **int** temp = 0; 6. **for**(**int** i=0; i < n; i++){ 7. **for**(**int** j=1; j < (n-i); j++){ 8. **if**(arr[j-1] > arr[j]){ 9. //swap elements 10. temp = arr[j-1]; 11. arr[j-1] = arr[j]; 12. arr[j] = temp; 13. } 14. } 15. } 16. } 17. **public** **static** **void** main(String[] args) { 18. **int** arr[] ={3,60,35,2,45,320,5}; 19. System.out.println("Array Before Bubble Sort"); 20. **for**(**int** i=0; i < arr.length; i++){ 21. System.out.print(arr[i] + " "); 22. } 23. System.out.println(); 24. bubbleSort(arr);//sorting array elements using bubble sort 25. System.out.println("Array After Bubble Sort"); 26. **for**(**int** i=0; i < arr.length; i++){ 27. System.out.print(arr[i] + " "); 28. } 29. } 30. } |

Output:

Array Before Bubble Sort

3 60 35 2 45 320 5

Array After Bubble Sort

2 3 5 35 45 60 320

Selection Sort in Java

**Selection Sort** is a simple sorting algorithm that works by repeatedly selecting the minimum (or maximum, depending on the order) element from the unsorted portion of the list and swapping it with the first unsorted element. This process is repeated until the entire list is sorted.

Steps of Selection Sort:

Start with the entire list as the unsorted portion.

Find the smallest (or largest, if sorting in descending order) element in the unsorted portion of the list.

Swap this minimum element with the first element of the unsorted portion.

Move the boundary of the sorted portion one element forward.

Repeat this process for the remaining unsorted portion of the list until the entire list is sorted.

Examples:

Ex 1-

|  |
| --- |
| 1. public **class** SelectionSort { 2. // Function to perform selection sort on an array 3. **public** **static** **void** selectionSort(**int**[] arr) { 4. **int** n = arr.length; 6. // Iterate through the array 7. **for** (**int** i = 0; i < n - 1; i++) { 8. // Find the index of the minimum element in the unsorted portion of the array 9. **int** minIndex = i; 10. // Search for the minimum element in the unsorted portion 11. **for** (**int** j = i + 1; j < n; j++) { 12. **if** (arr[j] < arr[minIndex]) { 13. minIndex = j; 14. } 15. } 16. // Swap the found minimum element with the first element of the unsorted portion 17. **int** temp = arr[minIndex]; 18. arr[minIndex] = arr[i]; 19. arr[i] = temp; 20. } 21. } 22. **public** **static** **void** main(String[] args) { 23. // Example array 24. **int**[] arr = {5, 1, 12, -5, 16, 2, 12, 14}; 25. System.out.println("Before Selection sort:"); 26. **for** (**int** num : arr) { 27. System.out.print(num + " "); 28. } 29. // Sort the array using the selection sort 30. selectionSort(arr); 31. // Print the sorted array 32. System.out.println("After Selection sort:"); 33. **for** (**int** num : arr) { 34. System.out.print(num + " "); 35. } 36. } 37. } |

Output:

Before Selection sort:

5 1 12 -5 16 2 12 14

After Selection sort:

-5 1 2 5 12 12 14 16

Insertion Sort in Java

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. It works similarly to how you might sort playing cards in your hands: starting with an empty hand, you take one card at a time and insert it into the correct position in the sorted part of the hand.

Steps of Insertion Sort:

Start from the second element (since the first element is considered already sorted).

Compare this element with the one before it.

If the current element is smaller, shift the previous element one position to the right.

Continue comparing the current element with the previous elements and shift those larger than the current element to the right until you find the correct position for the current element.

Insert the current element in its correct position.

Repeat this process for each element until the entire list is sorted.

Examples:

Ex 1-

|  |
| --- |
| 1. public **class** InsertionSortExample { 2. **public** **static** **void** insertionSort(**int** array[]) { 3. **int** n = array.length; 4. **for** (**int** j = 1; j < n; j++) { 5. **int** key = array[j]; 6. **int** i = j-1; 7. **while** ( (i > -1) && ( array [i] > key ) ) { 8. array [i+1] = array [i]; 9. i--; 10. } 11. array[i+1] = key; 12. } 13. } 15. **public** **static** **void** main(String a[]){ 16. **int**[] arr1 = {9,14,3,2,43,11,58,22}; 17. System.out.println("Before Insertion Sort"); 18. **for**(**int** i:arr1){ 19. System.out.print(i+" "); 20. } 21. System.out.println(); 23. insertionSort(arr1);//sorting array using insertion sort 25. System.out.println("After Insertion Sort"); 26. **for**(**int** i:arr1){ 27. System.out.print(i+" "); 28. } 29. } 30. } |

Output:

Before Insertion Sort

9 14 3 2 43 11 58 22

After Insertion Sort

2 3 9 11 14 22 43 58