UNIT-I

Introduction to Linear Data Structures: Definition and importance of linear data structures, Abstract data types (ADTs) and their implementation, Overview of time and space complexity analysis for linear data structures. Searching Techniques: Linear & Binary Search, Sorting Techniques: Bubble sort, Selection sort, Insertion Sort

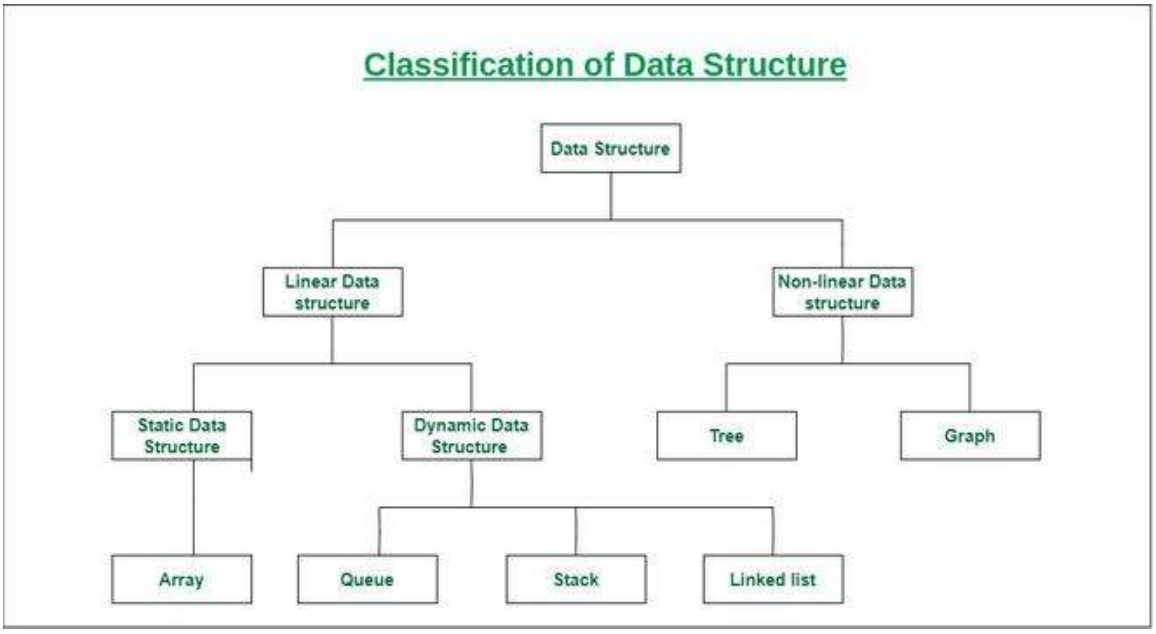
Definition of data structures

Defintion1:

A data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later

Defintion2:

A data structure is a specialized format for organizing, processing, retrieving and storing data.



- Linear Data Structure: A data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure. Examples are array, stack, queue, etc.

Examples of linear data structures are array, stack, queue, linked list, etc.

- Static data structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. An example of this data structure is an array.

- Dynamic data structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code. Examples of this data structure are queue, stack, etc.

- Non-linear Data Structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. Examples are trees and graphs.

1.Linear Data structures

1.1Definition:

A data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure. Examples are array, stack, queue

1.2Characteristics/importance of Linear Data Structures

The characteristics of linear data structures are as follows:

1. Sequential order: In a linear data structure, the elements are arranged in sequential order, and each element has a predecessor and successor, except for the first and last elements.
2. Traversed in a single run: Linear data structures can be traversed in a single run, which means that each element can be accessed by following a specific order.
3. Contiguous memory allocation: The elements in a linear data structure are stored in a contiguous block of memory, which allows for efficient memory management.
4. Simple implementation: Linear data structures have a simple implementation and are easy to understand.
5. Efficient data retrieval: Linear data structures provide efficient data retrieval, as they allow for easy access to the required data elements.
6. Linear complexity: Linear data structures have a linear time complexity, which means that the time required to perform operations on these structures increases linearly with the size of the structure.
7. Fixed or Dynamic Size: Linear data structures can have either fixed or dynamic sizes. Arrays typically have a fixed size when they are created, while other structures like linked lists, stacks, and queues can dynamically grow or shrink as elements are added or removed.

2.Abstract Data Type

2.1What is Abstract Data Type?

An Abstract Data Type (ADT) is a programming concept that defines a high-level view of a data structure, without specifying the implementation details. In other words, it is a blueprint for creating a data structure that defines the behavior and interface of the structure, without specifying how it is implemented.

An ADT in the data structure can be thought of as a set of operations that can be performed on a set of values. This set of operations actually defines the behavior of the data structure, and they are used to manipulate the data in a way that suits the needs of the program.

Some examples of ADT are Stack, Queue, List etc.

2.2 Implementation of ADT

Let us see some operations of those mentioned ADT −

- Stack ADT
  - isFull(), This is used to check whether stack is full or not
  - isEmpry(), This is used to check whether stack is empty or not
  - push(x), This is used to push x into the stack
  - pop(), This is used to delete one element from top of the stack
  - peek(), This is used to get the top most element of the stack
  - size(), this function is used to get number of elements present into the stack

- Queue ADT
  - isFull(), This is used to check whether queue is full or not
  - isEmpry(), This is used to check whether queue is empty or not
  - insert(x), This is used to add x into the queue at the rear end
  - delete(), This is used to delete one element from the front end of the queue
  - size(), this function is used to get number of elements present into the queue

- List ADT
  - get() – Return an element from the list at any given position.
  - insert() – Insert an element at any position of the list.
  - remove() – Remove the first occurrence of any element from a non-empty list.
  - removeAt() – Remove the element at a specified location from a non-empty list.
  - replace() – Replace an element at any position by another element.
  - size() – Return the number of elements in the list.
  - isEmpty() – Return true if the list is empty, otherwise return false.
  - isFull() – Return true if the list is full, otherwise return false.

3.Overview of Time and Space Complexity Analysis for Linear Data Structures in C

3.1 Overview of Time and Space Complexity

What is Time Complexity?

Time complexity is a measure of the amount of time an algorithm takes to execute as a function of the size of the input data.

The time complexity of an algorithm is typically expressed using "Big O" notation, which is a way of describing the upper bound of the growth rate of the running time of an algorithm as the input size grows. In other words, it describes how the running time of an algorithm changes as the input size increases.

For example,

- If an algorithm has a time complexity of O(n), where n is the size of the input data, it

  means that the running time of the algorithm is proportional to the size of the input data.

- This is called linear time complexity, because the running time increases linearly with the input size.
- Other common time complexity classes include O(1), which is constant time complexity, meaning that the running time of the algorithm is constant regardless of the input size.
- O(log n), which is logarithmic time complexity, meaning that the running time of the algorithm grows at a logarithmic rate as the input size increases.

What is Space Complexity?

Space complexity is a measure of the amount of memory or storage space required by an algorithm to solve a problem, as a function of the size of the input data.

It is an important concept in computer science, because algorithms that require a large amount of memory may not be practical for systems with limited resources, such as embedded systems or mobile devices.

For example,

- an algorithm with a space complexity of O(n) means that the amount of memory required

  by the algorithm grows linearly with the size of the input data.

3.2.Analysis for Linear Data Structures in C

Time Complexity in arrays of size 'N':

- It takes O(1) time to access an element at a specific memory address because its relative address can be calculated in constant time in an array called "ARR," where the address of the first element is "A" and the size of each element is "S."
- In a similar vein, editing any array element takes O(1) time.
- In arrays, the only way to insert a new element at a specific index is to skip all of the elements before that index, which takes O(N) time.
- O(N) time is required for the deletion operation as well.
- Without a specific algorithm, the search operation in an array takes O(N) times because we have to iterate over and check each element.

Space Complexity in Arrays:
The space complexity of fetching, overwriting, inserting, and deleting is constant, or O(1), as we do not require additional space to perform any of the aforementioned operations. The auxiliary space consists solely of the space used to construct the array.

Time complexity of Queue with N elements:

- The time required to access or edit any element in a queue is O(N), and in order to reach any given element, all elements after it must be removed.
- Due to the fact that reaching any particular element necessitates popping the elements stored after it, the searching operation also takes O(N) total time.
- In a queue, insertion and deletion take the same amount of time, O(1). At any given time, only the front component can be removed, and only the back component can be inserted.

The queue's complexity in space:
Since no additional space is required for any operation, the space complexity of each operation in a queue is O(1).

Time and complexity of a stack with N elements

- To access or edit any element in a stack, it takes O(N) times the number of elements to reach before it needs to be removed.
- Due to the fact that reaching any particular element necessitates popping the elements that were stored before it, the searching operation also takes O(N) total time.
- In a stack, operations like insertion and deletion take O(1) of time.

Space Complexity of Stack
Because no additional space is required for any operation, the space complexity of a stack for each operation is O(1).

Time Complexity of a Linked List with N Elements

- The time complexity of inserting an element into the linked list is O(1) if done on the head and O(N) if done anywhere else because we have to go through the linked list to get there.
- The time complexity of deletion is O(1) if carried out on the head and O(N) if carried out at any other location due to the necessity of traversing the linked list to get there.
- The time complexity for searching and accessing any elements is O(N).

Space Complexity of a Linked List
Because no additional space is required for any operation, the space complexity of a linked list is O(1).

The time and space complexities of linear data structures are summarized in below table:

Data Structure	Insert	Delete	Access	Search
Array	O(N)	O(N)	O(1)	O(N)
String	O(N)	O(N)	O(1)	O(N)
Queue	O(1)	O(1)	O(N)	O(N)
Stack	O(1)	O(1)	O(N)	O(N)
Linked List	O(1), if inserted on head O(N), elsewhere	O(1), if deleted on head O(N), elsewhere	O(N)	O(N)

Where the respective data structure's size is N.

II. Searching Techniques

Definition :

Searching is the fundamental process of locating a specific element or item within a collection of data.

1.Linear Search:

Defintion1:

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.

Defintion2:

Linear search: This is the most simple searching algorithm in the data structures that checks each element of the data structure until the desired element is found.

Procedure for Linear Search

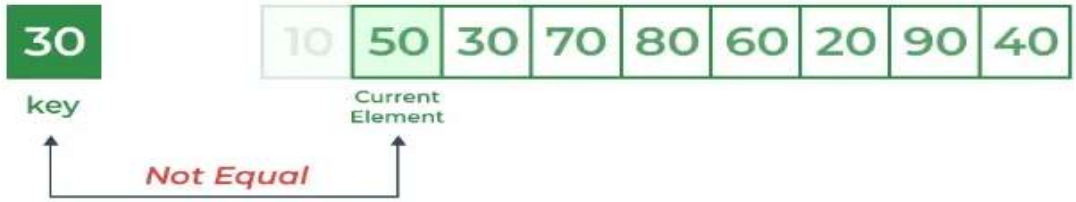
1. Start at the beginning of the list.
2. Compare the target value with the current element in the list.
3. If the current element matches the target value, the search is successful, and the position or index of the element is returned.
4. If the current element does not match the target value, move to the next element in the list.
5. Repeat steps 2-4 until a match is found or the end of the list is reached.
6. If the end of the list is reached without finding a match, the search is unsuccessful, and a special value (e.g., -1) may be returned to indicate the absence of the target value

For example: Consider the array **arr[] = {10, 50, 30, 70, 80, 20, 90, 40}** and **key** = 30

Step 1: Start from the first element (index 0) and compare key with each element (arr[i]).

- Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.

- Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.



Step 2: Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).


```
// C code to linearly search x in arr[].
#include <stdio.h>
int search(int arr[], int N, int x)
{
	for (int i = 0; i < N; i++)
		if (arr[i] == x)
			return i;
	return -1;
}
// Driver code
int main(void)
{
	int arr[] = { 2, 3, 4, 10, 40 };
	int x = 10;
	int N = sizeof(arr) / sizeof(arr[0]);

	// Function call
	int result = search(arr, N, x);
	(result == -1)
		? printf("Element is not present in array")
		: printf("Element is present at index %d", result);
	return 0;
}
```

Output:
Element is present at index 3

Complexity Analysis of Linear Search:
Time Complexity:

- Best Case: In the best case, the key might be present at the first index. So the best case complexity is O(1)
- Worst Case: In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- Average Case: O(N)

Auxiliary Space:

O(1) as except for the variable to iterate through the list, no other variable is used.

Advantages of Linear Search:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

Drawbacks of Linear Search:

- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
- Not suitable for large arrays.

2.Binary Search

Defintion1:

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Defintion2:

Binary search: This algorithm is used for searching in a sorted array or list. It works by repeatedly dividing the search interval in half until the desired element is found or the search interval is empty.

Procedure for Binary Search

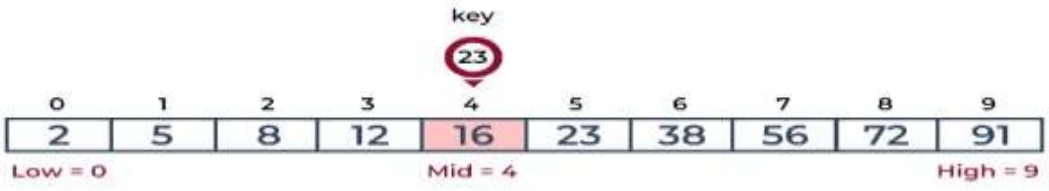
1. Start with a sorted array or list. For binary search to work correctly, the elements must be in ascending or descending order.
2. Set two pointers, low and high, to the beginning and end of the search space, respectively. Initially, low = 0 and high = length of the array - 1.
3. Calculate the middle index using the formula: mid = (low + high) / 2. This will give you the index of the element in the middle of the current search space.
4. Compare the target value with the element at the middle index:
   - If they are equal, the target value has been found. Return the index of the middle element.
   - If the target value is less than the middle element, set high = mid - 1 and go to step 3.
   - If the target value is greater than the middle element, set low = mid + 1 and go to step 3.
5. Repeat steps 3-4 until the target value is found or low > high. If low becomes greater than high, it means the target value is not present in the array.

Example:

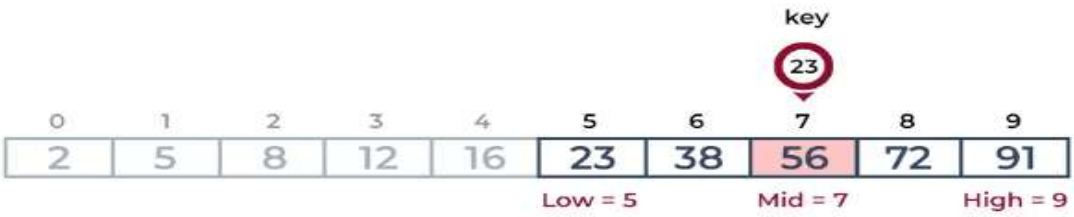
Consider an array arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, and the target = 23.

First Step: Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.

- Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.



- Key is less than the current mid 56. The search space moves to the left.



Second Step: If the key matches the value of the mid element, the element is found and stop search.


```
// C program to implement iterative Binary Search
#include <stdio.h>
// An iterative binary search function.
int binarySearch(int arr[], int l, int r, int x)
{
	while (l <= r) {
		int m = l + (r - l) / 2;

		// Check if x is present at mid
		if (arr[m] == x)
			return m;

		// If x greater, ignore left half
		if (arr[m] < x)
			l = m + 1;

		// If x is smaller, ignore right half
		else
			r = m - 1;
	}
```

```
        // If we reach here, then element was not present
        return -1;
}
// Driver code
int main(void)
{
        int arr[] = { 2, 3, 4, 10, 40 };
        int n = sizeof(arr) / sizeof(arr[0]);
        int x = 10;
        int result = binarySearch(arr, 0, n - 1, x);
        (result == -1)
? printf("Element is not present in array")
        : printf("Element is present at index %d",result);
        return 0;
}
```

Output:

Element is present at index 3

Time Complexity:

- Best Case: O(1)
- Average Case: O(log N)
- Worst Case: O(log N)

Auxiliary Space:

O(1), If the recursive call stack is considered then the auxiliary space will be O(logN).

Advantages of Binary Search:

- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Drawbacks of Binary Search:

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

III. Sorting Techniques

Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Types of Sorting Techniques

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. Comparison-based: We compare the elements in a comparison-based sorting algorithm)
2. Non-comparison-based: We do not compare the elements in a non-comparison-based sorting algorithm)

The sorting algorithm is important in Computer Science because it reduces the complexity of a problem.

1. Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Bubble Sort Algorithm

- Traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

- **Total no. of passes:** n-1
- **Total no. of comparisons:** n*(n-1)/2

Working of Bubble Sort algorithm:

- Lets consider the following array as an example: arr[] = {5, 1, 4, 2, 8}
- First Pass:
  Bubble sort starts with very first two elements, comparing them to check which one is greater.
  (5 1 4 2 8) –> (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  (1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4
  (1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2
  (1 4 2 5 8) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
- Second Pass:
  Now, during second iteration it should look like this:

(1 4 2 5 8) –> (1 4 2 5 8)
(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

- Third Pass:
  Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.
  (1 2 4 5 8) –> (1 2 4 5 8)
  (1 2 4 5 8) –> (1 2 4 5 8)
  (1 2 4 5 8) –> (1 2 4 5 8)
  (1 2 4 5 8) –> (1 2 4 5 8)

```
// Optimized implementation of Bubble sort
#include <stdbool.h>
#include <stdio.h>

void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// 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++)
```

```
        printf("%d ", 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);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Output:

Sorted array:

11 12 22 25 34 64 90

Time Complexity: $O(N^2)$
Auxiliary Space: O(1)

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

2.Selection sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted subarray and putting it at the beginning of the sorted subarray. In this article, we will learn to write a C program to implement selection sort.

Selection sort is another sorting technique in which we find the minimum element in every iteration and place it in the array beginning from the first index. Thus, a selection sort also gets divided into a sorted and unsorted subarray.

Selection Sort Algorithm

- Initialize minimum value **min_idx** to location 0.
- Traverse the array to find the minimum element in the array.
- While traversing if any element smaller than **min_idx** is found then swap both values.
- Then, increment **min_idx** to point to the next element.
- Repeat until the array is sorted.

Working of Selection Sort algorithm:

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}

First pass:

For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.

64	25	12	22	11

Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

11	25	12	22	64

Second Pass:

For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

11	25	12	22	64

After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

11	12	25	22	64

Third Pass:

Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array.

11	12	25	22	64

While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.

11	12	22	25	64

Fourth pass:

Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
As 25 is the 4th lowest value hence, it will place at the fourth position.

11	12	22	25	64

Fifth Pass:

At last the largest value present in the array automatically get placed at the last position in the array
The resulted array is the sorted array.

11	12	22	25	64

```
// C program for implementation of selection sort
#include <stdio.h>

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++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

```
// 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);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Sorted array:

11 12 22 25 64

Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is $O(N^2)$ as there are two nested loops:

- One loop to select an element of Array one by one = O(N)
- Another loop to compare that element with every other Array element = O(N)
- Therefore overall complexity = O(N) * O(N) = O(N*N) = $O(N^2)$

Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

3.Insertion sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

To sort an array of size N in ascending order:

- Iterate from arr[1] to arr[N] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Working of Insertion Sort algorithm:

Consider an example: arr[]: {12, 11, 13, 5, 6}

12	11	13	5	6

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6

Second Pass:

- Now, move to the next two elements and compare them

11	12	13	5	6

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

- Now, two elements are present in the sorted sub-array which are 11 and 12
- Moving forward to the next two elements which are 13 and 5

11	12	13	5	6

- Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6

- After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6

- Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6

- here, it is at its correct position

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are 5, 11 and 12
- Moving to the next two elements 13 and 6

5	11	12	13	6

- Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13

- Now, 6 is smaller than 12, hence, swap again

5	11	6	12	13

- Here, also swapping makes 11 and 6 unsorted hence, swap again

5	6	11	12	13

- Finally, the array is completely sorted.

```
// C program for insertion sort
#include <math.h>
#include <stdio.h>

/* 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++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```
5 6 11 12 13

Time Complexity: O(N^2)
Auxiliary Space: O(1)

Complexity Analysis of Insertion Sort:

Time Complexity of Insertion Sort

- The worst-case time complexity of the Insertion sort is O(N^2)
- The average case time complexity of the Insertion sort is O(N^2)
- The time complexity of the best case is O(N).

Space Complexity of Insertion Sort

The auxiliary space complexity of Insertion Sort is O(1)