*Additional note on linear and binary searching, sort ( selection, insertion and selection)*

Let us look at an example to compare the two:

Linear Search to find the element “J” in a given sorted list from A-X



Binary Search to find the element “J” in a given sorted list from A-X



Important Differences

|  |  |
| --- | --- |
| Linear Search  | Binary Search |
| In linear search input data need not to be in sorted. | In binary search input data need to be in sorted order. |
| It is also called sequential search. | It is also called half-interval search. |
| The time complexity of linear search O(n).  | The time complexity of binary search O(log n). |
| Multidimensional array can be used. | Only single dimensional array is used. |
| Linear search performs equality comparisons | Binary search performs ordering comparisons |
| It is less complex. | It is more complex. |
| It is very slow process. | It is very fast process. |

LINER SEARCHING EXAMPLE:

1. write a C++ code that return the index of value 4 in the given elements of array below using linear search

 array[] = { 12, 114, 0, 4, 9 }

 **Answer**

#include <iostream>

using namespace std;

int search(int array[], int n, int x)

{

 for (int i = 0; i < n; i++)

 if (array[i] == x)

 return i;

 return -1;

}

int main()

{

 int array[] = { 12, 114, 0, 4, 9 };

 int x = 4;

 int n = sizeof(array) / sizeof(array[0]);

 int result = search(array, n, x);

 (result == -1)

 ? cout << "Element not found"

 : cout << "Element found at i

2. write a c++ code using binary search to return the value of 7 from the given array?

Arr = {2, 4, 5, 17, 14, 7, 11, 22}

Answer

#include<bits/stdc++.h>

using namespace std;

int binarySearch(vector<int> arr,int x,int low,int high){

 while(low <= high){

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

 if (arr[mid] == x)

 return mid;

 else if (arr[mid] < x)

 low = mid + 1;

 else

 high = mid - 1;

 }

 return -1;

}

int main(){

 vector<int> arr = {2, 4, 5, 17, 14, 7, 11, 22};

 int x = 22;

 int result = binarySearch(arr, x, 0, arr.size()-1);

 if (result != -1)

 cout << result << endl;

 else

 cout << "Not found" << endl;

 return 0;

}

# Comparison among Bubble Sort, Selection Sort and Insertion Sort

### [Bubble Sort](https://www.geeksforgeeks.org/bubble-sort/)

[**Bubble sort**](https://www.geeksforgeeks.org/bubble-sort/) repeatedly compares and swaps(if needed) adjacent elements in every pass. In i-th pass of Bubble Sort (ascending order), last (i-1) elements are already sorted, and i-th largest element is placed at (N-i)-th position, i.e. i-th last position.

Algorithm:

BubbleSort (Arr, N) // Arr is an array of size N.

{

 For ( I:= 1 to (N-1) ) // N elements => (N-1) pass

 {

 // Swap adjacent elements of Arr[1:(N-I)]such that

 // largest among { Arr[1], Arr[2], ..., Arr[N-I] } reaches to Arr[N-I]

 For ( J:= 1 to (N-I) ) // Execute the pass

 {

 If ( Arr [J] > Arr[J+1] )

 Swap( Arr[j], Arr[J+1] );

 }

 }

}

Optimization of Algorithm: Check if there happened any swapping operation in the inner loop (pass execution loop) or not. If there is no swapping in any pass, it means the array is now fully sorted, hence no need to continue, stop the sorting operation. So we can optimize the number of passes when the array gets sorted before the completion of all passes. And it can also detect if the given / input array is sorted or not, in the first pass.

BubbleSort (Arr, N) // Arr is an array of size N.

{

 For ( I:= 1 to (N-1) ) // N elements => (N-1) pass

 {

 // Swap adjacent elements of Arr[1:(N-I)]such that

 // largest among { Arr[1], Arr[2], ..., Arr[N-I] } reaches to Arr[N-I]

 noSwap = true; // Check occurrence of swapping in inner loop

 For ( J:= 1 to (N-I) ) // Execute the pass

 {

 If ( Arr [J] > Arr[J+1] )

 {

 Swap( Arr[j], Arr[J+1] );

 noSwap = false;

 }

 }

 If (noSwap) // exit the loop

 break;

 }

Time Complexity:

* Best Case Sorted array as input. Or almost all elements are in proper place. [ O(N) ]. O(1) swaps.
* Worst Case: Reversely sorted / Very few elements are in proper place. [ O(N2) ] . O(N2) swaps.
* Average Case: [ O(N2) ] . O(N2) swaps.

2. **Selection Sort**

Selection sort selects i-th smallest element and places at i-th position. This algorithm divides the array into two parts: sorted (left) and unsorted (right) subarray. It selects the smallest element from unsorted subarray and places in the first position of that subarray (ascending order). It repeatedly selects the next smallest element.

Algorithm:

SelectionSort (Arr, N) // Arr is an array of size N.

{

 For ( I:= 1 to (N-1) ) // N elements => (N-1) pass

 {

 // I=N is ignored, Arr[N] is already at proper place.

 // Arr[1:(I-1)] is sorted subarray, Arr[I:N] is unsorted subarray

 // smallest among { Arr[I], Arr[I+1], Arr[I+2], ..., Arr[N] } is at place min\_index

 min\_index = I;

 For ( J:= I+1 to N ) // Search Unsorted Subarray (Right lalf)

 {

 If ( Arr [J] < Arr[min\_index] )

 min\_index = J; // Current minimum

 }

 // Swap I-th smallest element with current I-th place element

 If (min\_Index != I)

 Swap ( Arr[I], Arr[min\_index] );

 }

}

Time Complexity:

 Best Case [ O(N2) ]. And O(1) swaps.

 Worst Case: Reversely sorted, and when the inner loop makes a maximum comparison. [ O(N2) ] . Also, O(N) swaps.

 Average Case: [ O(N2) ] . Also O(N) swaps.

Space Complexity: [ auxiliary, O(1) ]. In-Place sort.(When elements are shifted instead of being swapped (i.e. temp=a[min], then shifting elements from ar[i] to ar[min-1] one place up and then putting a[i]=temp). If swapping is opted for, the algorithm is not In-place.)

Advantage:

 It can also be used on list structures that make add and remove efficient, such as a linked list. Just remove the smallest element of unsorted part and end at the end of sorted part.

 The number of swaps reduced. O(N) swaps in all cases.

 In-Place sort.

Disadvantage:

 Time complexity in all cases is O(N2), no best case scenario.

 It requires n-squared number of steps for sorting n elements.

 It is not scalable.

3. **Insertion Sort**

Insertion Sort is a simple comparison based sorting algorithm. It inserts every array element into its proper position. In i-th iteration, previous (i-1) elements (i.e. subarray Arr[1:(i-1)]) are already sorted, and the i-th element (Arr[i]) is inserted into its proper place in the previously sorted subarray.

Find more details in this GFG Link.

Algorithm:

InsertionSort (Arr, N) // Arr is an array of size N.

{

 For ( I:= 2 to N ) // N elements => (N-1) pass

 {

 // Pass 1 is trivially sorted, hence not considered

 // Subarray { Arr[1], Arr[2], ..., Arr[I-I] } is already sorted

 insert\_at = I; // Find suitable position insert\_at, for Arr[I]

 // Move subarray Arr [ insert\_at: I-1 ] to one position right

 item = Arr[I]; J=I-1;

 While ( J ? 1 && item < Arr[J] )

 {

 Arr[J+1] = Arr[J]; // Move to right

 // insert\_at = J;

 J--;

 }

 insert\_at = J+1; // Insert at proper position

 Arr[insert\_at] = item; // Arr[J+1] = item;

 }

 }

}

Time Complexity:

 Best Case Sorted array as input, [ O(N) ]. And O(1) swaps.

 Worst Case: Reversely sorted, and when inner loop makes maximum comparison, [ O(N2) ] . And O(N2) swaps.

 Average Case: [ O(N2) ] . And O(N2) swaps.

Space Complexity: [ auxiliary, O(1) ]. In-Place sort.

Advantage:

 It can be easily computed.

 Best case complexity is of O(N) while the array is already sorted.

 Number of swaps reduced than bubble sort.

 For smaller values of N, insertion sort performs efficiently like other quadratic sorting algorithms.

 Stable sort.

 Adaptive: total number of steps is reduced for partially sorted array.

 In-Place sort.

Disadvantage:

 It is generally used when the value of N is small. For larger values of N, it is inefficient.

 Similar as selection sort it requires n-squared number of steps for sorting n elements.

Example :

1. write a C++ code that perform bubble sort the following array

 arr[] = { 5, 1, 4, 2, 8}

Answer

// C++ program for implementation

// of Bubble sort

#include <bits/stdc++.h>

using namespace std;

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

 int i, j;

 for (i = 0; i < n - 1; i++)

 // Last i elements are already

 // in place

 for (j = 0; j < n - i - 1; j++)

 if (arr[j] > arr[j + 1])

 swap(arr[j], arr[j + 1]);

}

// Function to print an array

void printArray(int arr[], int size)

{

 int i;

 for (i = 0; i < size; i++)

 cout << arr[i] << " ";

 cout << endl;

}

// Driver code

int main()

{

 int arr[] = { 5, 1, 4, 2, 8};

 int N = sizeof(arr) / sizeof(arr[0]);

 bubbleSort(arr, N);

 cout << "Sorted array: \n";

 printArray(arr, N);

 return 0;

}

2. write C++ code that sorts the following array using Optimized Implementation of Bubble Sort:

arr[] = {64, 34, 25, 12, 22, 11, 90}

Answer

// Optimized implementation of Bubble sort

#include <bits/stdc++.h>

using namespace std;

// An optimized version of Bubble Sort

void bubbleSort(int arr[], int n)

{

 int i, j;

 bool swapped;

 for (i = 0; i < n-1; i++)

 {

 swapped = false;

 for (j = 0; j < n-i-1; j++)

 {

 if (arr[j] > arr[j+1])

 {

 swap(arr[j], arr[j+1]);

 swapped = true;

 }

 }

 // IF no two elements were swapped

 // by inner loop, then break

 if (swapped == false)

 break;

 }

}

// Function to print an array

void printArray(int arr[], int size)

{

 int i;

 for (i = 0; i < size; i++)

 cout <<" "<< arr[i];

}

// Driver program to test above functions

int main()

{

 int arr[] = {64, 34, 25, 12, 22, 11, 90};

 int N = sizeof(arr)/sizeof(arr[0]);

 bubbleSort(arr, N);

 cout <<"Sorted array: \n";

 printArray(arr, N);

 return 0;

}

2. write the C++ code using selection sorting

 arr[] = {64, 25, 12, 22, 11}

Answer

// C++ program for implementation of

// selection sort

#include <bits/stdc++.h>

using namespace std;

//Swap function

void swap(int \*xp, int \*yp)

{

 int temp = \*xp;

 \*xp = \*yp;

 \*yp = temp;

}

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++)

 {

 // Find the minimum element in

 // unsorted array

 min\_idx = i;

 for (j = i+1; j < n; j++)

 {

 if (arr[j] < arr[min\_idx])

 min\_idx = j;

 }

 // Swap the found minimum element

 // with the first element

 if (min\_idx!=i)

 swap(&arr[min\_idx], &arr[i]);

 }

}

//Function to print an array

void printArray(int arr[], int size)

{

 int i;

 for (i=0; i < size; i++)

 {

 cout << arr[i] << " ";

 cout << endl;

 }

}

// Driver program to test above functions

int main()

{

 int arr[] = {64, 25, 12, 22, 11};

 int n = sizeof(arr)/sizeof(arr[0]);

 selectionSort(arr, n);

 cout << "Sorted array: \n";

 printArray(arr, n);

 return 0;

}

3. write a c++ code using the insertion sort of following array

 arr[] = { 12, 11, 13, 5, 6 }

Answer

// C++ program for insertion sort

#include <bits/stdc++.h>

using namespace std;

// Function to sort an array using

// insertion sort

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;

 }

}

// A utility function to print an array

// of size n

void printArray(int arr[], int n)

{

 int i;

 for (i = 0; i < n; i++)

 cout << arr[i] << " ";

 cout << endl;

}

// Driver code

int main()

{

 int arr[] = { 12, 11, 13, 5, 6 };

 int N = sizeof(arr) / sizeof(arr[0]);

 insertionSort(arr, N);

 printArray(arr, N);

 return 0;

}

**Singly linked list** is a type of data structure that is made up of nodes that are created using self referential structures. Each of these nodes contain two parts, namely the data and the reference to the next list node. Only the reference to the first list node is required to access the whole linked list. This is known as the head. The last node in the list points to nothing so it stores NULL in that part.



Example for single linked list implementation

#include <iostream>

using namespace std;

struct Node {

 int data;

 struct Node \*next;

};

struct Node\* head = NULL;

void insert(int new\_data) {

 struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

 new\_node->data = new\_data;

 new\_node->next = head;

 head = new\_node;

}

void display() {

 struct Node\* ptr;

 ptr = head;

 while (ptr != NULL) {

 cout<< ptr->data <<" ";

 ptr = ptr->next;

 }

}

int main() {

 insert(3);

 insert(1);

 insert(7);

 insert(2);

 insert(9);

 cout<<"The linked list is: ";

 display();

 return 0;

}