LECTURE NOTES

Name of the Staff:: M Santhi

MODULE – 1

INTRODUCTION TO DATA STRUCTURES

Introduction: Overview of Data Structures

1.1 Definition of Data structure:

Data structure is a way to store and organize data in a computer, so that it can be used efficiently.

1.1.1Introduction of Data structures and algorithms:

The term DATA STRUCTURE is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

- **Data:** Data means a value or set of values.
- **Information:** It means processed data or meaningful data.

Data	Information
33	Age of a person
12-11-1999	Date of birth

- **Data type:** Data type specifies the type of data stored in a variable.

  Example: data types would be integer, floats, string and characters.
- **Built-in data type**: In every programming language there is a set of data types called built in data type.

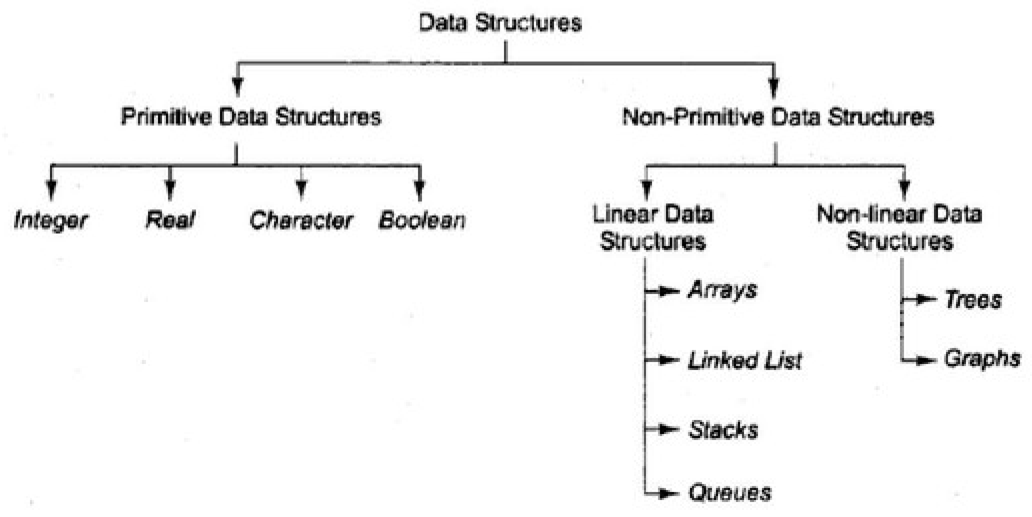
Example: Pascal: Integers, real, char, etc

C: int, float

- **Abstract Data type (ADT):** An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.

  Example: Lists, stacks and graphs are examples of ADT along with their operations.

1.2 Classification of Data structures (Types of Data structures):



Data structures are classified into two types: They are Primitive and non-primitive Data structures.

<u>**1.2.1 Primitive Data structure:**</u>

The primitive data types are the basic data types that are available in most of the programming languages. The primitive data types are used to represent single values.

- **Integer:** This is used to represent a number without decimal point.

  Eg: 12, 90
- **Float and Double:** This is used to represent a number with decimal point.

  Eg: 45.1, 67.3
- **Character :** This is used to represent single character

  Eg: ‘C’, ‘a’

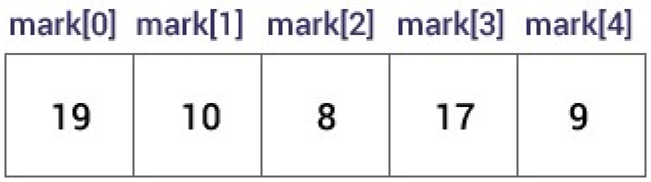
- **String:** This is used to represent group of characters.
  Eg: "M.S.P.V.L Polytechnic College"
- **Boolean:** This is used represent logical values either true or false.

1.2.2 Non-primitive Data structure:

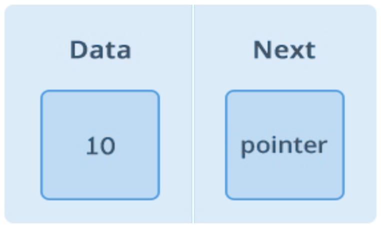
The data types that are derived from primitive data types are known as non-Primitive data types. These data types are used to store group of values. Non- Primitive data types are classified into two types. They are linear data structure and non-linear data structure.

- Linear Data structure:
  - Arrays
  - linked list
  - Stacks
  - Queue
- Non-Linear Data structure:
  - Trees
  - Graphs

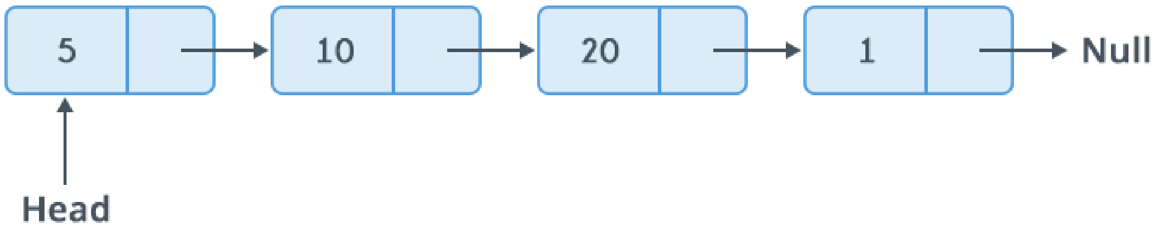
a) **Arrays:** An array is a collection of data that holds fixed number of values of same type.
Example: int mark[5] = {40, 60, 80, 70, 90}



b) **Linked list:** A **linked list** is a way to store a collection of elements. Each element in a linked list is stored in the form of a **node**. A **data** part stores the element and a **next** part stores the link to the next node.



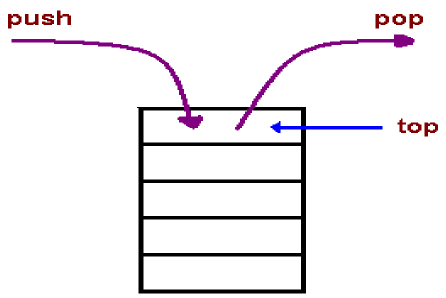
Linked List:



c) **Stack:** Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

The following are the basic operations in stack:

- **Push:** Adds an item in the stack.
- **Pop:** Removes an item from the stack.



d) **Queue:** Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO).

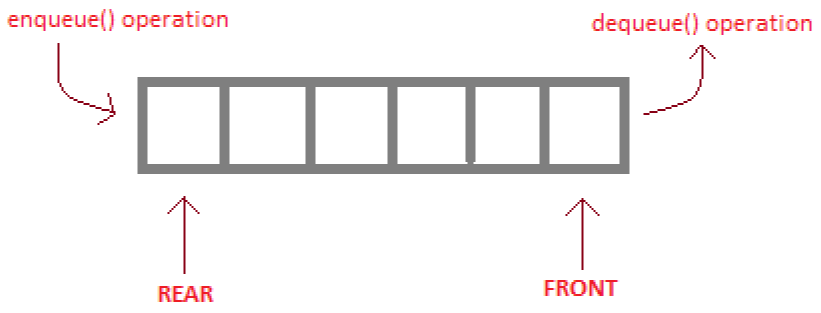
The following ate the four basic operations are performed on queue:

Enqueue: Adds an item to the queue.

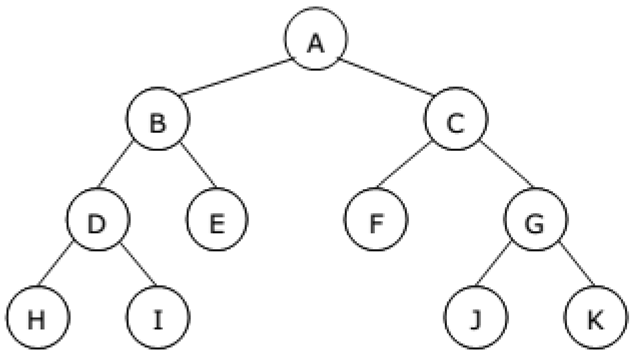
Dequeue: Removes an item from the queue

Front: Get the front item from queue.

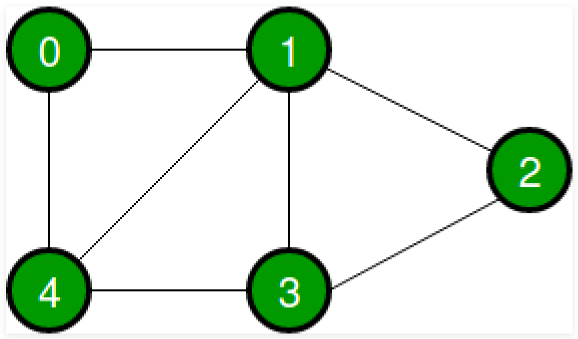
Rear: Get the last item from queue.



e) Trees: Tree represents the nodes connected by edges. The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent.



f) Graphs: Graphs are used to represent networks. Graph is a data structure that consists of following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v.



IMPLEMENTATION OF DATA STRUCTURES

ARRAY MPLEMENTATION

Following program traverses and prints the elements of an array:

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int i = 0,n=5;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

STACK IMPLEMENTATION

```
#include<stdio.h>

#include<stdlib.h>

#define Size 4

int Top=-1, inp_array[Size];
void Push();
void Pop();
void show();

int main()
{
    int choice;

    while(1)
    {
        printf("\nOperations performed by Stack");
        printf("\n1.Push the element\n2.Pop the element\n3.Show\n4.End");
        printf("\n\nEnter the choice:");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: Push();
                    break;
            case 2: Pop();
                    break;
            case 3: show();
                    break;
            case 4: exit(0);

            default: printf("\nInvalid choice!!");
        }
    }
}

void Push()
{
    int x;

    if(Top==Size-1)
    {
        printf("\nOverflow!!");
    }
    else
    {
        printf("\nEnter element to be inserted to the stack:");
        scanf("%d",&x);
        Top=Top+1;
        inp_array[Top]=x;
    }
}

void Pop()
{
    if(Top==-1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nPopped element:  %d",inp_array[Top]);
        Top=Top-1;
    }
}

void show()
{


    if(Top==-1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nElements present in the stack: \n");
        for(int i=Top;i>=0;--i)
            printf("%d\n",inp_array[i]);
    }
}
```

Output:

```
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:1

Enter element to be inserted to the stack:10

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:3

Elements present in the stack:
10

Operations performed by Stack
1.Push the element
```

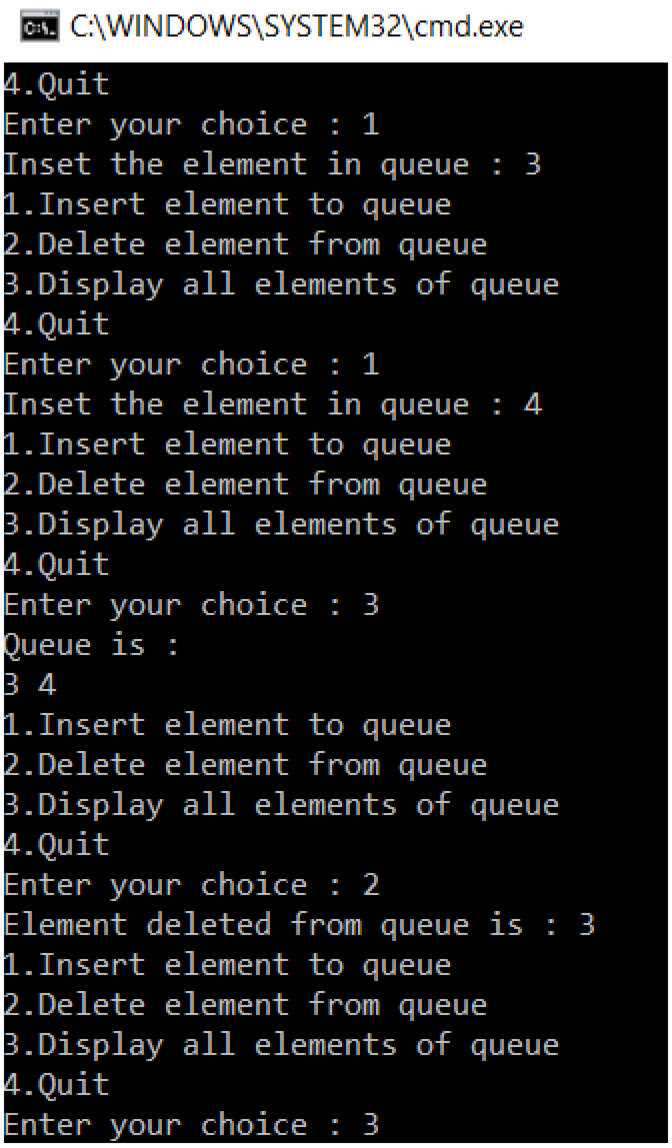
```
2.Pop the element
3.Show
4.End

Enter the choice:2
Popped element:  10

Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice:3
Underflow!!
```

QUEUE IMPLEMENTATION

```
#include <stdio.h>
#include<stdlib.h>
#define MAX 50
void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
int main()
{
int choice;
while (1)
{
printf("1.Insert element to queue n");
printf("2.Delete element from queue n");
printf("3.Display all elements of queue n");
printf("4.Quit n");
printf("Enter your choice : ");
scanf("%d", &choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(1);
default:
printf("Wrong choice n");
}
}
}
void insert()
{
int item;
if(rear == MAX - 1)
printf("Queue Overflow n");
else
{
if(front== - 1)
front = 0;
printf("Inset the element in queue : ");
scanf("%d", &item);
rear = rear + 1;
queue_array[rear] = item;
}
}
void delete()
{
if(front == - 1 || front > rear)
{
printf("Queue Underflow n");
return;
}
else
{
printf("Element deleted from queue is : %dn", queue_array[front]);
front = front + 1;
}
}
void display()
{
int i;
if(front == - 1)
printf("Queue is empty n");
else
{
```

```
printf("Queue is : n");
for(i = front; i <= rear; i++)
printf("%d ", queue_array[i]);
printf("n");
}
}
```

Output



C:\WINDOWS\SYSTEM32\cmd.exe

```
4.Quit
Enter your choice : 1
Inset the element in queue : 3
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 4
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
3 4
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 3
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
```

ALGORITHM SPECIFICATIONS

ALGORITHM:

DEFNITION: An algorithm is a step-by-step procedure of solving a given problem statement in finite number of steps.

The properties of an algorithm are:

Input: Algorithm should be accepting 0 or more inputs supplied externally.

Output: Algorithm should be generating at least one output.

Definiteness: Each step of an algorithm must be precisely defined. Meaning the step should perform a clearly defined task without much complication.

Finiteness: An algorithm must always terminate after a finite number of steps.

Effectiveness: The efficiency of the steps and the accuracy of the output determine the effectiveness of the algorithm.

Correctness: Each step of the algorithm must generate a correct output.

An Algorithm is expressed generally as flow chart or as an informal high level description called as pseudocodeAlgorithm can be defined as "a sequence of steps to be performed for getting the desired output for a given input."

Basic rules followed while designing algorithms are:

- START / BEGIN statement is used to indicate beginning of the algorithm.
- STOP / END statement is used to indicate ending of the algorithm.
- READ / INPUT statement is used for input statements.
- WRITE / OUTPUT statement is used for output statements.
- ← Symbol is used to assign values to the variables.
- RETURN statement is used to return back from either procedure or function.

Examples: Design algorithm for the following problem statements.

1. Addition of given two numbers.
2. Addition, Subtraction, Multiplication and Division of given two numbers.
3. Average of given three numbers.
4. Swapping of given two numbers.

1. Addition of given two numbers	2. Addition, Subtraction, Multiplication and Division of given two numbers
Step 1: START Step 2: READ x, y	Step 1: START Step 2: READ x, y Step 3: Sum ← x + y Sub ← x – y

ALGORITHM TYPES:

In general, the steps in an algorithm can be divided into three basic categories as:

a) Sequence algorithm
b) Selection algorithm
c) Iteration algorithm

a) Sequence algorithm:

A sequence algorithm is a series of steps in sequential order without any break. Here, instructions are executed from top to bottom without any disturbances.

Example: Algorithm for addition of given two numbers

```
Step 1:     START
Step 2:     READ x, y
Step 3:     sum ← x + y
Step 4:     WRITE sum
Step 5:     STOP
```

b) Selection algorithm:

Steps of an algorithm are designed by selecting appropriate condition checking is called as selection algorithms. Selection algorithms are designed using selection control statements such as IF, IF-ELSE, Nested IF-ELSE, ELSE-IF Ladder and SWITCH statements.

Example: Algorithm for maximum of given three numbers

```
Step 1:     START
Step 2:     READ x, y and z values
Step 3:     IF x>y AND x>z THEN
                Max ← x
            ELSEIF y>z THEN
                Max ← y
            ELSE
                Max ← z
            ENDIF
Step 4:     WRITE Max
Step 5:     STOP
```

c) Iteration algorithm:

Steps of an algorithm are designed based on certain conditions and repeatedly processed the same statements until the specified condition becomes false is called as iteration algorithms. Iteration algorithms are designed using iterative control statements such as WHILE, D0-WHILE and FOR statements.

Example: Algorithm for reverse of a given number

```
Step 1:    START
Step 2:    READ n value
Step 3:    rev ← 0
Step 4:    Repeat WHILE n > 0
               k ← n MOD 0
               rev ← rev * 10 + k
               n ← n / 10
           EndRepeat
Step 5:    WRITE rev
Step 6:    STOP
```

ANALYSIS OF AN ALGORITHM

Analysis of algorithms (or) performance analysis refers to the task of determining how much computing time (time complexity) and storage (space complexity) of an algorithm requires.

Algorithm efficiency describes the properties of an algorithm which relates to the amount of resources used. An algorithm must be analyzed to determine its resources usage.

The ***time complexity*** of an algorithm is the amount of computer time it needs to run for its completion. The ***space complexity*** of an algorithm is the amount of memory it needs to run for its completion.

These complexities are calculated based on the size of the input. With this, analysis can be divided into three cases as:

- Best case analysis
- Worst case analysis
- Average case analysis

Best case analysis: In best case analysis, problem statement takes minimum number of computations for the given input parameters.

Worst case analysis: In worst case analysis, problem statement takes maximum number of computations for the given input parameters.

Average case analysis: In average case analysis, problem statement takes average number of computations for the given input parameters.

Based on the size of input requirements, complexities can be varied. Hence, exact representation of time and space complexities is not possible. But they can be shown in some approximate representation using mathematical notations known as **asymptotic notations**.

SPACE COMPLEXITY

The process of estimating the amount of **memory space** to run for its completion is known as space complexity.

Space complexity S(P) of any problem P is sum of fixed space requirements and variable space requirements as:

Space requirement S(P) = Fixed Space Requirements + Variable Space Requirements

1. Fixed space that is independent of the characteristics (Ex: number, size) of the input and outputs. It includes the instruction space, space for simple variables and fixed-size component variables, space for constants and so on.

2. Variable space that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by the referenced variables and the recursion stack space.

When analyzing the space complexity of any problem statement, concentrate solely on estimating the variable space requirements. First determine which instance characteristics to use to measure the space requirements. Hence, the total space requirement S(P) of any program can be represented as:

$$\mathbf{S(P) = C + S_P(I)}$$

Where,
C is a constant representing the fixed space requirements and I refer to instance characteristics.

Example 1:

```
float sum(float a, float b, float c)
{
        return a+b+c;
}
```

Here, the variables a,b and c are simple variables.
Therefore $S_{sum} = 0$.

TIME COMPLEXITY

The process of estimating the amount of **computing time** to run for its completion is known as time complexity.

The time T(P) taken by a program P is the sum of its compile time and its run time.

i.e., Time complexity T(P) = Compile time + Run time

Here,
Compile time is a fixed component and does not depends on the instance characteristics. Hence,

$T(P) = C + T_P$ (Instance characteristics)
Where, C is a fixed constant value

$$\boxed{T(P) \geq T_P(I)}$$

Where, I refer instance characteristic.

Time complexity of a program is calculated by determining the number of steps that a program/function needs to solve known as ***step count*** and then express it in terms of ***asymptotic notations.***

Example 1:

```
float sum(float a, float b, float c)          count = 0
{
        return a+b+c;                          count=count+1
                                               count=count+1
}
```

Here, count variable is incremented by twice one for addition operation and one for return statement.

Therefore $T_{sum} = 2$.

ASYMPTOTIC NOTATIONS

Asymptotic Notations are languages that allow us to analyze an algorithm’s running time by identifying its behavior as the input size for the algorithm increases.

For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n). Usually, the time required by an algorithm falls under three types −

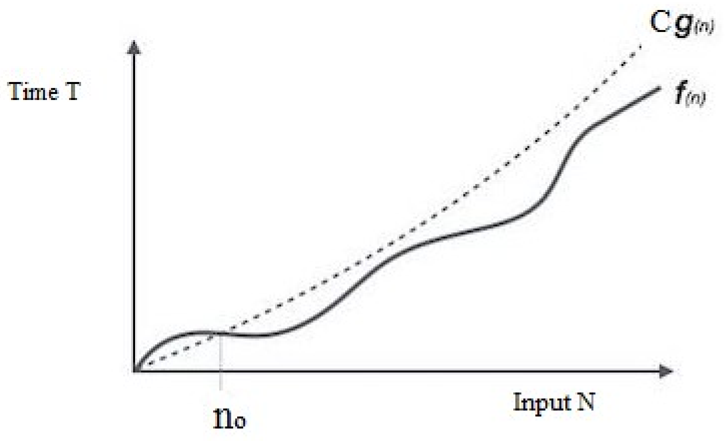
- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

- The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

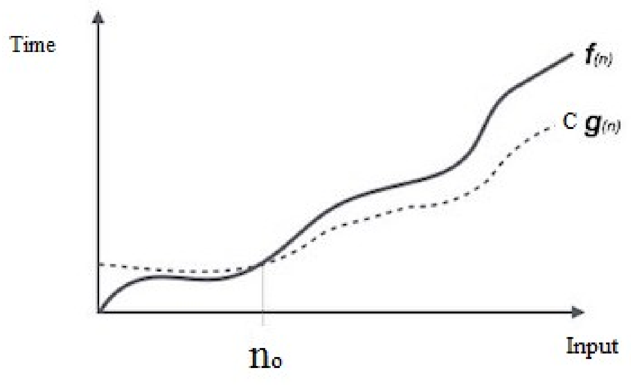


If **f(n) <= C g(n)**for all **n >= n_0**, **C > 0** . Then we can represent **f(n)** as **O(g(n))**.

Chapter 2 f(n) = O(g(n))

Omega Notation, Ω

- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

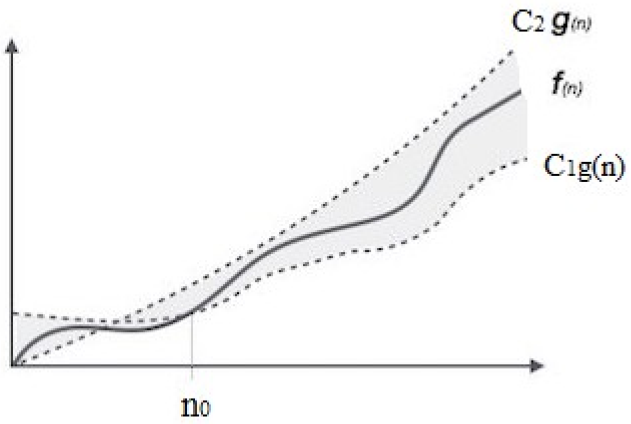


If **f(n) >= C g(n)**for all **n >= n_0**, **C > 0**. Then we can represent **f(n)** as **Ω(g(n))**.

Chapter 3 f(n) = Ω(g(n))

Theta Notation, θ

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



If **C_1 g(n) <= f(n) >= C_2 g(n)** for all **n >= n_0**, **C_1, C_2 > 0**. Then we can represent **f(n)** as **Θ(g(n))**.
Chapter 4 f(n) = Θ(g(n))

TIME-SPACE TRADE OFF

A space–time tradeoff can be **applied to the problem of data storage**. If data is stored uncompressed, it takes more space but access takes less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm).

A tradeoff is a situation where one thing increases and another thing decreases. It is a way to solve a problem in:

- Either in less time and by using more space, or
- In very little space by spending a long amount of time.

The best Algorithm is that which helps to solve a problem that requires less space in memory and also takes less time to generate the output. But in general, it is not always possible to achieve both of these conditions at the same time. The most common condition is an algorithm using a lookup table. This means that the answers to some questions for every possible value can be written down. One way of solving this problem is to write down the entire **lookup table**, which will let you find answers very quickly but will use a lot of space. Another way is to calculate the answers without writing down anything, which uses very little space, but might take a long time. Therefore, the more time-efficient algorithms you have, that would be less space-efficient.

Types of Space-Time Trade-off

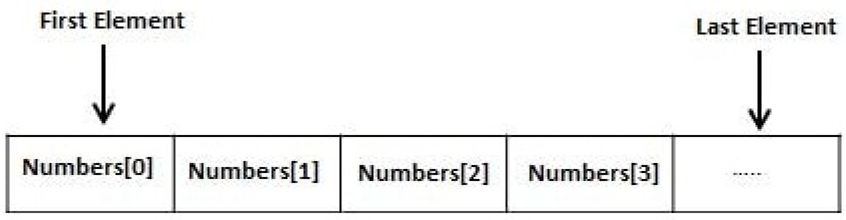
- Compressed or Uncompressed data
- Re Rendering or Stored images
- Smaller code or loop unrolling
- Lookup tables or Recalculation

Compressed or Uncompressed data: A space-time trade-off can be applied to the problem of **data storage**. If data stored is uncompressed, it takes more space but less time. But if the data is stored compressed, it takes less space but more time to run the decompression algorithm. There are many instances where it is possible to directly work with compressed data. In that case of compressed bitmap indices, where it is faster to work with compression than without compression.
Re-Rendering or Stored images: In this case, storing only the source and rendering it as an image would take more space but less time i.e., storing an image in the cache is faster than re-rendering but requires more space in memory.
Smaller code or Loop Unrolling: Smaller code occupies less space in memory but it requires high computation time that is required for jumping back to the beginning of the loop at the end of each iteration. Loop unrolling can optimize execution speed at the cost of increased binary size. It occupies more space in memory but requires less computation time.
Lookup tables or Recalculation: In a lookup table, an implementation can include the entire table which reduces computing time but increases the amount of memory needed. It can recalculate i.e., compute table entries as needed, increasing computing time but reducing memory requirements.

ARRAYS

An array is a collection of data that holds fixed number of values of same type.

Example: int mark[5] = {40, 60, 80, 70, 90}



Declaring Arrays

Syntax: `datatype array_name[size];`

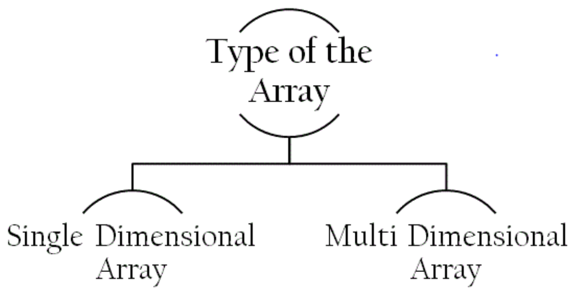
datatype: It denotes the type of the elements in the array.

array_name: it is the name given to an array.

size: It is the number of elements an array can hold.

1.4.1 Types of arrays:

There are two types of arrays. They are a) Single dimensional array b) Multi-dimensional array.



1.4.2 Single Dimensional array:

- Single or One Dimensional array is used to represent and store data in a linear form.
- Array having only one subscript variable is called One-Dimensional array
- It is also called as Single Dimensional Array or Linear Array
- **Syntax:** datatype array_name[size];
- **Example:** int mark[5] = {40, 60, 80, 70, 90};

Operations on Arrays:

The following are the operations that can be performed on arrays:

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Sorting --** It is used to arrange the data items in some order i.e. in ascending or descending order.
- **Merging --** It is used to combine the data items of two sorted files into single file in the sorted form

1.Traverse: print all the array elements one by one.

Example: Consider linear array A as below:

1	2	3	4	5
10	20	30	40	50

A [1] = 10
A [2] = 20
A [3] = 30
A [4] = 40
A [5] = 50

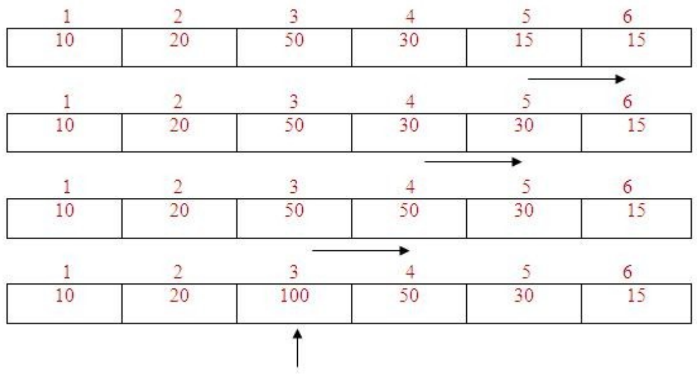
Algorithm:

```
Step 1: START
Step 2: Take an array A
Step 3: Define its values
Step 4: Loop for each value of A
Step 5: Display A[i] where i is the value of current iteration
Step 6: STOP
```

Insertion: It is used to add a new data item in the given collection of data items.
Example: Consider linear array A as below:

1	2	3	4	5	
10	20	50	30	15	

New element to be inserted is 100 and location for insertion is 3. So shift the elements from 5th location to 3rd location downwards by 1 place. And then insert 100 at 3rd location. It is shown below:



Algorithm:

Let **A** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the Kth position of LA

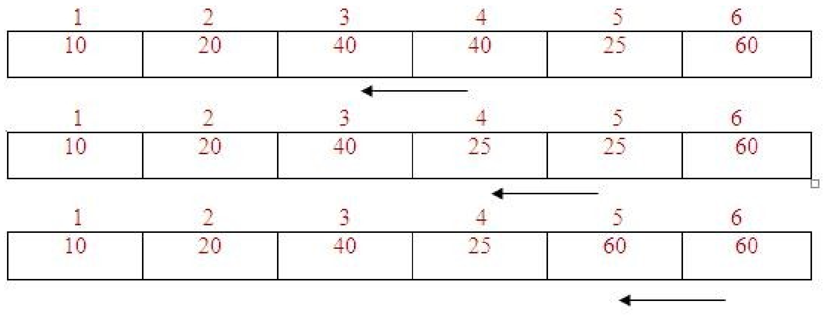
Algorithm

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set A[J+1] = A[J]
6. Set J = J-1
7. Set A[K] = ITEM
8. Stop
```

3.Deletion: It is used to delete an existing data item from the given collection of data items. **Example:**

1	2	3	4	5	
10	20	50	40	25	60

The element to be deleted is 50 which is at 3rd location. So shift the elements from 4th to 6th location upwards by 1 place. It is shown below:



1	2	3	4	5	6
10	20	40	40	25	60

1	2	3	4	5	6
10	20	40	25	25	60

1	2	3	4	5	6
10	20	40	25	60	60

After deletion the array will be:

1	2	3	4	5	6
10	20	40	25	60	

Algorithm:

Consider **A** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of L.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set A[J] = A[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

4.Search: Searches an element using the given index or by the value.

Example:

We have linear array A as below:

1	2	3	4	5
15	50	35	20	25

Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here:

1) Compare 20 with 15

20 # 15, go to next element.

2) Compare 20 with 50

20 # 50, go to next element.

3) Compare 20 with 35

20 #35, go to next element.

4) Compare 20 with 20

20 = 20, so 20 is found and its location is 4.

Algorithm

Consider **A** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF A[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

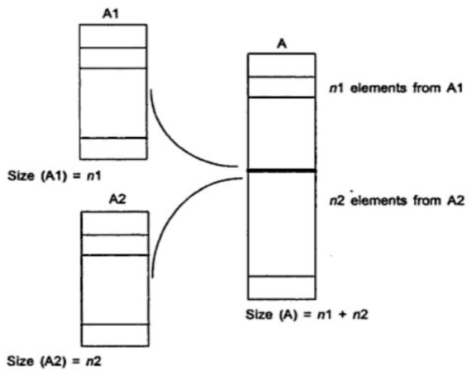
5.Sorting: It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data. E.g.

1	2	3	4	5
10	50	40	20	30

After arranging the elements in increasing order by using a sorting technique, the array will be:

1	2	3	4	5
10	20	30	40	50

6. Merging -- It is used to combine the data items of two sorted files into single file in the sorted form.



Merging of A_1 and A_2 to A.

Multi-Dimensional Arrays

A **multi-dimensional array** is an array that has more than one dimension. It is an array of arrays; an array that has multiple levels. 2-Dimensional array and 3-Dimensional array are examples of Multi-Dimensional arrays.

Syntax: datatype array_name[size 1][size 2]…..[size n];

a) <u>Two-Dimensional Array:</u>

A 2D array is also called a **matrix**, or a table of rows and columns. The simplest form of multidimensional array is the two-dimensional array. Syntax of two-dimensional array:

Syntax: datatype array_name[x][y];

datatype: It denotes the type of the elements in the array.

array_name: it is the name given to an array.

[x] is number of rows

[y] is number of columns

. There are two conventions of storing any matrix in memory:
I. Row-major order
2. Column-major order.

Row-major order: In row-major order, elements of a matrix are stored on a row-by-row basis.

$$\text{Address}\ (a_{ij}) = M + (i - 1) \times n + j - 1$$

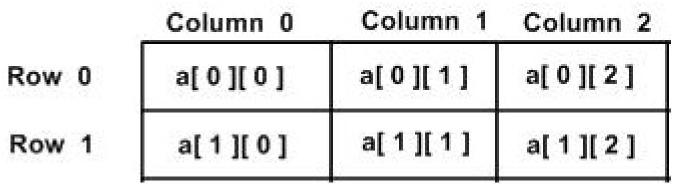
Column-major order: column-major order. elements are stored column-by-column.

$$\text{Address}\ (a_{ij}) = M + (j - 1) \times m + i - 1$$

Example: A two dimensional array 'a' of type int with 2 rows and 3 columns can be defined as:

int a[2][3];

This array will contain 2x3(6) elements and they can be represented as:



	Column 0	Column 1	Column 2
Row 0	a[0][0]	a[0][1]	a[0][2]
Row 1	a[1][0]	a[1][1]	a[1][2]

Where a[0][0] to a[1][2] represent the elements of the array.

b) Initializing two-Dimensional array

Consider the following 2D array:

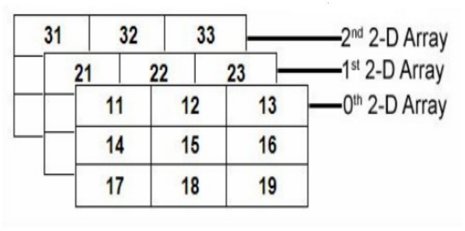
```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

The above 2D array can also represented as: array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
  {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
  {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
  {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

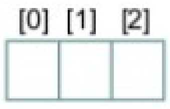
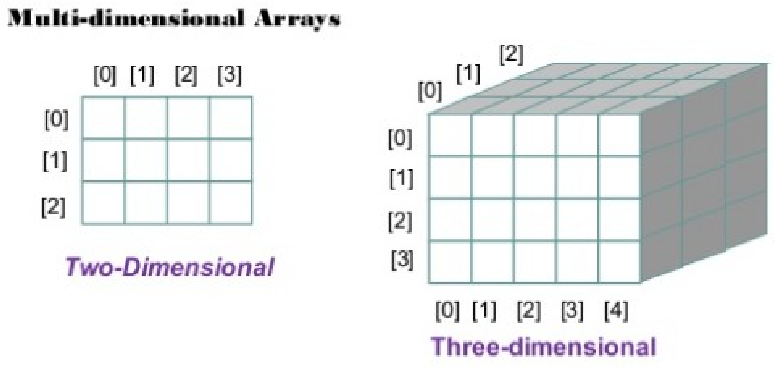
c) Three – Dimensional Array (3D) Array

A three-dimensional (3D) array is an array of arrays of arrays. In C programming an array can have two, three, or even ten or more dimensions. The maximum dimensions a C program can have depends on which compiler is being used.



```
int arr[3][3][3]=
        {
          {11, 12, 13}, {14, 15, 16},{17, 18, 19},
          {21, 22, 23}, {24, 25, 26},{27, 28, 29},
          {31, 32, 33},{34, 35, 36},{37, 38, 39}
        };
```

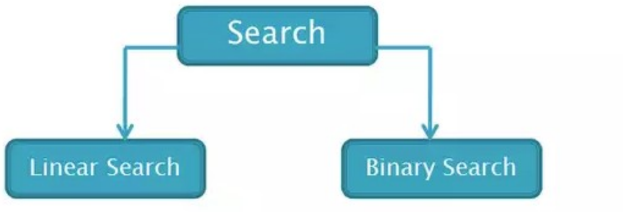
d) Difference between one-dimensional and multi-dimensional

one-dimensional	**multi-dimensional**
1) 1D Stores single list of elements of similar data type.	1) Usually, 2D and 3D are used in Multi-dimensional. 2D Stores 'array of arrays' 3D Stores 'array of array of arrays'
2) Syntax: Datatype arrayname[size];	**2)** Datatype arrayname[size1] [size2];
3) Representation of 1D: 	**3)**  Multi-dimensional Arrays [0] [1] [2] [3] [0] [1] [2] Two-Dimensional [0] [1] [2] [0] [1] [2] [3] [0] [1] [2] [3] [4] Three-dimensional

4) Ex: int a[5] = {10,20,30,40,50};	4) Ex: int a[3][4] = {
	{0,1,2,3},
	{4,5,6,7},
	{8,9,10,11}
	};

MODULE 1 PART II

Searching: Searching is a process of finding a value in a list of values. Searching can be performed by following two techniques.



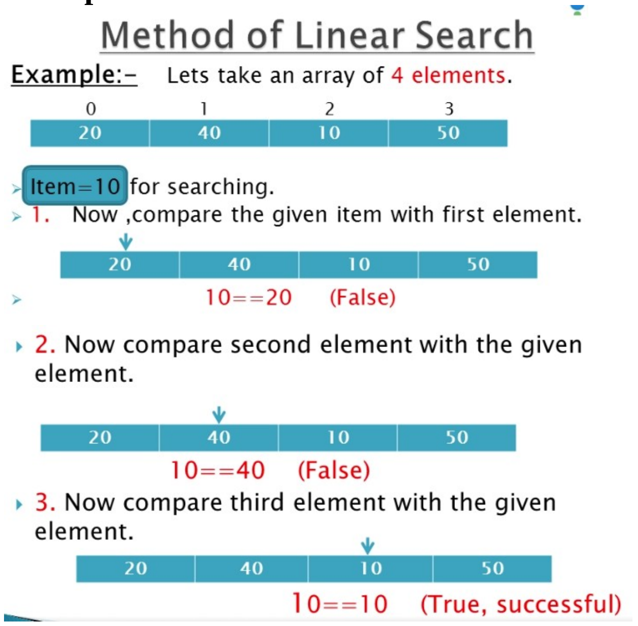
List search

There are two types of list search. They are a) sequential (linear) search b) Binary search

5.1 Sequential search (linear search):

- Linear search is a very simple search algorithm.
- In this, searching starts from beginning of an array and compares each element with the given element and continues until the desired element is found or end of the array is reached.
- Linear search is used for small and unsorted arrays.

Example:


Method of Linear Search
Example:- Lets take an array of 4 elements.
0 1 2 3
20 40 10 50
Item=10 for searching.
1. Now ,compare the given item with first element.
20 40 10 50
10==20 (False)
2. Now compare second element with the given element.
20 40 10 50
10==40 (False)
3. Now compare third element with the given element.
20 40 10 50
10==10 (True, successful)

Algorithm:

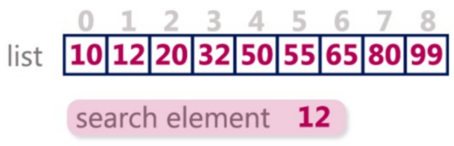
```
Step 1: Linear Search ( Array A, Value x)
Step 2: Set i=1
Step 3: if (A[i] == x) then
        Print "search is successful and x is found at index i"
        stop
Step 4: else
        i = i + 1
        if ( i ≤ n ) then go to step 3
```

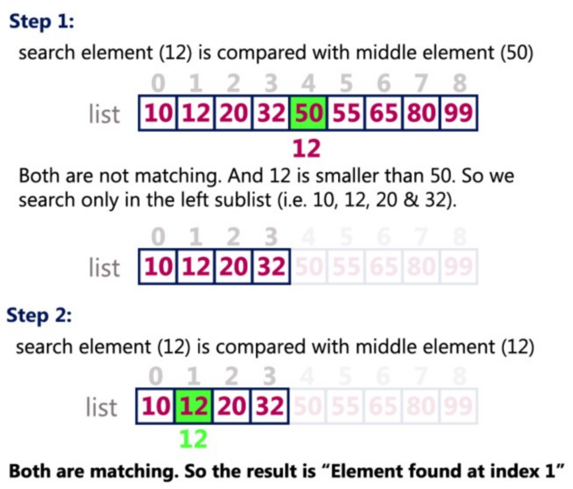
```
Step 5: else
        Print "unsuccessful"
        stop
```

5.2 Binary search:

- Binary search is a fast search algorithm with run-time complexity of O(log n).
- Binary search algorithm works on the principle of divide and conquer.
- Binary search is used for sorted arrays.
- In this, searching starts at middle of the array.
- Search element = middle element (search successful)
- Search element < middle element (search in left sub-list)
- Search element > middle element (search in right sub-list)

Example:



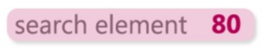


Step 1:
search element (12) is compared with middle element (50)

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:
search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"



Pseudocode/ Algorithm:

```
Step 1: Binarysearch(list, key, low, high)
Step 2: if (low ≤ high) then
            Mid = (low + high)/2
Step 3: if(List[mid] = key) then
            return mid              // search success
Step 4: else if(key < list[mid]) then
            return Binarysearch(list, key, low, mid-1)      // search left sub-list
Step 5: else return Binarysearch(list,key,mid + 1, high)  // search right sub-list
Step 6: end if
Step 7: end if
```

Analysing search algorithm:
Analysing search algorithm means which algorithm is efficient for searching.

Sequential search: In this, searching starts from beginning of an array and compares each element with the given element and continues until the desired element is found or end of the array is reached.

Algorithm:

```
Step 1: Linear Search ( Array A, Value x)
Step 2: Set i=1
Step 3: if (A[i] == x) then
        Print "search is successful and x is found at index i"
        stop
Step 4: else
        i = i + 1
        if ( i ≤ n ) then go to step 3
Step 5: else
        Print "unsuccessful"
        stop
```

Time complexity of linear search:

Unsuccessful search: O(n)
Successful search:
Best-case: Item is in the first location of an array = O(1)
Worst-case: Item is in the last location of an array = O(n)
Average case: The sum of comparisons 1,2,…….n = O(n+1)/2

Binary search: In binary search no need of searching entire list because of if target element is greater than mid value search only right of the list. if target is less than mid value search only left half the list.

```
Step 1: Binarysearch(list, key, low, high)
Step 2: if (low ≤ high) then
        Mid = (low + high)/2
Step 3: if(List[mid] = key) then
        return mid                 // search success
Step 4: else if(key < list[mid]) then
        return Binarysearch(list, key, low, mid-1)        // search left sub-list
Step 5: else return Binarysearch(list,key,mid + 1, high)  // search right sub-list
Step 6: end if
Step 7: end if
```

Time complexity of Binary search:

Unsuccessful search: O($\log_2 n$)

Successful search:
Best-case: no. of iterations is 1 = O(1)
Worst-case: no. of iterations = $O(\log_2 n)$
Average case: no. of iterations = $O(\log_2 n)$

No. of items	Linear	Binary
16	16	4
64	64	6
256	256	8
1024	1024	10
16,384	16,384	14
131,072	131,072	17
262,144	262,144	18
524,288	524,288	19

From the above comparison, binary search algorithm has less number of comparisons and it is more efficient than linear search algorithm.

<u>Here's the information on data structure applications with a bit more detail:</u>

- **Arrays:** Imagine an array like a box with a set number of compartments. Each compartment can hold one item, and they all line up in a specific order. This makes arrays efficient for accessing items by their position, like finding a specific name on a pre-populated list. Video games use arrays to store level data, where enemy positions or item locations need to be quickly referenced.

- **Linked Lists:** Unlike an array's box, a linked list is more like a chain. Each link holds data and a reference (like an address) pointing to the next link in the chain. This allows the list to grow or shrink on the fly, making it useful for things like managing music playlists where songs can be added or removed easily. Text editors also use linked lists to implement undo/redo functionality, where each keystroke is a new link added to the chain.

- **Stacks:** Think of a stack like a stack of plates. The plate you just put on top is the one you can grab first. This "Last In, First Out" (LIFO) principle is ideal for keeping track of function calls in a program. Imagine a function calling another function, which in turn might call a third function. The stack remembers these calls and ensures they all finish processing in the correct order. Many software applications use stacks to implement the "undo" button, where the last action is the first to be undone.

- **Queues:** Queues are the opposite of stacks. Imagine a line at a store - the person who joined the line first gets served first, following a "First In, First Out" (FIFO) principle. This makes queues perfect for managing tasks in a computer system, where jobs are processed in the order they are received. Similarly, simulations of lines waiting for service, like customers waiting for a cashier, often use queues.

- **Trees:** Trees are like organizational charts, with a root node at the top and branches extending downwards. Each branch can have sub-branches, creating a hierarchical structure. This is ideal for storing sorted data efficiently. Imagine a dictionary - words are organized alphabetically, like branches on a tree. Since the tree is sorted, searching for a specific word is much faster than searching a plain list. Computer scientists also use trees for super-efficient search algorithms like binary search.

- **Graphs:** Graphs connect things together, like a social network. Imagine users as circles (nodes), and friendships between them as lines (edges) connecting the circles. Graphs are powerful tools for modeling real-world connections. Navigation systems use graphs to show road connections between cities, allowing you to find the fastest route. Social media platforms leverage graphs to connect people with their friends and recommend new connections.

Linear vs binary

Linear Search vs. Binary Search

Feature	Linear Search	Binary Search
Search Method	Examines each element sequentially	Divides the search space in half with each comparison
Data Structure	Unordered list	Sorted list
Time Complexity	O(n) - Worst case, examines all elements	O(log n) - Faster as data size grows
Average Case Performance	Depends on data distribution	Usually faster than linear search
Best Case Performance	O(1) - If target element is the first element	O(1) - If target element is the middle element
Ease of Implementation	Simpler to implement	More complex to implement
Space Complexity	O(1) - Constant extra space	O(1) - Constant extra space

Use Cases	Suitable for small datasets or unsorted data	Ideal for large and sorted datasets