DATA STRUCTURES

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: A data structure is a way of organizing and storing data in a computer so that it can be accessed and used efficiently. It refers to the logical or mathematical representation of data, as well as the implementation in a computer program.

Type of Data Structures:
1. Linear Data Structures
2. Non Linear Data Structures

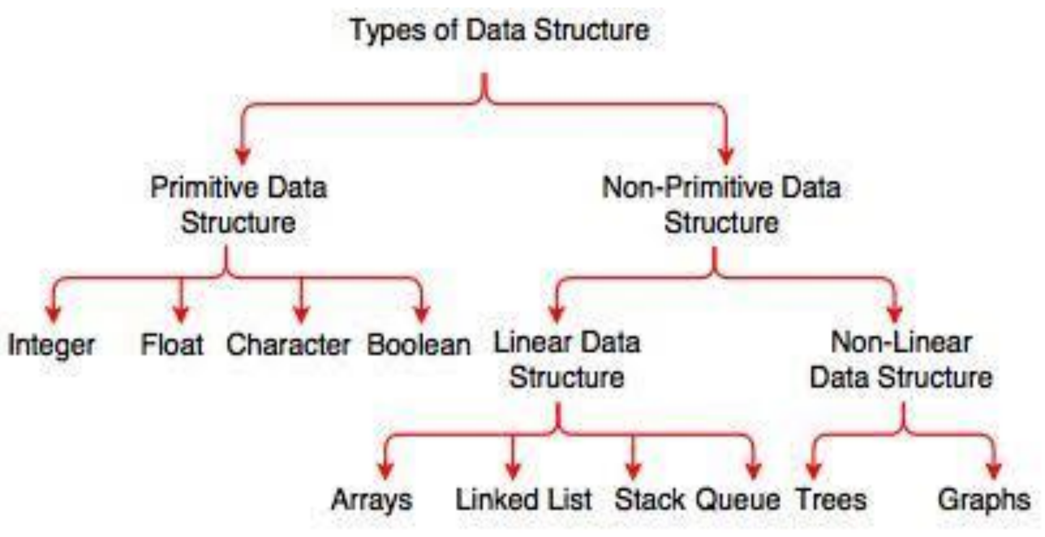
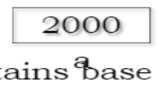


Fig. Types of Data Structure

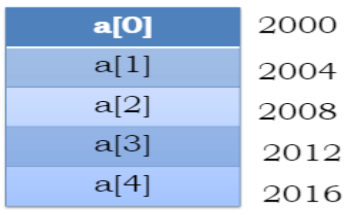
Linear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure. Examples of linear data structures are array, stack, queue etc.

Array Array is collection of similar data elements stored in contiguous memory locations Under a single name. c int a[5];



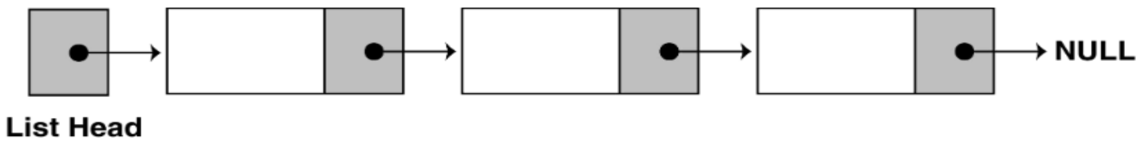
Here a, is the name of array, which contains base Address. Base address refer to address of first element in the array.
Now , 3rd memory location is accessed with a[2], whose effective Address is calculated as
base Address + index*sizeof(type), that is

2000 + 2 * 4 = 2008
is effective address of 3rd element



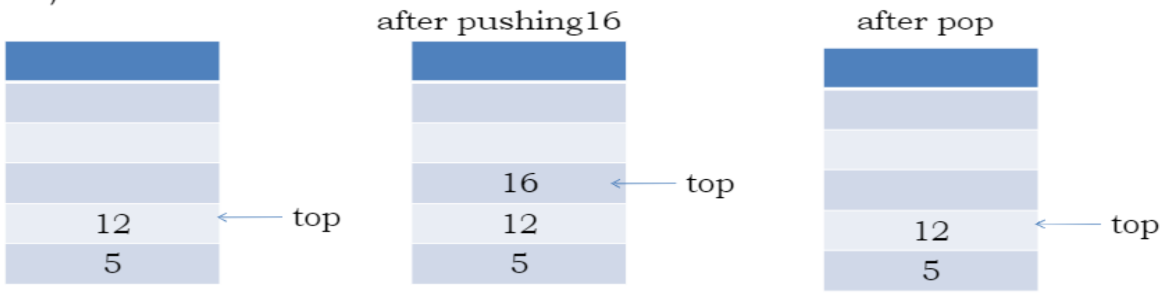
Linked List

Linked list is collection connected node. In Single Linked list every node contains data and address of the next node.



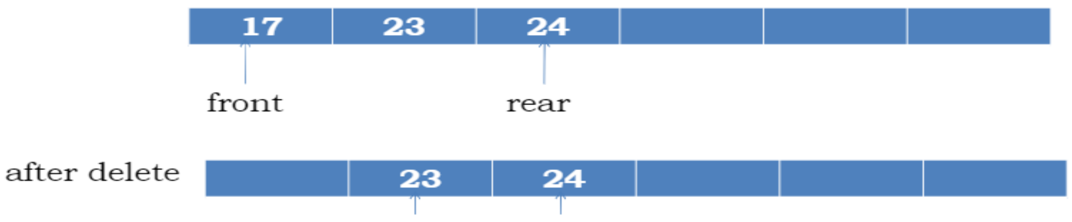
Stack

Stack is collection similar data elements, in which insertion and deletion can be done at only one end called top. In stack, element insert at last will be the first to delete, so we call stack as LIFO(Last In First Out).



Queue

Queue is collection similar data elements, in which elements are inserted at the rear and deleted from the front. In Queue, first element inserted will the first to delete, so it called FIFO(First In First Out).



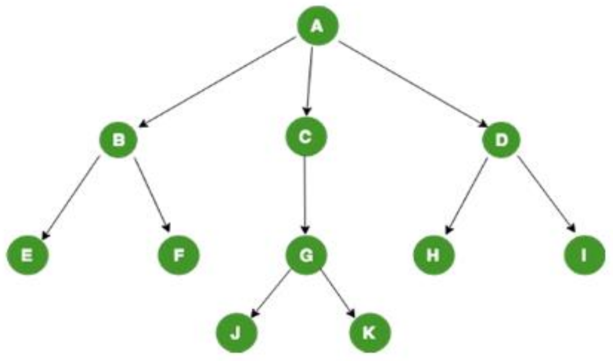
Non Linear Data Structures

A data structure in which elements are not sequential is called Non Linear Data Structures. In Non Linear Data Structures, elements may be stored in hierarchical way.

Trees and graph are Non Linear Data Structures

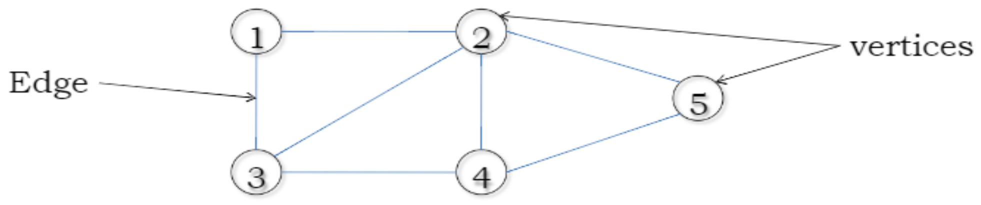
Tree

Tree is non linear data structure, which represents a parent child kind of hierarchical data structure.



Graph

Graph is non liner data structure, which is collection of vertices and edges set.



V = {1,2,3,4,5} and E = {<1,2>, <1,3>, <2,3>, <3,4>, <2,4>, <2,5>, <4,5>}.

Importance of linear data structures

- **Data can be inserted or deleted efficiently.**

Arrays make it easy to access elements by index, and linked lists help insert or delete data at the beginning or end of a given list.

- **Linear data structures are easy to understand and implement.**

For instance, arrays help to store elements in a straightforward way, and linked lists use pointers that help to connect the nodes.

- **Linear data structures offer flexibility as they allow dynamic resizing.**

This facilitates easy accommodation of the varying data amounts. It can easily grow or shrink in size by removing or adding the nodes as and when required.

Let us now have a look at the disadvantages associated with linear data structures:

- **During the initialization of arrays, they have a fixed size determined.**

As a result, the flexibility is limited. If the size is too small to store any additional elements, the entire array may be resized.

- **There are insufficient search operations in linear data structures.**

For instance, in the case of linked lists and arrays, one has to traverse the elements one by one until the desired result is obtained. The time complexity of O(n) in the case of linear search may lead to inefficient results if the data set is large.

- **Linked lists need more memory to store pointers than arrays.**

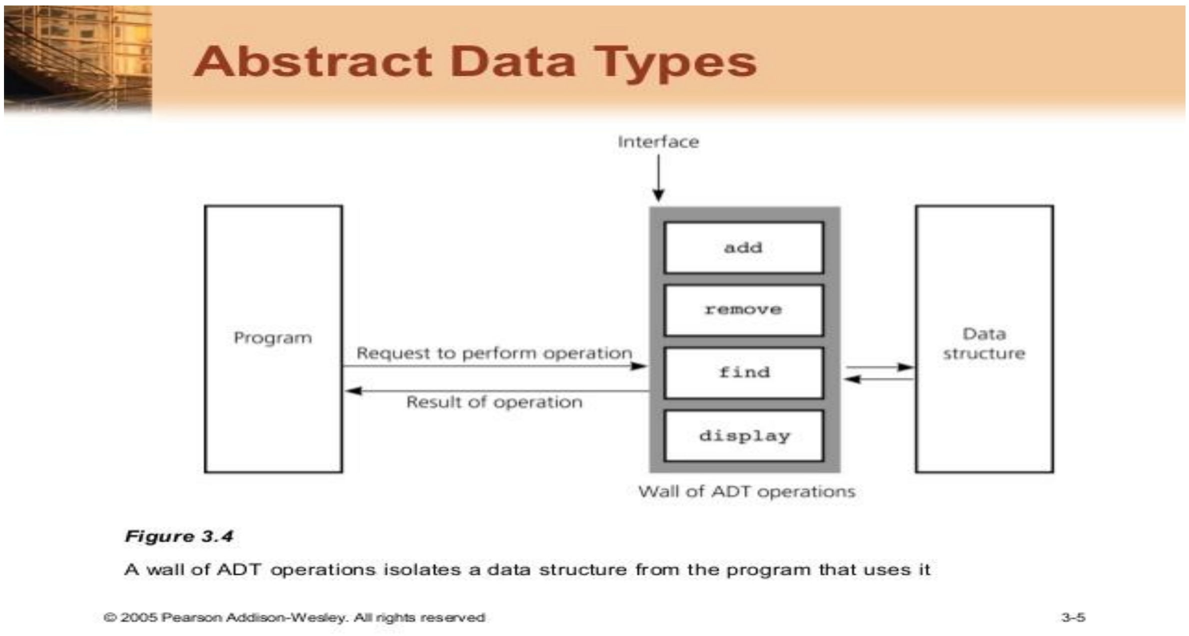
This increases the overall memory usage of the data structure and causes the efficiency to decrease.

Abstract data types (ADTs) and their implementation

Abstract data type (ADT) is a user defined data types which defines operations on values using functions without specifying what is inside the function and how the operations are performed.

Implementing ADTs

1. Choice of data structure depends on details of the ADT's operations and context in which the operations will be used.
2. Implementation details should be hidden behind a wall of ADT operations.



Abstract Data Types

Figure 3.4

A wall of ADT operations isolates a data structure from the program that uses it

3-5

<u>Array as an Abstract Data Type</u>

Abstract Data Type   Array

{

<u>Instances</u>: ordered finite collection of zero or more elements.
<u>Operations</u>:
int max (): It will return the maximum value in the given array.
int size (): return the number of elements in an array.
int get (index): return the element in given index.
int min (): It will return the minimum value in the given array.
void del(index): delete the element in the given index
void Insert (index, x): insert an element x in the given index position
void output (): It will print the array elements from left to right.
}

Overview of time and space complexity analysis for linear data structures.

What is Time Complexity?

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered.

The time complexity of an algorithm is the amount of time it takes for each statement to complete.

What is Space Complexity?

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

Example to find time and space complexity for linear search:

Analysis of Best Case Time Complexity of Linear Search

The Best Case will take place if:

- The element to be search is on the first index.

The number of comparisons in this case is 1. Thereforce, Best Case Time Complexity of Linear Search is O (1).

Analysis of Average Case Time Complexity of Linear Search

Let there be N distinct numbers: a1, a2, ..., a(N-1), aN

We need to find element P.

There are two cases:

- Case 1: The element P can be in N distinct indexes from 0 to N-1.
- Case 2: There will be a case when the element P is not present in the list.

There are N case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total.

If element P is in index K, then Linear Search will do K+1 comparisons.

Number of comparisons for all cases in case 1 = Comparisons if element is in index 0 + Comparisons if element is in index 1 + ... + Comparisons if element is in index N-1
= 1 + 2 + ... + N
= N * (N+1) / 2 comparisons

If element P is not in the list, then Linear Search will do N comparisons.

Number of comparisons for all cases in case 2 = N

Therefore, total number of comparisons for all N+1 cases = N * (N+1) / 2 + N
= N * ((N+1)/2 + 1)

Average number of comparisons = (N * ((N+1)/2 + 1)) / (N+1)
= N/2 + N/(N+1)

The dominant term in "Average number of comparisons" is N/2. So, the Average Case Time Complexity of Linear Search is O(N).

Analysis of Worst Case Time Complexity of Linear Search

The worst case will take place if:

- The element to be search is in the last index
- The element to be search is not present in the list

In both cases, the maximum number of comparisons take place in Linear Search which is equal to N comparisons.

Hence, the Worst Case Time Complexity of Linear Search is O(N).

Number of Comparisons in Worst Case: N

Analysis of Space Complexity of Linear Search

In Linear Search, we are creating a boolean variable to store if the element to be searched is present or not.

The variable is initialized to false and if the element is found, the variable is set to true. This variable can be used in other processes or returned by the function.

In Linear Search function, we can avoid using this boolean variable as well and return true or false directly.

The input to Linear Search involves:

- A list/ array of N elements
- A variable storing the element to be searched.

As the amount of extra data in Linear Search is fixed, the Space Complexity is O(1).

Therefore, Space Complexity of Linear Search is O(1).

Searching and Sorting Techniques

Searching is a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. The basic characteristics of any searching algorithms is – the searching method should be efficient; it should have less number of computations involved into it as well as the space occupied by such a technique should be less. The searching can be done by various methods. Some of the standard searching technique that is being followed in data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

Linear or Sequential Search: Linear Search does not expect specific ordering of the data. In fact, the data is arranged in the list. At every iteration the record will be compared with the help of the key. In some cases, all the elements get compared with the key value. All through this is a simple searching technique. Some unnecessary comparisons has to be performed. This method does not give the satisfactory solution for the system for large number of elements.

The time complexity of this algorithm is O(n). The time complexity will increase linearly with the value of n.

Advantages of Linear Search

- It is simple and easy method
- It is efficient for small data
- It is suitable for unsorted data
- It is suitable for storage structures which do not support direct access to data, for example, magnetic tape, linked list etc.
- Time complexity is in the order of n denoted as O(n).

Disadvantages of Linear Search

- It is inefficient for large data.
- In the case or ordered data other search techniques such as binary search are found more suitable.

Algorithm for Linear Search: -

```
Linear Search (Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Program for Linear Search: (without recursion)

```
#include <stdio.h>
 void main ()
{
   int a[10];
   int i, n, key, found = 0;
   clrscr();
   printf("Enter the value of num \n");
   scanf("%d", &n);
   printf("Enter %d values \n",n);
   for (i = 0; i < n; i++)
   {
          scanf("%d", &a[i]);
   }
   printf("Enter the element to be searched \n");
   scanf("%d", &key);



 /*  Linear search begins */
   for (i = 0; i < n ; i++)
   {
          if (key == a[i] )
          {
             found = 1;
             break;
          }
   }
   if (found == 1)
          printf("Linear Search Success\n");
   else
          printf("Linear Search is Unsuccess\n");
   getch();
}
```

Program for Linear Search: (using recursion)

```
#include<stdio.h>
   int recursiveLinearSearch (int a[],int key,int n)
 {
          n=n-1;
          if(n<0)
          {
          return -1;
          }
          else if(a[n]==key)
          {
          return 1;
          }
          else
          {
          return recursiveLinearSearch(a,key,n);
          }
 }


   void main()
  {

          int n,a[20],key,i,result;
          clrscr();
          printf("Enter The Size Of Array:\n");
          scanf("%d",&n);
          printf("\n Enter %d values\n",n);
          for(i=0;i<n;i++)
          scanf("%d",&a[i]);
          printf("Enter Key To Search  in Array\n");
          scanf("%d",&key);
          result=recursiveLinearSearch(a,key,n--);
          if(result==1)
          {
          printf("Linear Search Success");
          }
          else
          {
          printf("Linear Search Unsuccessful");
          }
          getch();
  }
```

Binary Search: -

Binary search is a fast search algorithm with run-time complexity of **O ($\log_2$ n)**. This search algorithm works on the principle of divide and conquer. The prerequisite for this searching technique is that the list should be sorted.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Pros:

- ➢ Suitable for sorted data
- ➢ Efficient for large data
- ➢ Suitable for storage structures that support direct access to data
- ➢ Compare to linear search, binary search is much faster
- ➢ Time complexity is **O ($\log_2$ n)**

Disadvantages of Binary Search:

- ➢ Not suitable for unsorted data
- ➢ Not Suitable for storage structures that does not support direct access to data, for example, magnetic tape and linked list.
- ➢ Inefficient for small list
- ➢ It is complex to implement.

Algorithm for Binary Search: -

```
Step 1 − Start searching data from middle of the list.
Step 2 − If it is a match, return the index of the item, and exit.
Step 3 − If it is not a match, probe position.
Step 4 − Divide the list using probing formula and find the new middle.
Step 5 − If data is greater than middle, search in higher sub-list.
Step 6 − If data is smaller than middle, search in lower sub-list.
Step 7 − Repeat until match.
```

Program for Binary Search: (without recursion)

```
#include <stdio.h>
void binarysearch ();
int a[50], n, item, loc, low, mid, high, i;
void main()
{
   printf("\nEnter size of an array: ");
   scanf("%d", &n);
   printf("\nEnter %d elements of an array in sorted form:\n", n);
   for(i=0; i<n; i++)
```

```
        scanf("%d", &a[i]);
    printf("\nEnter element to be searched: ");
    scanf("%d", &item);
    binarysearch();
    getch();
}
void binarysearch()
{
    low = 0;
    high = n-1;
    mid = (low + high) / 2;
    while ((low<=high) && (a[mid]!=item))
    {
        if (item < a[mid])
            high = mid - 1;
        else
            low = mid + 1;
        mid = (low + high) / 2;
    }
    if (a[mid] == item)
        printf("\n\nBinary Search is Success");
    else
        printf("\nBinary Search is Unsuccessful");
}
```

Program for Binary Search: (Using recursion)

```
#include<stdio.h>
void main(){
    int a[10], i, n, m, c, l, u;
    printf("Enter the size of an array: ");
    scanf("%d", &n);

    printf("Enter %d values in Ascending Order\n",n);
    for(i=0;i<n;i++)
            scanf("%d",&a[i]);

    printf("Enter the number to be search: ");
    scanf("%d",&m);

    l=0,u=n-1;
    c=binary(a,n,m,l,u);
    if(c==0)
            printf("Number is not found.");
    else
            printf("Number is found.");

    return 0;
 }

int binary(int a[],int n, int m, int l, int u)
{

     int mid, c=0;
    if(l<=u){
            mid=(l+u)/2;
            if(m==a[mid])
            {
               c=1;
            }
            else if(m<a[mid]){
               return binary(a,n,m,l,mid-1);
            }
            else
               return binary(a,n,m,mid+1,u);
    }
     return c;
}
```

Sorting: The sorting is a technique by which we expect the list of elements to be arranged as we expect. Sorting order is nothing but the arrangement of the elements in some specific manner. Usually the sorting order of two types: -

Ascending order: It is the sorting order in which the elements are arranged from low value to high value. In other words, elements are in increasing order.
For example: 10, 50, 40, 20, 30
Can be arranged in ascending order after applying some sorting technique as
10, 20, 30, 40, 50

Descending order: It is the sorting order in which the elements are arranged from high value to low value. In other words, elements are in decreasing order.
For example: 10, 50, 40, 20, 30
Can be arranged in ascending order after applying some sorting technique as
50, 40, 30, 20, 10

Sorting algorithms are divided into two categories:

(1) **Internal Sorting:** Any sorting algorithm that uses main memory exclusively during the sorting is called as an internal sorting algorithm. Internal sorting is faster than external sorting. The various internal sorting techniques are the following:
- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Heap Sort
- Shell sort
- Bucket Sort
- Radix Sort

(2) External Sorting: Any sorting algorithm that uses external memory, such as tape or disk, during the sorting is called as an external sort algorithm. Merge sort uses external memory.

Bubble Sort: (Sorting by Exchange)

- Bubble sort, sometimes referred to as sinking sort.
- Time complexity of bubble sort is O(n^2)
- It is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

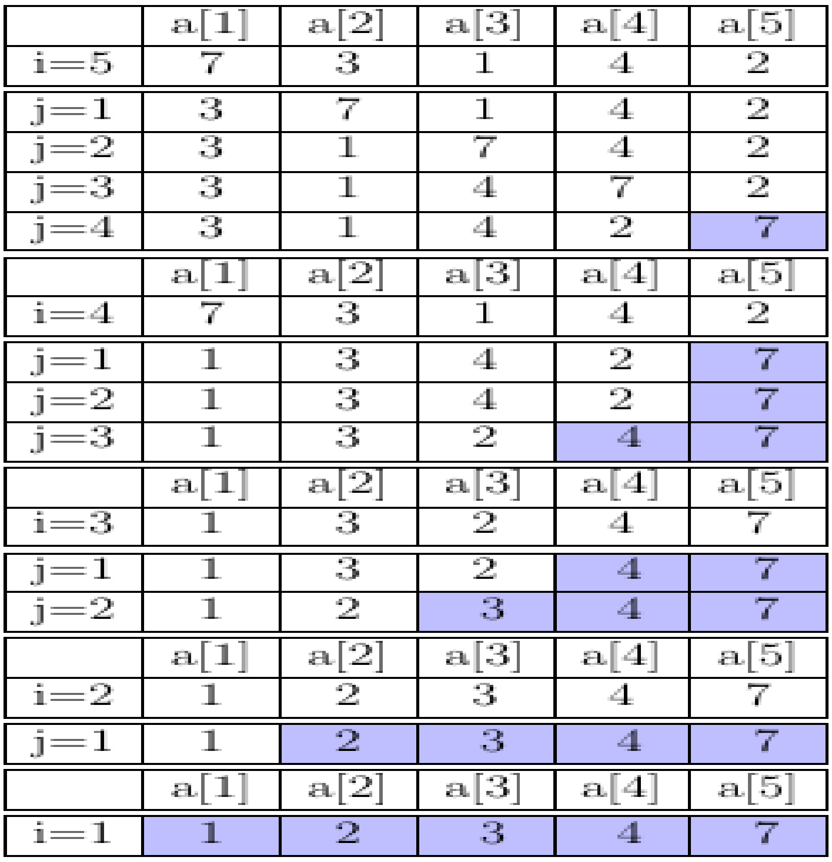
- Although the algorithm is simple,
- It is too slow and impractical for most problems even when compared to insertion sort.

Algorithm for Bubble Sort:

1. Read the total number of elements say n
2. Store the elements in the array
3. Set i=0
4. Compare the adjacent elements
5. Repeat step-4 for all n elements
6. Increment the value of i by 1 and repeat step 4,5 for i<n
7. Print the sorted list of elements
8. stop

Program for Bubble Sort:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[20],n,i;
void bubble(int a[],int n);
clrscr();
printf("\nEnter N Value\n");
scanf("%d",&n);
printf("\nEnter %d values\n",n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("\nBefore Sorting\t\t\t");
for(i=0;i<n;i++)
printf("%4d",a[i]);
bubble(a,n);
printf("\nSorted List After Bubble Sort\t");
for(i=0;i<n;i++)
printf("%4d",a[i]);
getch();
}
void bubble(int a[20],int n)
{
int i, j, temp, p;
for(i=0;i<n-1;i++)
{
        for(j=0;j<n-1-i;j++)
        {
                if(a[j]>a[j+1])
                {
                        temp=a[j];
                        a[j]=a[j+1];
                        a[j+1]=temp;
                }
        }
printf ("\n After %d Iteration\t\t",i+1);
for(p=0;p<n;p++)
printf("%4d",a[p]); }
}
```



	a[1]	a[2]	a[3]	a[4]	a[5]
i=5	7	3	1	4	2
j=1	3	7	1	4	2
j=2	3	1	7	4	2
j=3	3	1	4	7	2
j=4	3	1	4	2	7
	a[1]	a[2]	a[3]	a[4]	a[5]
i=4	7	3	1	4	2
j=1	1	3	4	2	7
j=2	1	3	4	2	7
j=3	1	3	2	4	7
	a[1]	a[2]	a[3]	a[4]	a[5]
i=3	1	3	2	4	7
j=1	1	3	2	4	7
j=2	1	2	3	4	7
	a[1]	a[2]	a[3]	a[4]	a[5]
i=2	1	2	3	4	7
j=1	1	2	3	4	7
	a[1]	a[2]	a[3]	a[4]	a[5]
i=1	1	2	3	4	7

<u>Selection Sort:</u> **Selection sort** is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

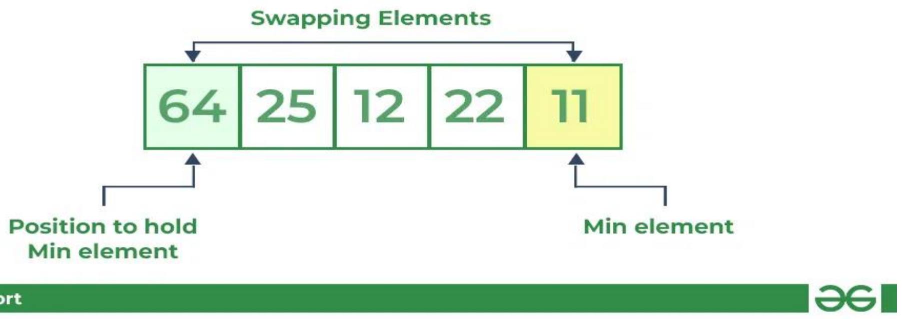
<u>Tracing of Selection Sort Technqiue:</u>

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

First pass:

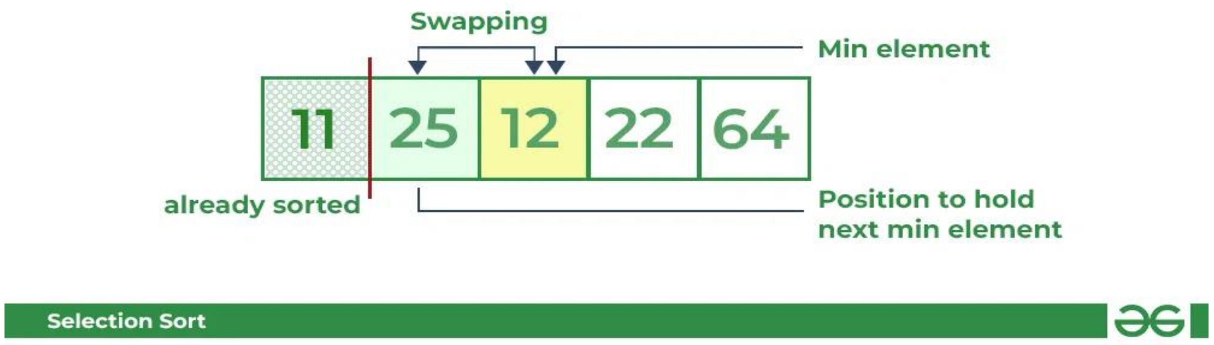
- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where **64** is stored presently, after traversing whole array it is clear that **11** is the lowest value.

- 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.



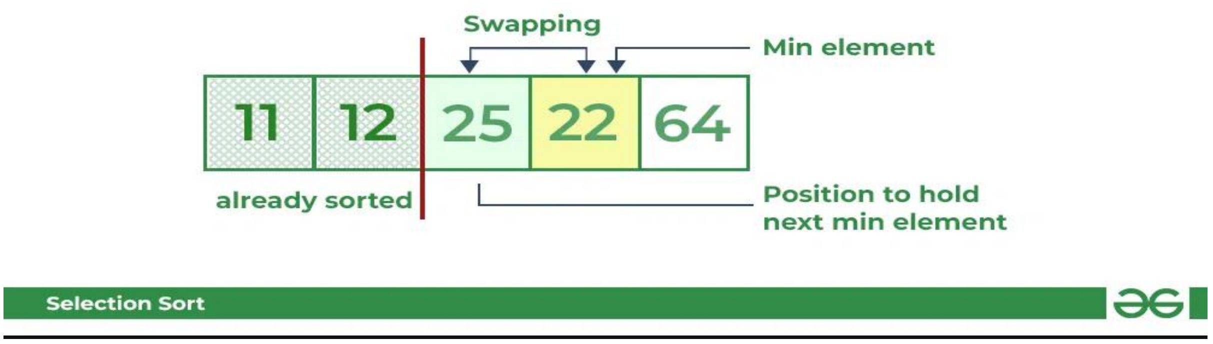
Second Pass:

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



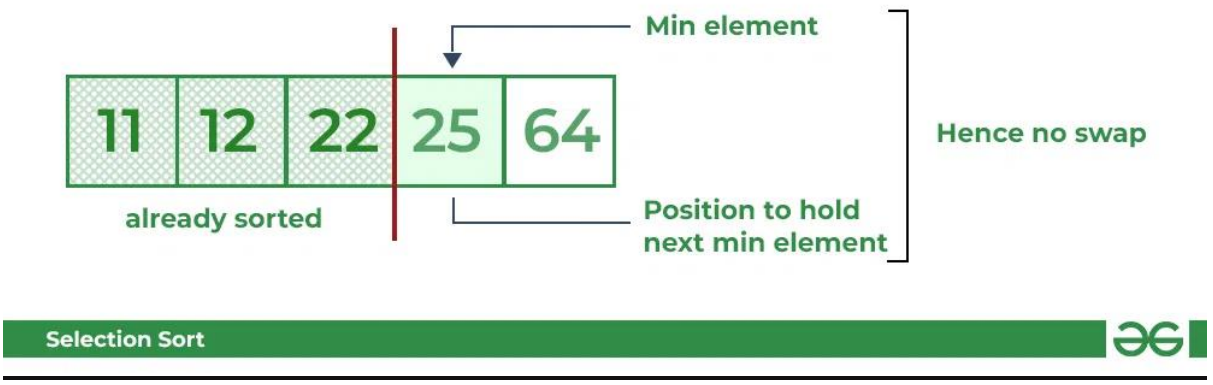
Third Pass:

- 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.
- 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.



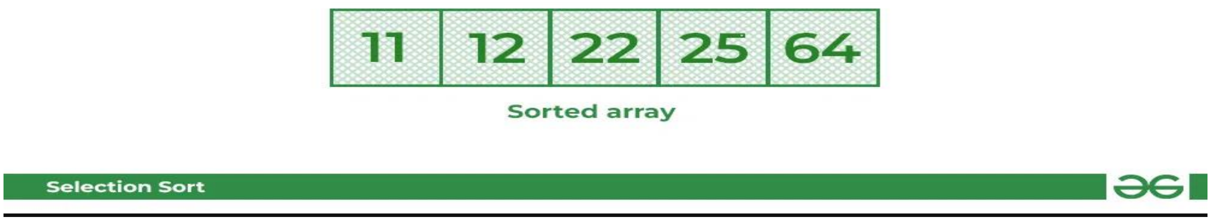
Fourth pass:

- 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.



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.



Insertion Sort: (Sorting by Insertion)

In this method the elements are inserted at their appropriate place. Hence is the name insertion sort.

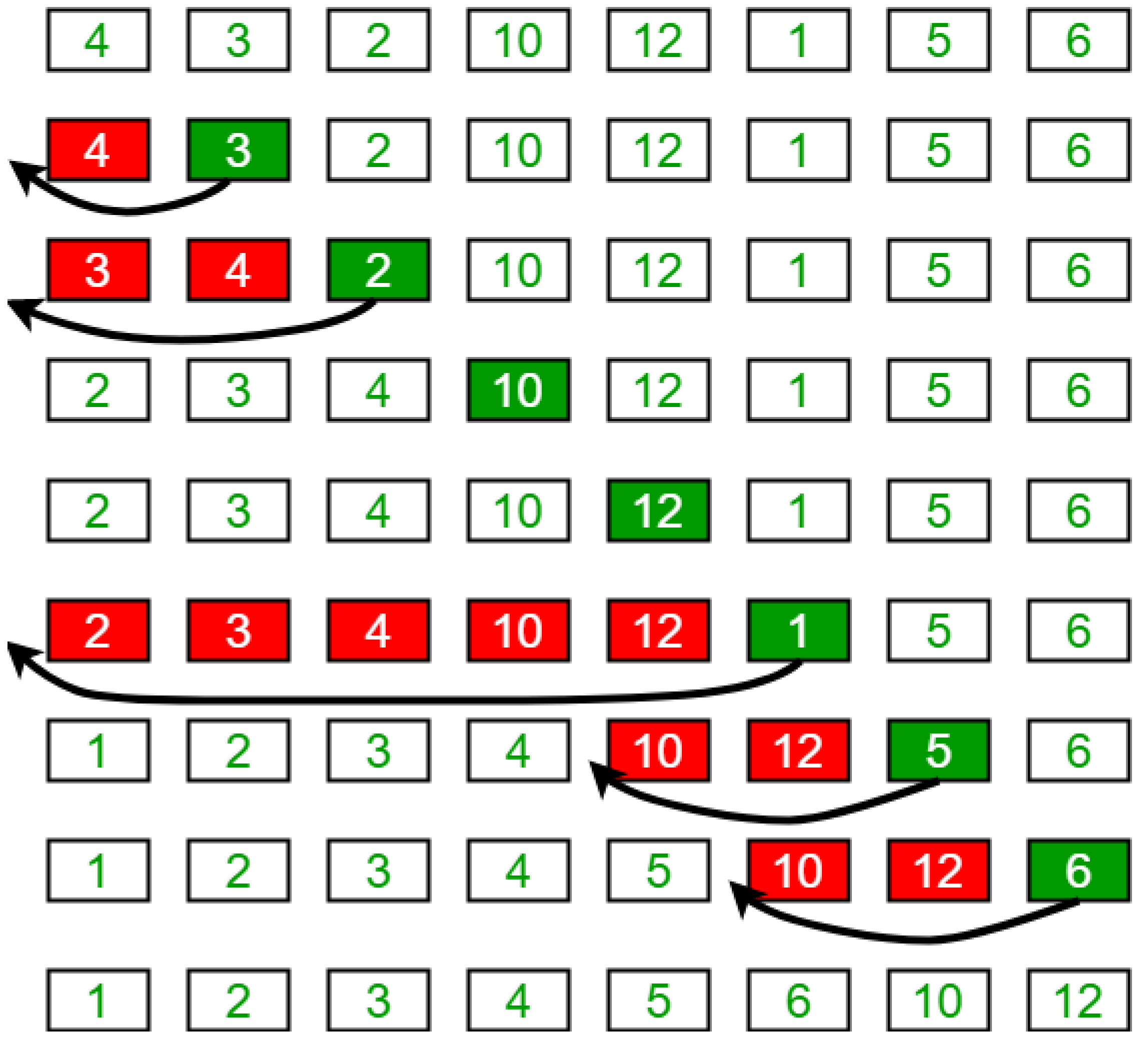
Advantages of insertion Sort:

- Simple to implement
- This method is efficient when we want to sort small number of elements.
- More efficient than most other algorithms such as selection and bubble sort
- This is stable (does not change the relative order of equal elements)
- It is called in-place sorting algorithm (only require a constant amount O(1) of extra memory space). The in-place sorting algorithm is an algorithm in which the input is overwritten by output and to execute the sorting method it does not require any more additional space.

Program for Insertion Sort:

```
#include<stdio.h> #include<conio.h>
void main()
{
int a[20],n,i;
void insert(int a[],int n);
clrscr();
printf("\nEnter N Value\n");
scanf("%d",&n);
printf("\nEnter %d values\n",n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
insert(a,n);
printf("\nSorted List After Insertion Sort\n");
for(i=0;i<n;i++)
printf("%4d",a[i]);
getch();
}
void insert(int a[20],int n)
{
int  i, j, temp;
for(i=1;i<=n-1;i++)
{         temp=a[i]; j=i-1;
          while(j>=0 && a[j]>temp)
          {
          a[j+1]=a[j]; j=j-1;
          }
          a[j+1]=temp;
}
}
```

Insertion Sort Execution Example


4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12