RESEARCH INNOVATION DISCOVERY

Reg.No: 048884
Foundation Day
05-09-2024

RID संस्था समस्या का समाधान

RUN BY TWKSAA WELFARE FOUNDATION

Computer Fundamentals

E-Book

Er. Rajesh Prasad(B.E, M.E)
Founder: TWF & RID Org.

- **RID ORGANIZATION** यानि **Research, Innovation and Discovery** संस्था जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही मैं राजेश प्रसाद **इस RID संस्था** की स्थपना किया हूँ |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

"त्वक्सा कंप्यूटर फंडामेंटल के इस ई-पुस्तक में आप कंप्यूटर से जुड़ी सभी बुनियादी अवधारणाएँ सीखेंगे। । मुझे आशा है कि इस ई -पुस्तक को पढ़ने के बाद आपके ज्ञान में वृद्धि होगी और आपको कंप्यूटर विज्ञान के बारे में और अधिक जानने में रुचि होगी"

In this E-Book of Computer Fundamentals, you will learn all the basic concepts related to computers. I hope after reading this E-Book your knowledge will be improve and you will get more interest to know more thing about computer Science.

RID हमें क्यों करना चाहिए ?

(Research)

अनुसंधान हमें क्यों करना चाहिए ?

Why should we do research?

1. नई ज्ञान की प्राप्ति (Acquisition of new knowledge)
2. समस्याओं का समाधान(To Solving problems)
3. सामाजिक प्रगति (To Social progress)
4. विकास को बढ़ावा देने(To promote development)
5. तकनीकी और व्यापार में उन्नति(To advances in technology & business)
6. देश विज्ञान और प्रौद्योगिकी के विकास(To develop the country's science & technology)

(Innovation)

नवीनीकरण हमें क्यों करना चाहिए ?

Why should we do Innovation?

1. प्रगति के लिए(To progress)
2. परिवर्तन के लिए(For change)
3. उत्पादन में सुधार(To Improvement in production)
4. समाज को लाभ(To Benefit to society)
5. प्रतिस्पर्धा में अग्रणी (To be ahead of competition)
6. देश विज्ञान और प्रौद्योगिकी के विकास।
(To develop the country's science & technology)

(Discovery)

खोज हमें क्यों करना चाहिए?

Why should we do Discovery?

1. नए ज्ञान की प्राप्ति(Acquisition of new knowledge)
2. अविष्कारों की खोज(To Discovery of inventions)
3. समस्याओं का समाधान(To Solving problems)
4. ज्ञान के विकास में योगदान(Contribution to development of knowledge)
5. समाज के उन्नति के लिए (for progress of society)
6. देश विज्ञान और तकनीक के विकास
(To develop the country's science & technology)

❖ Research(अनुसंधान):

- अनुसंधान एक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिज़ाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

❖ Innovation(नवीनीकरण): -

- Innovation एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

नोट : अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

सुविचार:	
1.	**समस्याओं का समाधान करने का उतम मार्ग हैं |** → शिक्षा ,RID, प्रतिभा, सहयोग, एकता एवं समाजिक-कार्य
2.	**एक इंसान के लिए जरूरी हैं |** → रोटी, कपड़ा, मकान, शिक्षा, रोजगार, इज्जत और सम्मान
3.	**एक देश के लिए जरूरी हैं |** - → संस्कृति-सभ्यता, भाषा, एकता, आजादी, संविधान एवं अखंडता
4.	**सफलता पाने के लिए होना चाहिए |** → लक्ष्य, त्याग, इच्छा-शक्ति, प्रतिबद्धता, प्रतिभा, एवं सतता
5.	**मरने के बाद इंसान छोडकर जाता हैं |** → शरीर, अन-धन, घर-परिवार, नाम, कर्म एवं विचार
6.	**मरने के बाद इंसान को इस धरती पर याद किया जाता हैं उनके**
➔ नाम, काम, दान, विचार, सेवा-समपर्ण एवं कर्मो से...	

आशीर्वाद (बड़े भैया जी)

Mr. RAMASHANKAR KUMAR

मार्गदर्शन एवं सहयोग

Mr. GAUTAM KUMAR

.....सोच है जिनकी नये कुछ कर दिखाने की, खोज हैं मुझे आप जैसे इंसान की.....

"अगर आप भी **Research, Innovation and Discovery** के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें एवं अपनी समस्या का समाधान **RID** के माध्यम से करना चाहतें हैं तो **RID ORGANIZATION (रीड संस्था)** से जरुर जुड़ें" || धन्यवाद || **Er. Rajesh Prasad (B.E, M.E)**

Index

DATA

❖ What is Data?

- Data is a collection of facts, statistics, or information that is represented in a structured or unstructured format. (डेटा तथ्यों, आँकड़ों या सूचनाओं का एक संग्रह है जिसे संरचित या असंरचित प्रारूप में दर्शाया जाता है।)
- In simple words data can be facts related to any object.
- Data usually refers to raw data, or unprocessed data just collected from different sources.

Example: Text data, Numeric data, Image data, Audio data, Video data, Geospatial data etc.

WHAT IS DATA STRUCTURE?

- Data structure is a way of organizing and storing data in a computer's memory.
- **डेटा संरचना कंप्यूटर की मेमोरी में डेटा को व्यवस्थित और संग्रहीत करने का एक तरीका है।**
- Data structures provide a systematic way of managing and arranging data elements to perform various operations effectively, such as **Insertion, Deletion, Searching, Sorting, Accessing/Retrieving, Updating/Modifying, Merging/Joining, Splitting/Partitioning.**

Use:

- ✓ Data structures are used to organize and manage data efficiently in computer science and software development.
- ✓ Data Structures play a vital role in designing algorithms, storing information in **databases**, managing memory in **operating systems**.

Terms:

- **Data:** a collection of values.
- **Data Items:** A Single unit of value is known as Data Item.
- **Group Items:** Data Items that have subordinate data items are known as Group Items.
- **Elementary Items:** it is Data Items that are unable to divide into sub-items
- **Entity and Attribute:** A class of certain objects is represented by an Entity.

❖ What is Data Base:

- Data base is a systematic collection of data. (डेटा बेस डेटा का एक व्यवस्थित संग्रह है।)
- Database is a structured collection of data organized and stored in a way that allows for efficient storage, retrieval, and manipulation of data.

Example:

- ✓ Employee Database, Student Records Database, Social Media Database, Banking Database, Flight Booking Database, Healthcare Database etc.

❖ What is DBMS (Database Management System):

- DBMS is a software system that enables users to interact with databases and efficiently manage, organize, store, retrieve, and manipulate data. (DBMS एक सॉफ्टवेयर सिस्टम है जो उपयोगकर्ताओं को डेटाबेस के साथ इंटरैक्ट करने और डेटा को कुशलतापूर्वक प्रबंधित, व्यवस्थित, संग्रहीत, पुनर्प्राप्त और हेरफेर करने में सक्षम बनाता है।)

Example:

- ✓ **MySQL:** MySQL AB (acquired by Oracle Corporation in 2010) - Published in 1995.
- ✓ **Oracle Database:** Oracle Corporation - Published in 1979.
- ✓ **Microsoft SQL Server:** Microsoft Corporation - Published in 1989.
- ✓ **PostgreSQL:** PostgreSQL Global Development Group - Published in 1996.
- ✓ **MongoDB:** MongoDB, Inc. - Published in 2009.
- ✓ **SQLite:** D. Richard Hipp (public domain) - First released in 2000.
- ✓ **IBM Db2:** IBM - Published in 1983.
- ✓ **MariaDB:** MariaDB Corporation (original developers of MySQL) - Published in 2009.
- ✓ **Amazon DynamoDB:** (AWS), a subsidiary of Amazon- Published in 2012.
- ✓ **Cassandra:** - Apache Software Foundation in 2008.

TYPES OF DATA STRUCTURE

❖ There are two types of data structures:

1). Primitive data structure
2). Non-primitive data structure

1). Primitive data structure:

- Primitive data types, also known as basic data types. Primitive data types are standard predefined types.

Example:

- ✓ Integer, Floating-Point, Character, Boolean, Void (used mainly as a return type) and pointer.
- Basic data types can store only one individual piece of data at a time. These data types are atomic and represent a single value of a particular kind.
  (Atomic data types means indivisible data units)
- They cannot directly hold collections or multiple elements of data.

Example:

- ✓ **Integer:** A data type to represent whole numbers **(e.g., 3,6,9 -5,-3,-6,-9, 100).**
- ✓ **Floating-Point:** It's represent real numbers with fractional parts **(e.g. 3.14, -6.001, 9.718).**
- ✓ **Character:** A data type to represent individual characters **(e.g., 'A', 'b', 'T', '@').**
- ✓ **Boolean:** A data type to represent binary values of **true or false.**
- ✓ **void:** A special primitive data type that indicates the absence of any data type or value.
- ✓ **Pointer:** it stores memory addresses that point to the location of other data types in memory.

2). Non-primitive data structure:

- It can hold multiple values or it is collections of data elements that can hold multiple values.
- Multiple value may be collection of primitive or non-primitive data types.
- Non-primitive data types, also known as composite or user-defined data types.

Example:

- Arrays, Linked Lists, Trees, Graphs, Hash Tables, Stacks, Queues, Sets, Maps, Graphical Data Structures, Linked Hash Maps.

The non-primitive data structure is divided into two types:

1). Linear data structure
2). Non-linear data structure

1). Linear data structure:

- The arrangement of data in a sequential manner is known as a linear data structure.

2). Non-linear data structure:

- Data elements are not arranged in a sequential, linear order.

❖ Linear Data Structure:

- A linear data structure is a type of data structure in which data elements are arranged sequentially, one after the other, in a linear order.
- This means that each element has a unique predecessor and successor, except for the first and last elements.

❖ Characteristics:

- characteristics of a linear data structure includes operations like insertion, deletion, searching, accessing, updating, traversal, and merging, allowing for efficient organization, manipulation, and retrieval of data in a sequential order.

❖ **Example:**

- ✓ **Arrays:** An Array is a data structure used to collect multiple data elements of the same data type into one variable. accessed using an index.
- ✓ **Linked Lists:** used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers.
- ✓ **Stacks:** A Last-In-First-Out (LIFO) data structure where elements are inserted and removed from the same end (the top).
- ✓ **Queues:** A First-In-First-Out (FIFO) data structure where elements are inserted at the rear and removed from the front.
- ✓ **Tuples:** An ordered collection of elements of different data types, typically fixed in size.

❖ **Based on memory allocation, Linear Data Structures are classified into two types:**

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled

Example: Array

2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code.

Example: Linked Lists, Stacks, and Queues

❖ **Non-linear data structure:**

- ➢ Non-linear data structures allow elements to be connected in more complex ways, creating relationships other than a simple linear sequence.
- ➢ These structures are often used to represent hierarchical or interconnected relationships among data elements.

Example:

- ✓ Trees, Graphs, Hash Tables (Hash Maps), Graphical Data Structures, Disjoint-set Union (Union-Find), Linked Hash Maps

❖ **Trees:**

- Binary Trees, Binary Search Trees, AVL Trees, B-Trees, Red-Black Trees, Heap (Binary Heap, Min Heap, Max Heap), Trie (Prefix Tree)

❖ **Graphs:**

- Directed Graphs (Digraphs), Undirected Graphs, Weighted Graphs, Connected Graphs, Acyclic Graphs (DAG - Directed Acyclic Graphs)

❖ **Graphical Data Structures**

- Points
- Lines
- Polygons
- Shapes

❖ **Advantage of data structure:**

1) **Efficient Data Organization:** Data structures provide systematic ways to organize and store data.
2) **Optimized Algorithm Design:** Many algorithms rely on specific data structures to perform tasks effectively. leading to faster and more reliable solutions.
3) **Memory Management:** Data structures help manage memory effectively. They allow for dynamic memory allocation.

4) **Fast Data Retrieval and Search:** Data structures enable efficient searching for specific elements, improving the performance of search operations.
5) **Resource Utilization:** Data structures efficiently use resources like memory and processing power, making programs more efficient and cost-effective.
6) **Problem-Solving:** Data structures help break down complex problems into simpler
7) **Flexibility and Adaptability:** Having a variety of data structures allows developers to choose the best one for their particular application.
8) **Application Development:** Data structures are foundational for building various applications, from simple to complex.
9) **Data Storage and Databases:** Data structures form the foundation of databases, enabling the storage, organization, and retrieval of data.
10) **Code Reusability:** Data structures facilitate the creation of reusable code modules.

Note:

- Data structures play a crucial role in computer science and software development, providing necessary tools to efficiently manage data, optimize algorithms, and solve problems.

❖ Simple Explanation:

1. Efficient data organization, storage, and manipulation.
2. Optimized algorithm design for faster solutions.
3. Effective memory management and allocation.
4. Fast data retrieval and search operations.
5. Resource utilization, saving memory and processing power.
6. Systematic approach to problem-solving.
7. Flexibility with various data structure options.
8. Foundation for application development in diverse domains.
9. Support for databases and data storage systems.
10. Code reusability for better software design.

DATA STRUCTURE LAB IN PYTHON

#Primitive data types

Program

```
a=6
print(type(a))
b=-6
print(type(b))
c=3.4
print(type(c))
d=True
print(type(d))
e="A"
print(type(e))
ab=None
print(type(ab))
```

Output

```
<class 'int'>
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
<class 'NoneType'>
```

 # Primitive (single value) & non-Primitive

Program

```
a, b =6, 3.4
c=30,9,23
print(type(a))
print(type(b))
print(type(c))
```

Output

```
➔ <class 'int'>
➔ <class 'float'>
➔ <class 'tuple'>
```

 # Non- Primitive data types (Multiple value)

Program

```
a=[3,3.4, True, 'skills', None,23]
print(a)
print(type(a), len(a))
b=3,3.4, True, 'skills', None,30
print(b)
print(type(b),len(b))
```

Output

```
[3,3.4, True, 'skills',None,23]
<class 'list'> 6
(3,3.4, True,'skills',None, 30)
<class 'tuple'> 6
```

 # tuple size is fixed but list size is not fixed

Program

```
a=3,3.4, True, 'skills', None,3
print(type(a),len(a),a)
b=[3,3.4, True, 'skills', None,3]
b.append(393)
b.append('center')
print(b,type(b),len(b))
a.append(393)
print(type(a),len(a))
```

Output

```
<class 'tuple'> 6 (3, 3.4, True,
'skills', None, 3)
[3, 3.4, True, 'skills', None, 3,
393, 'center'] <class 'list'> 8
AttributeError: 'tupleno no 'append'
```

 # list & tuple maintain ordered but not set

Program

```
s={1,2.3,4.6,39, True, False,'skills','skills'}
print(s,type(s),len(s))
```

Output

```
{False, 1, 2.3, 4.6, 39, 'skills'}
<class 'set'> 6
```

DATA STRUCTURE AND ALGORITHM

❖ **Algorithm:**

- An algorithm is a step-by-step procedure to solve a specific problem or perform a task effectively. (एल्गोरिदम किसी विशिष्ट समस्या को हल करने या किसी कार्य को प्रभावी ढंग से करने के लिए चरण-दर-चरण प्रक्रिया है।)
- An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer.
- It is not the complete program or code; it is just a solution (logic) of a problem.

❖ **Characteristics of an Algorithm:**

- **Input:** An algorithm has some input values. like 0
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** Algorithm should be unambiguous, means instructions should clear & simple.
- **Finite:** Algorithm should have finite. Means algorithm should limited number of instructions.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm.
- **Language independent:** An algorithm must be language-independent

❖ **Dataflow of an Algorithm:**

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem
- **Algorithm:** An algorithm will be designed for a problem which is a step-by-step procedure.
- **Input:** The desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

❖ **Algorithm can develop for:**

- **Sort:** Algorithm developed for sorting the items in a certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

❖ **Types of algorithms:**

1. Sorting Algorithms
2. Searching Algorithms
3. Recursive Algorithms
4. Greedy Algorithms
5. Dynamic Programming Algorithms
6. Backtracking Algorithms
7. Divide and Conquer Algorithms
8. Randomized Algorithms
9. Pattern Matching Algorithms
10. Graph Algorithms
11. Numerical Algorithms

Note: We will learn in details this all things in DAA (Design and Analysis of Algorithms) Subject.

Example:

Example-1: Problem: Write the algorithm for add the two numbers.

- **Algorithm:**

Step 1: Start
Step 2: Declare three variables a, b, and sum.
Step 3: Enter the values of a and b.
Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum=a+b.
Step 5: Print sum
Step 6: Stop

- **Program in python**

```
num1 = eval(input("Enter the first number: "))
num2 = eval(input("Enter the second number: "))
sum = num1 + num2
print("Sum:", sum)
```

- **output**

```
Enter the first number: 25
Enter the second number: 30
Sum: 55
```

#Addition of two number in c

```
#include <stdio.h>
int main() {
   int num1, num2, sum;
   printf("Enter the first number: ");
   scanf("%d", &num1);
   printf("Enter the second number: ");
   scanf("%d", &num2);
   sum = num1 + num2;
   printf("Sum: %d\n", sum);
   return 0;}
```

Output:

```
Enter the first number: 25
Enter the second number: 30
Sum: 55
```

Example-2 Problem: Find the sum of even numbers from a given positive number.

- **Algorithm:**

1. Start
2. Read the value of 'n'.
3. Set a variable 'sum' to 0 to store the sum of even numbers.
4. Set a variable 'i' read the i value one by one from given range.
5. While 'i' is less than or equal to 'n', do steps 6-7.
6. Add 'i' to 'sum'.
7. Increment 'i' by 2 to move to the next even number. End the loop.
8. Display the value of 'sum' as the result. 9. End

- **Program:**

```
n = eval(input("Enter a positive number: "))
sum = 0
if n > 0:
   for i in range(2, n + 1, 2):
      sum=sum+i
   print("Sum of even numbers=", sum)
else:
   print("Please enter a positive number.")
```

- **output**

```
Enter a positive number: 10
Sum of even numbers= 30
```

Example-2 swape the two number

- **Algorithm:**

  Step-1. Start
  Step-2. Input the first number (num1)
  Step-3. Input the second number (num2)
  4. Display "Before swapping: print (num1, num2)
  5. Perform the swap:
    a. temp variable
    b. temp=num1
    c. num1=num2 and num2=temp
  6. Display "After swapping: print (“After swapping”, num1, num2)
  7. End

- **Program in python method-1:**

```
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
print("Before swapping: ",num1, num2)
temp=num1
num1=num2
num2=temp
print("after swapping: ", num1,num2)
```

- **output**

```
Enter the first number:  25
Enter the second number:  50
Before swapping:  25 50
after swapping:  50 25
```

- **Program in python method-2:**

```
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
print("Before swapping: ",num1, num2)
c=num1+num2
num1=c-num1
num2=c-num1
print("After swapping: ", num1,num2)
```

- **output**

```
Enter the first number:  25
Enter the second number:  50
Before swapping:  25 50
After swapping:  50 25
```

- **Program in python method-3:**

```
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
print("Before swapping: ",num1, num2)
c=num1*num2
num1=c//num1
num2=c//num1
print("After swapping: ", num1,num2)
```

- **Output**

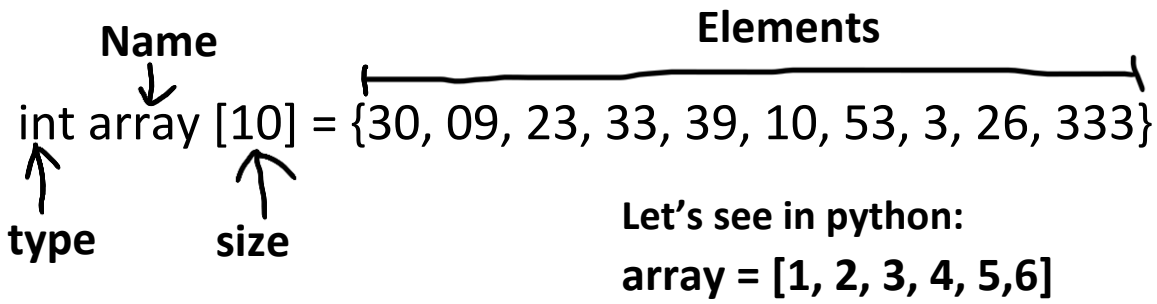
```
Enter the first number:  10
Enter the second number:  20
Before swapping:  10 20
After swapping:  20 10
```

ARRAY

- Arrays are defined as a collection of similar data items stored at contiguous memory locations.
  - **Contiguous:** means elements stored adjacently in memory without any gaps

❖ **Representation of an array:**

- We can represent an array in various ways in different programming languages.



- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

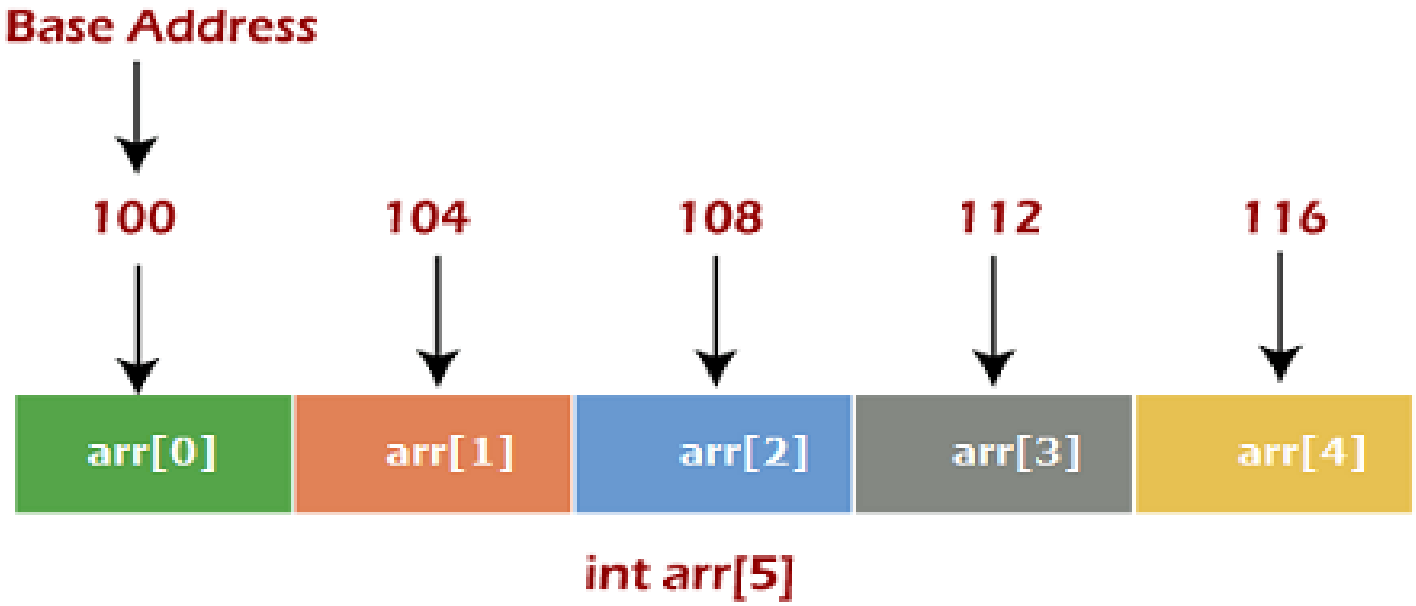
❖ **Why are arrays required?**

➢ **Arrays are useful because -**

- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.
- Arrays are good for storing multiple values in a single variable

Memory allocation of an array:

- 0 (zero-based indexing): The first element of the array will be arr[0].



❖ Basic operations:

1. **Traversal -** This operation is used to print the elements of the array.
2. **Insertion -** It is used to add an element at a particular index.
3. **Deletion -** It is used to delete an element from a particular index.
4. **Search -** It is used to search an element using the given index or by the value.
5. **Update -** It updates an element at a particular index.

❖ Basic operations in Python:

1. **Creating an array:**
   l=[1,2,3,6]
2. **Show elements:**
   `print(l) # [1, 2, 3, 6]`
3. **Access elements:**
   ```
   l=[1,2,3,'skills']
   print(len(l)) #4
   print(l[3]) #skills
   print(l[0]) #1
   ```
4. **Adding elements:**
   ```
   l=[1,2,3,'skills']
   l.append('center')
   l.insert(2,33)
   print(l) # [1, 2, 33, 3, 'skills', 'center']
   ```
5. **Length of the array:**
   l=[1,2,3,4,5,6]
   ```
   print(len(l)) # Output: 6
   ```
6. **Removing elements (by value):**
   ```
   l=[1,2,3,4,5,6]
   print("lenght=",len(l)) # lenght= 6
   print(l) # [1, 2, 3, 4, 5, 6]
   l.pop()
   l.pop(3)
   print(l) # [1, 2, 3, 5]
   del l[2]
   print(l) # [1, 2, 5]
   l.remove(1)
   print(l) #[2, 5]
   ```
7. **update the elements**
   ```
   l=[1,2,3]
   l[2]=9
   print(l) # [1, 2, 9]
   l[0]=339
   print(l) # [339, 2, 9]
   ```
8. **Concatenation:**
   ```
   l1=[1,2,3]
   l2=[4,5,6]
   print(l1+l2) # [1, 2, 3, 4, 5, 6]
   ```

❖ Basic operations in C:

Traversal operation

```
#include <stdio.h>
void main() {
  int Arr[5] = {18, 30, 15, 70, 12};
int i;
  printf("Elements of the array are:\n");
  for(i = 0; i<5; i++) {
    printf("Arr[%d] = %d,  ", i, Arr[i]);
  } }
```

Output:

Elements of the array are:
Arr[0] = 10, Arr[1] = 30, Arr[2] = 15,
Arr[3] = 70, Arr[4] = 12, Arr[5] = 24,

Deletion operation

```
#include <stdio.h>
void main() {
  int arr[] = {18, 30, 15, 70, 12};
  int k = 30, n = 5;
  int i, j;
  printf("Given array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("arr[%d] = %d,  ", i, arr[i]);}
  j = k;
  while( j < n) {
    arr[j-1] = arr[j];
    j = j + 1; }
  n = n -1;
  printf("\nElements of array after
deletion:\n");
  for(i = 0; i<n; i++) {
    printf("arr[%d] = %d,  ", i, arr[i]);}}
```

Output:

Given array elements are :
arr[0] = 18, arr[1] = 30, arr[2] = 15, arr[3] = 70,
arr[4] = 12,
Elements of array after deletion:
arr[0] = 18, arr[1] = 30, arr[2] = 15, arr[3] = 70,

Insertion operation

```
#include <stdio.h>
int main()
{ int arr[24] = { 10, 33, 5, 20, 124 };
   int i, x, pos, n = 5;
   printf("Array elements before insertion\n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
   x = 339; // element to be inserted
   pos = 4;
   n++;
   for (i = n-1; i >= pos; i--)
      arr[i] = arr[i - 1];
   arr[pos - 1] = x;
   printf("Array elements after insertion\n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
   return 0; }
```

Output:

Array elements before insertion
10 33 5 20 124
Array elements after insertion
10 33 5 339 20 124

Search operation

```
#include <stdio.h>
void main() {
  int arr[5] = {18, 30, 15, 70, 12};
  int item = 70, i, j=0 ;
  printf("Given array elements are :\n");
  for(i = 0; i<5; i++) {
    printf("arr[%d] = %d,  ", i, arr[i]);}
  printf("\nElement to be searched = %d", item);
  while( j < 5){
    if( arr[j] == item ) {
      break;
    }
    j = j + 1; }
printf("\nElement %d is found at %d position", item, j+1);}
```

Output:

Given array elements are :
arr[0] = 18, arr[1] = 30, arr[2] = 15, arr[3] = 70, arr[4] = 12,
Element to be searched = 70
Element 70 is found at 4 position

❖ 2D Array:

- 2D array can be defined as an array of arrays.
- 2D array is organized as matrices which can be represented as collection of rows and columns.

❖ How to declare 2D Array

➢ **Syntax:**

- **int arr[max_rows] [max_columns];**

	0	1	2	3		n-1
0	A [0][0]	A [0][1]	A [0][2]	A [0][3]	……	A [0] [n-1]
1	A [1][0]	A [1][1]	A [1][2]	A [1][3]	……	A [1] [n-1]
2	A [2][0]	A [2][1]	A [2][2]	A [2][3]	……	A [2] [n-1]
3	A [3][0]	A [3][1]	A [3][2]	A [3][3]	……	A [3] [n-1]
	. . .	. . .	. . .	. . .	……	. . .
n-1	A[n-1][0]	A[n-1][1]	A[n-1][2]	A[n-1][3]		A [n-1] [n-1]

❖ How do we access data in a 2D array?

Syntax:

- int x = a[i][j];
- where i and j is the row and column number of the cell respectively

How to declare nested list in python

```
n=[[10,20,30],[40,50,60],[70,80,90]]
print(n)
#Elements by row wise
print("elements by row wise")
for r in n:
   print(r)
#print elemets by matrix style
for i in range(len(n)):
   for j in range(len(n[i])):
      print(n[i][j],end=" ")
  print()
```

Output

```
[[10, 20, 30], [40, 50, 60], [70, 80, 90]]
elements by row wise
[10, 20, 30]
[40, 50, 60]
[70, 80, 90]
10 20 30
40 50 60
70 80 90
```

❖ Basic operation in 2D Array:

Python:

1. Creating a 2D array:

```
matrix = [
 [1, 2, 3],
[4, 5, 6],
 [7, 8, 9]]
```

2. Accessing elements:

```
element = matrix [1][2]
print(element) # Output: 6
```

3. Modifying elements:

```
matrix [2][1] = 10
print(matrix)
```

4. Length of the 2D array (number of rows):

```
rows = len(matrix)
print(rows)  # Output: 3
```

5. Length of a row (number of columns):

```
cols = len(matrix[0])
print(rows) # Output: 3
```

6. Adding a new row:

```
new_row = [11, 12, 13]
matrix.append(new_row)
print(matrix) #[[1, 2, 3], [4, 5, 6], [7, 10, 9], [11, 12, 13]]
rows = len(matrix)
print(rows) #4
cols = len(matrix[0])
print(cols) #3
```

7. Adding a new column:

```
new_column = 14
for row in matrix:
        row.append(new_column)
print(matrix) #[[1, 2, 3, 14], [4, 5, 6, 14], [7, 10, 9, 14], [11, 12, 13, 14]]
rows = len(matrix)
print(rows) #4
```

8. Accessing a row:

```
#print matrix
for i in range(len(matrix)):
   for j in range(len(matrix[i])):
      print(matrix[i][j],end=" ")
   print()
row_index = 1
row = matrix[row_index]
print(row) # [4, 5, 6, 14]
row_index = 0
row = matrix[row_index]
print(row) # [1, 2, 3, 14]
```

```
1 2 3 14
4 5 6 14
7 10 9 14
```

9. Accessing a column:

```
col_index = 3
column = [row[col_index] for row in matrix]
print(column) #[14, 14, 14, 14]
col_index = 1
column = [row[col_index] for row in matrix]
print(column) #[2, 5, 10, 12]
```

10. Iterating through the 2D array:

```
for row in matrix:
   for element in row:
      print(element)
```

Output: 1,2,3,14,4,5,6,14,7,10,9,14,11,12,13,14

11. Printing the 2D array:

```
for row in matrix:
        print(row)
```

Output:

[4, 5, 6, 14]
[7, 10, 9, 14]
[11, 12, 13, 14]

12. Checking if an element exists in the 2D array:

```
if 6 in matrix [1]:
print ("Element 6 exists in row 1.")   # Output: Element 6 exists in row 1.
```

❖ Some Pattern Example 2D-array

```
n=6
for i in range(1,n+1):
    print(" * "*n)
```

Output

```
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

```
n=6
for i in range(1,n+1):
   for j in range(1,n+1):
      print(i,end=" ")
   print()
```

Output

```
1 1 1 1 1 1
2 2 2 2 2 2
3 3 3 3 3 3
4 4 4 4 4 4
5 5 5 5 5 5
6 6 6 6 6 6
```

```
n=6
for i in range(1,n+1):
   for j in range(1,n+1):
      print(chr(64+i), end=" ")
   print()
```

Output

```
A A A A A A
B B B B B B
C C C C C C
D D D D D D
E E E E E E
F F F F F F
```

```
n=6
for i in range(1,n+1):
   for j in range(1,n+1):
      print(chr(96+i), end=" ")
   print()
```

Output

```
a a a a a a
b b b b b b
c c c c c c
d d d d d d
e e e e e e
f f f f f f
```

```
n=6
for i in range(1,n+1):
   for j in range(1,i+1):
      print("*",end=" ")
   print()
```

Output

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

```
n=6
for i in range(1,n+1):
   for j in range(1,i+1):
      print(chr(64+i),end=" ")
   print()
```

Output

```
A
B B
C C C
D D D D
E E E E E
F F F F F F
```

❖ Advantages of Array:

- Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

❖ Disadvantages of Array:

- Array is homogenous. It means that the elements with similar data type can be stored in it.
- In array, there is static memory allocation that is size of an array cannot be altered.
- There will be wastage of memory if we store a smaller number of elements than declared size.

❖ Application of Array:

➢ Arrays have a wide range of applications in computer science and real-world scenarios due to their simplicity and efficiency for storing and accessing elements.

1. **Storing Data:** Arrays are used to store collections of data, such as lists of numbers, strings, or objects. They provide a convenient way to organize and access data elements using indexes.
2. **Matrices and Multidimensional Arrays:** Arrays can be used to represent matrices and multidimensional data structures, making them useful in graphics, image processing, and scientific computations.
3. **Dynamic Programming:** Arrays are often used in dynamic programming algorithms to store intermediate results, making it easier to solve complex problems with optimal substructure.
4. **Sorting and Searching:** Arrays are commonly used for sorting and searching algorithms like binary search, bubble sort, quicksort, etc.
5. **Hash Tables:** Arrays are the underlying data structure for hash tables, which provide efficient data retrieval based on key-value pairs.
6. **Buffers and Caches:** Arrays are used to implement buffers and caches to temporarily store data and improve data access efficiency.
7. **Graph Algorithms:** Arrays are used to store graph adjacency lists and other representations of graphs used in graph algorithms like breadth-first search (BFS) and depth-first search (DFS).
8. **Memory Management:** Arrays are used in memory management to allocate and deallocate memory blocks.
9. **Simulation:** Arrays are used in simulations to represent states and track changes over time.
10. **Game Development:** Arrays are used in game development to store game maps, character inventories, and various game-related data.
11. **Data Structures:** Arrays are the basis for many other data structures like stacks, queues, and heaps.
12. **Digital Signal Processing:** Arrays are used in digital signal processing to represent and manipulate signals.

❖ Real Time Application of list in number game program.

- **Program:**

```
from random import *
l=[1,2,3,4,5,6]
play="yes"
while play=="yes":
   guess=int(input("Enter your guess for dice roll:"))
   if 1<=guess<=6:
      comp=choice(l)
      print("Computer's dice roll:",comp)
      if guess==comp:
         print("You win!!!")
      else:
         print("You lost..")
   else:
      print("Invalid choice")
   play=input("Do you want play again(yes/no)? ")
```

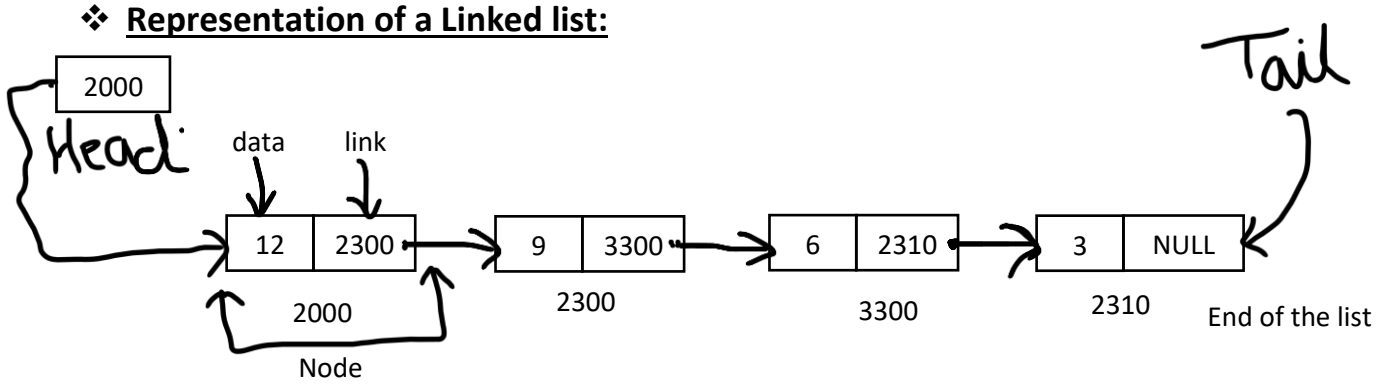
- **Output:**

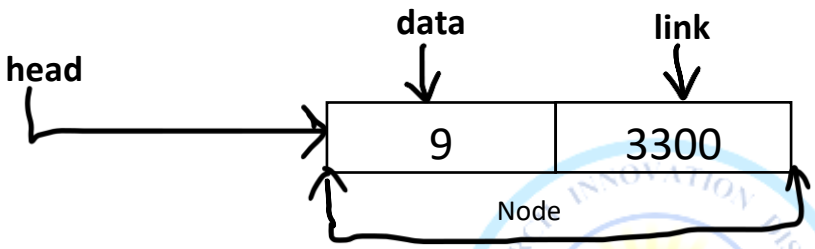
```
Enter your guess for dice roll:3
Computer's dice roll: 4
You lost..
Do you want play again(yes/no)? yes
Enter your guess for dice roll:6
Computer's dice roll: 2
You lost..
Do you want play again(yes/no)? yes
Enter your guess for dice roll:2
Computer's dice roll: 2
You win!!!
Do you want play again(yes/no)? yes
Enter your guess for dice roll:10
Invalid choice
```

LINKED LIST

- A linked list is a non-primitive, linear data structure used to store a collection of elements.
- linked list represents elements as nodes, where each node contains both the element's value and a reference (or pointer) to the next node in the sequence.

❖ Representation of a Linked list:





❖ Why Linked list is useful?

➢ **Linked list is useful because: -**

- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand.

❖ Advantages of Linked list:

1. **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
2. **Insertion and deletion -** insertion, and deletion in linked list is easier. Because elements in the linked list are stored at a random location. So that in linked list, we just have to update the address of the pointer of the node.
3. **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
4. **Implementation -** We can implement both stacks and queues using linked list.

❖ Disadvantages of Linked list:

1. **Memory usage -** node occupies more memory than array. Each node of linked list occupies two types of variables, i.e., one is a simple variable, and another one is pointer variable.
2. **Traversal -** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly
3. **Reverse traversing -** Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

❖ Application of linked list:

- Linked lists have a wide range of applications in computer science and real-world scenarios.

1. **Dynamic Data Structures:** Linked lists are used as a basic building block to create other dynamic data structures like stacks, queues, and graphs.
2. **Memory Allocation:** Linked lists are used in memory management systems to manage memory allocation and deallocation efficiently.
3. **File Systems:** Linked lists are used in file systems to maintain the hierarchical structure of directories and files.
4. **Music and Video Playlists:** Linked lists are used to create playlists in music and video players, allowing users to add, remove, and shuffle tracks.
5. **Browser History:** Linked lists are used in web browsers to maintain the browsing history, enabling users to navigate backward and forward through visited pages.
6. **Task Scheduling:** Linked lists are used in task scheduling algorithms to manage the order of tasks to be executed.
7. **Polynomial Addition & Multiplication:** Linked lists are used to represent polynomials in mathematical computations, where the coefficients are stored in nodes of the linked list.
8. **Symbol Table:** Linked lists are used in symbol table implementations for compilers and interpreters to manage variables and their values.
9. **Hash Table Collision Handling:** Linked lists are used in hash tables to handle collisions when two keys hash to the same index.
10. **Undo/Redo Functionality:** Linked lists can be used to implement undo/redo functionality in applications, allowing users to revert changes or redo previous actions.
11. **Representing Sparse Data:** Linked lists are suitable for representing sparse data structures where most of the elements are zero or not used.
12. **Graph Algorithms:** Linked lists are used to represent adjacency lists in graph data structures, enabling efficient graph algorithms like BFS and DFS.

❖ Operations performed on Linked list

1. **Insertion -** This operation is performed to add an element into the list.
2. **Deletion -** It is performed to delete an operation from the list.
3. **Display -** It is performed to display the elements of the list.
4. **Search -** It is performed to search an element from the list using the given key.

❖ Types of link list:

- There are several types of linked lists

1) Singly Linked List
2) Doubly Linked List
3) Circular Linked List
4) Singly Linked List with a Tail Pointer
5) Skip List
6) Self-Organizing List
7) Unrolled Linked List
8) XOR Linked List
9) Sparse Linked List

❖ Singly Linked List:

- Each node has a data element and a reference (pointer) to the next node in the list.
- Traversing can be done only in one direction (forward).
- Efficient for insertion and deletion at the beginning and end but not efficient for accessing elements in the middle.

Example in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def add_node(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
# Creating a singly linked list: 1 -> 2 -> 3 -> None
sll = SinglyLinkedList()
sll.add_node(1)
sll.add_node(2)
sll.add_node(3)
sll.display()
```

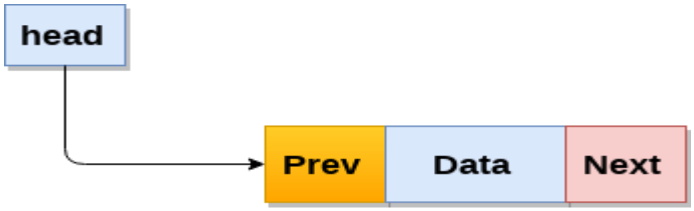
Output:

```
1 -> 2 -> 3 -> None
```

❖ Doubly Linked List:

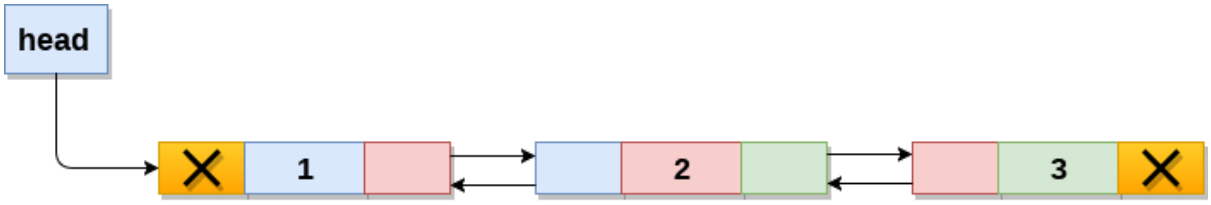
- Each node has a data element, a reference to the next node, and a reference to the previous node in the list.
- Traversing can be done in both forward and backward directions.
- Efficient for insertion and deletion at the beginning and end, and also for accessing elements in the middle.

➢ **A sample node in a doubly linked list**



Node

➢ **A doubly linked list containing three nodes having numbers from 1 to 3 in their data part**



Doubly Linked List

❖ Operations on doubly linked list:

- **Insertion at beginning**: Adding the node into the linked list at beginning.
- **Insertion at end**: Adding the node into the linked list to the end.
- **Insertion after specified node**: Adding the node into the linked list after the specified node.
- **Deletion at beginning**: Removing the node from beginning of the list
- **Deletion at the end**: Removing the node from end of the list.
- **Deletion of the node having given data**: Removing the node which is present just after the node containing the given data.
- **Searching**: Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
- **Traversing**: Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc

Example in Python:

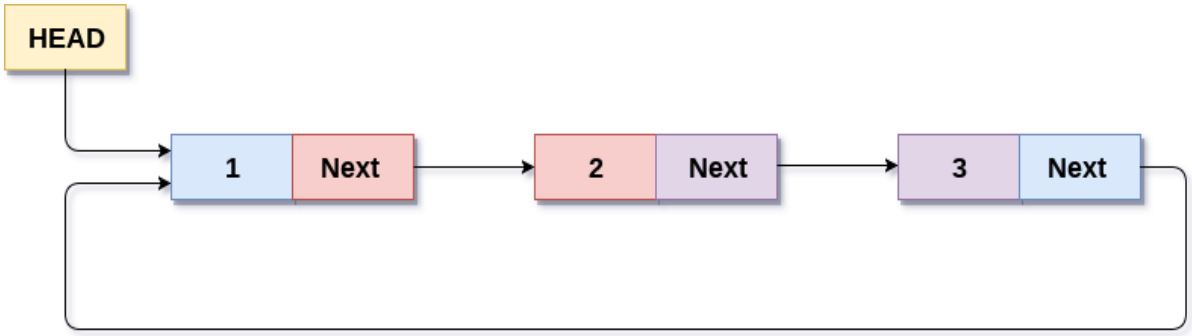
```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
    def add_node(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" <-> ")
            current = current.next
        print("None")
# Creating a doubly linked list: 1 <-> 2 <-> 3
dll = DoublyLinkedList()
dll.add_node(1)
dll.add_node(2)
dll.add_node(3)
dll.display()
```

Output:

```
1 <-> 2 <-> 3 <-> None
```

❖ Circular Linked List:

- Similar to a singly or doubly linked list, but the last node's reference points back to the first node, forming a circular structure.
- Can be useful in certain scenarios where you need continuous looping through the elements.



Circular Singly Linked List

Operations on Circular Singly linked list:

- **Insertion at beginning:** Adding a node into circular singly linked list at the beginning.
- **Insertion at the end:** Adding a node into circular singly linked list at the end.
- **Deletion at beginning:** Removing the node from circular singly linked list at the beginning.
- **Deletion at the end:** Removing the node from circular singly linked list at the end.
- **Searching:** Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
- **Traversing:** Visiting each element of the list at least once in order to perform some specific operation.

Example in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class CircularLinkedList:
    def __init__(self):
        self.head = None
    def add_node(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head
        else:
            current = self.head
            while current.next != self.head:
                current = current.next
            current.next = new_node
            new_node.next = self.head

    def display(self):
```

```
        current = self.head
        while True:
            print(current.data, end=" -> ")
            current = current.next
            if current == self.head:
                break
        print("Head")
# Creating a circular linked list: 1 -> 2 -> 3 -> Head
cll = CircularLinkedList()
cll.add_node(1)
cll.add_node(2)
cll.add_node(3)
cll.display()
```

Output:

1 -> 2 -> 3 -> Head

❖ Circular Doubly Linked List:

- Circular doubly linked list is a more complexed type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of list contains the address of the first node of the list. The first node of the list also contains address of the last node in its previous pointer.

Circular Doubly Linked List

Example in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            self.head.prev = self.head
            self.head.next = self.head
        else:
            last_node = self.head.prev
            last_node.next = new_node
```

```
            new_node.prev = last_node
            new_node.next = self.head
            self.head.prev = new_node
    def print_list(self):
        if not self.head:
            print("List is empty")
            return
        current = self.head
        print("Circular Doubly Linked List: ", end="")
        while True:
            print(current.data, end=" ")
            current = current.next
            if current == self.head:
                break
        print()
# Example program to demonstrate the usage of Circular Doubly Linked List
if __name__ == "__main__":
    cdl_list = CircularDoublyLinkedList()
    # Insert elements into the list
    cdl_list.insert_at_end(1)
    cdl_list.insert_at_end(2)
    cdl_list.insert_at_end(3)
    cdl_list.insert_at_end(4)
    cdl_list.insert_at_end(5)
    # Print the list
    cdl_list.print_list()
```

Output:

Circular Doubly Linked List: 1 2 3 4 5

❖ Skip List:

- A linked list with multiple layers of nodes, allowing for faster searching by "skipping" some nodes during traversal.
- Typically used in data structures to improve the efficiency of search operations.
- The linked list automatically reorganizes itself based on the frequency of node accesses to improve performance.
- When an element is accessed, it is moved to the front of the list to make subsequent access faster.

❖ Skip List Basic Operations

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.
- **Search Operation:** The search operation is used to search a particular node in a skip list.

Example in python:

- Skip list example code is quite complex and space-consuming.
- Here's a simple output to demonstrate insertion in a Skip List.

```
import random
class Node:
    def __init__(self, data, forward=None):
        self.data = data
        self.forward = [forward] * (random.randint(1, 4) if forward is not None else 1)
class SkipList:
    def __init__(self):
        self.head = Node(None)
    def find(self, target):
        current = self.head
        for level in reversed(range(len(current.forward))):
            while current.forward[level] is not None and current.forward[level].data < target:
                current = current.forward[level]
        return current.forward[0] if current.forward[0] is not None and current.forward[0].data ==
target else None
    def insert(self, data):
        current = self.head
        update = [None] * len(current.forward)
        for level in reversed(range(len(current.forward))):
            while current.forward[level] is not None and current.forward[level].data < data:
                current = current.forward[level]
            update[level] = current
        level = random.randint(1, len(update) + 1)
        new_node = Node(data)
        for i in range(level):
            new_node.forward.append(update[i].forward[i])
            update[i].forward[i] = new_node
    def display(self):
        current = self.head
        while current.forward[0] is not None:
            print(current.forward[0].data, end=" -> ")
            current = current.forward[0]
        print("None")
# Creating a skip list: (e.g., 1 -> 2 -> 3)
sl = SkipList()
sl.insert(1)
sl.insert(2)
sl.insert(3)
sl.display()
```

Output:

```
1 -> 2 -> 3 -> None
```

.

❖ Unrolled Linked List:

- Linked list in which each node contains multiple elements (usually an array or a block of data) rather than a single data element.
- Can be more memory-efficient and reduce overhead associated with individual node pointers.

Example in Python:

```
class Node:
    def __init__(self, capacity):
        self.capacity = capacity
        self.data = [None] * capacity
        self.next = None
class UnrolledLinkedList:
    def __init__(self, node_capacity=3):
        self.head = Node(node_capacity)
    def add_node(self, data):
        current = self.head
        while current.next:
            current = current.next
        if None in current.data:
            idx = current.data.index(None)
            current.data[idx] = data
        else:
            new_node = Node(current.capacity)
            new_node.data[0] = data
            current.next = new_node
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
# Creating an un
rolled linked list: [1, 2, 3] -> [4]
ull = UnrolledLinkedList()
ull.add_node(1)
ull.add_node(2)
ull.add_node(3)
ull.add_node(4)
ull.display()
```

Output:

```
[1, 2, 3] -> [4, None, None] -> None
```

❖ XOR Linked List:

- Each node stores the bitwise XOR of the addresses of the previous and next nodes instead of explicit references.
- Allows traversing list in both directions using only a single pointer, which can save memory.

Example in python:

- Note: XOR linked list implementation requires low-level pointer manipulation
- which is not directly supported in Python. Below is a simplified version using regular references.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.xor_pointer = None
class XORLinkedList:
    def __init__(self):
        self.head = None
    def add_node(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            prev = None
            current = self.head
            while current.xor_pointer:
                prev, current = current, self.get_pointer(prev, current.xor_pointer)
            current.xor_pointer = self.get_pointer(prev, new_node)
    def get_pointer(self, a, b):
        return a if b == None else b
    def display(self):
        prev = None
        current = self.head
        while current:
            print(current.data, end=" -> ")
            next_node = self.get_pointer(prev, current.xor_pointer)
            prev, current = current, next_node
        print("None")
# Creating an XOR linked list: 1 -> 2 -> 3
xll = XORLinkedList()
xll.add_node(1)
xll.add_node(2)
xll.add_node(3)
xll.display()
```

Output:

```
1    -> 2 -> 3 -> None
```

❖ Self-adjusting Linked List:

- A Self-Adjusting Linked List (SAL-list) is a data structure that automatically reorders its elements based on their access patterns. When an element is accessed, it moves to the front of the list, making it faster to access the same element again in the future.

Example in python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SelfAdjustingLinkedList:
    def __init__(self):
        self.head = None
    def add_node(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
    def find(self, target):
        prev = None
        current = self.head
        while current:
            if current.data == target:
                if prev:
                    prev.next = current.next
                    current.next = self.head
                    self.head = current
                return current
            prev = current
            current = current.next
        return None
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
# Creating a self-adjusting linked list: 1 -> 2 -> 3
sall = SelfAdjustingLinkedList()
sall.add_node(1)
sall.add_node(2)
sall.add_node(3)
sall.display()
```

Output:

```
1 -> 2 -> 3 -> None
```

STACK

- A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.
- Stack has one end; it contains only one pointer top pointer pointing to the topmost element of the stack.
- stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

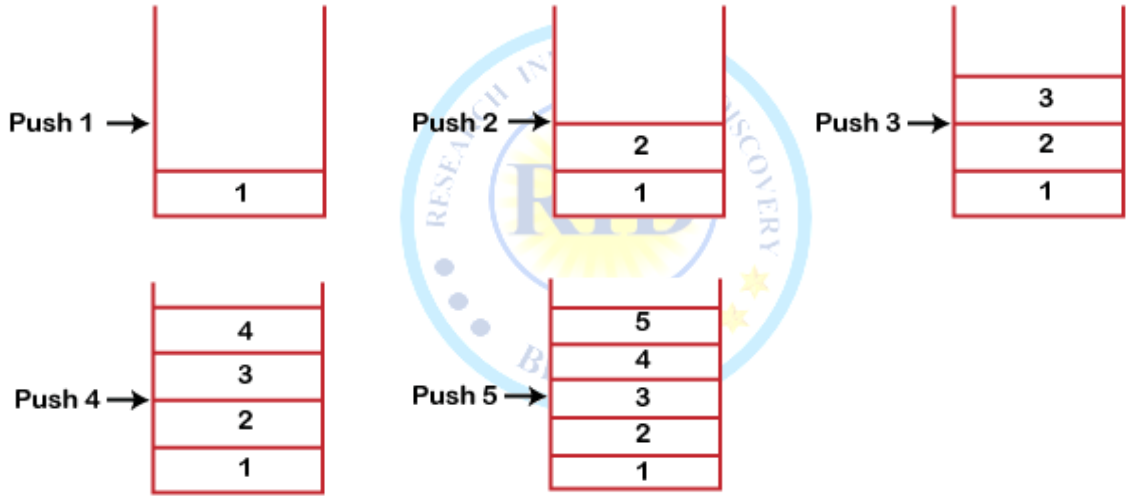
❖ **Application of stack:** Browser history navigation (back and forward buttons).

❖ **Some key points related to stack:**

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

❖ **Working of Stack:**

- Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.



❖ **Standard Stack Operations:**

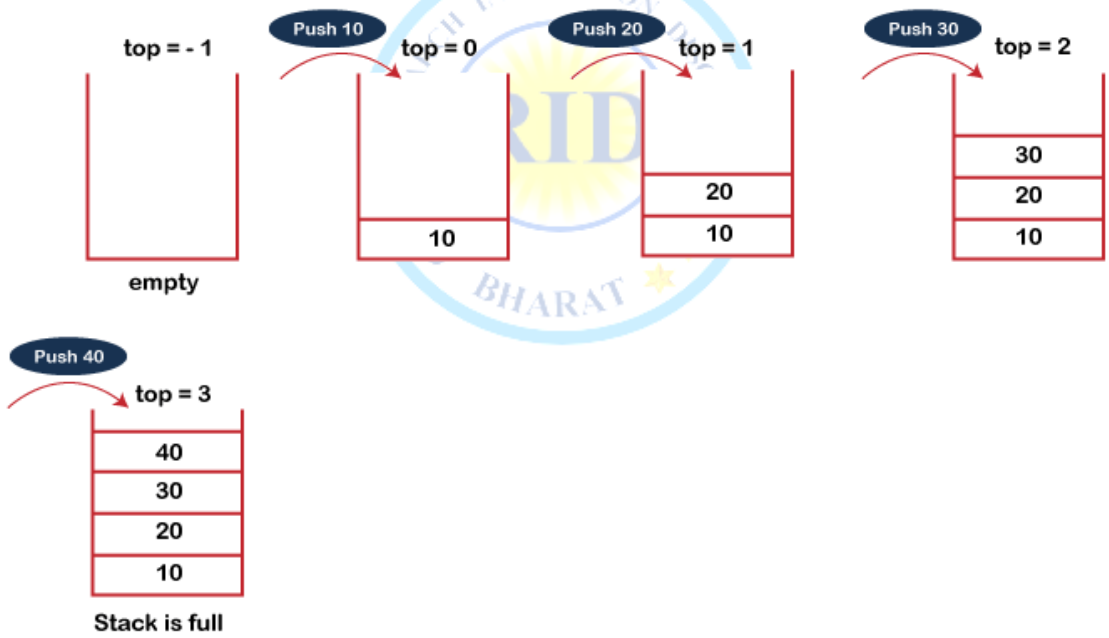
1. **push ():** When we insert an element in a stack then the operation is known as a push.
2. **Pop ():** When we delete an element from the stack, the operation is known as a pop.
3. **isEmpty ():** It determines whether the stack is empty or not.
4. **isFull ():** It determines whether the stack is full or not.'
5. **Peek ():** It returns the element at the given position.
6. **Count ():** It returns the total number of elements available in a stack.
7. **Change ():** It changes the element at the given position.
8. **Display ():** It prints all the elements available in the stack.

❖ Application of Stack:

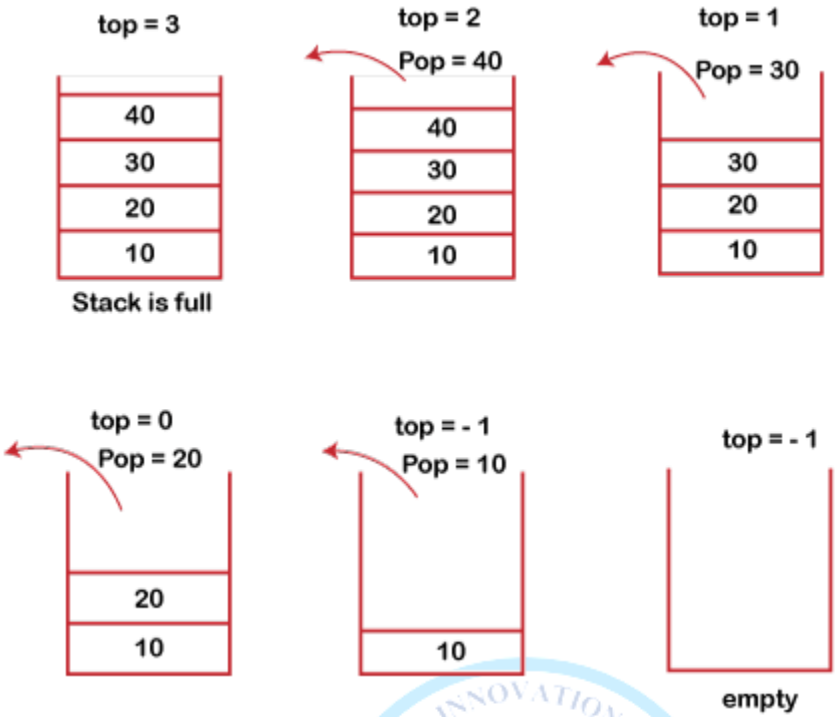
1. **Function Call Stack:** Stacks are used to manage function calls and their local variables in programming languages
2. **Expression Evaluation:** Stacks are used to evaluate expressions, both arithmetic and logical.

3. **Parsing and Syntax Checking:** Stacks are used in parsing algorithms like Dijkstra's Shunting Yard algorithm to convert infix expressions to postfix notation, which simplifies expression evaluation.
4. **Backtracking Algorithms:** In algorithms like depth-first search (DFS) and recursive backtracking, a stack is used to keep track of the nodes or states to be explored.
5. **Undo/Redo Functionality:** Stacks can be used to implement undo and redo functionalities in applications where actions need to be reversed or redone in a specific order.
6. **Browser History:** Stacks can be used to maintain the history of visited web pages in a web browser, allowing users to navigate backward (back button) and forward (forward button).
7. **Memory Management:** Stacks are used in memory management for managing the call stack and local variables in a program.
8. **Parentheses Matching:** Stacks can be used to check the validity of parentheses, braces, and brackets in expressions or code.
9. **Postfix Expression Evaluation:** Stacks can be used to evaluate postfix expressions efficiently.
10. **Tower of Hanoi:** Stacks can be used to solve the Tower of Hanoi problem, a classic recursive puzzle.

❖ Push operation:



❖ Pop operation:



Example in python:

```
class Stack:
    def __init__(self, max_size):
        self.max_size = max_size
        self.items = []
    def is_empty(self):
        return len(self.items) == 0
    def is_full(self):
        return len(self.items) == self.max_size
    def push(self, item):
        if not self.is_full():
            self.items.append(item)
        else:
            raise Exception("Stack overflow: Cannot push to a full stack.")
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise Exception("Stack underflow: Cannot pop from an empty stack.")
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            raise Exception("Stack is empty: Cannot peek from an empty stack.")
    def count(self):
        return len(self.items)
```

```
    def change(self, index, value):
        if 0 <= index < len(self.items):
            self.items[index] = value
        else:
            raise IndexError("Index out of range: Cannot change value at the given index.")
    def display(self):
        print("Stack:", self.items)
# Example usage:
stack = Stack(max_size=5)
stack.push(1)
stack.push(2)
stack.push(3)
stack.display()  # Output: Stack: [1, 2, 3]
print("Top element:", stack.peek())  # Output: Top element: 3
stack.pop()
stack.display()  # Output: Stack: [1, 2]
print("Is the stack empty?", stack.is_empty())  # Output: Is the stack empty? False
print("Is the stack full?", stack.is_full())  # Output: Is the stack full? False
print("Number of elements in the stack:", stack.count())  # Output: Number of elements in the stack:
2
stack.change(0, 10)
stack.display()  # Output: Stack: [10, 2]
```

ARRAY IMPLEMENTATION OF STACK

- An array implementation of a stack is a way to implement a stack data structure using an array. In this approach, the elements of the stack are stored in a one-dimensional array, and the top of the stack is represented by an index pointing to the last inserted element in the array. The stack operations (push, pop, peek, is empty, etc.) are performed using array operations.

Program in python:

```
class Stack:
    def __init__(self, max_size):
        self.max_size = max_size
        self.items = [None] * max_size
        self.top = -1
    def is_empty(self):
        return self.top == -1
    def is_full(self):
        return self.top == self.max_size - 1
    def push(self, item):
        if not self.is_full():
            self.top += 1
            self.items[self.top] = item
        else:
            raise Exception("Stack overflow: Cannot push to a full stack.")
    def pop(self):
        if not self.is_empty():
            item = self.items[self.top]
            self.top -= 1
```

```
            return item
        else:
            raise Exception("Stack underflow: Cannot pop from an empty stack.")
    def peek(self):
        if not self.is_empty():
            return self.items[self.top]
        else:
            raise Exception("Stack is empty: Cannot peek from an empty stack.")
    def count(self):
        return self.top + 1
    def display(self):
        if not self.is_empty():
            print("Stack:", self.items[:self.top + 1])
        else:
            print("Stack is empty.")
# Example usage:
stack = Stack(max_size=5)
stack.push(1)
stack.push(2)
stack.push(3)
stack.display()  # Output: Stack: [1, 2, 3]
print("Top element:", stack.peek())  # Output: Top element: 3
stack.pop()
stack.display()  # Output: Stack: [1, 2]
print("Is the stack empty?", stack.is_empty())  # Output: Is the stack empty? False
print("Is the stack full?", stack.is_full())  # Output: Is the stack full? False
print("Number of elements in the stack:", stack.count())  # Output: Number of elements in the stack:
2
```

LINKED LIST IMPLEMENTATION OF STACK

- A linked list implementation of a stack is a way to implement a stack data structure using a singly linked list. In this approach, the elements of the stack are stored in nodes of the linked list, and the top of the stack is represented by the head of the linked list. Each node in the linked list contains an element and a reference (a pointer) to the next node in the stack.

Example in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class Stack:
    def __init__(self):
        self.head = None
    def is_empty(self):
        return self.head is None
    def push(self, item):
        new_node = Node(item)
        new_node.next = self.head
        self.head = new_node
```

```
    def pop(self):
        if not self.is_empty():
            item = self.head.data
            self.head = self.head.next
            return item
        else:
            raise Exception("Stack underflow: Cannot pop from an empty stack.")
    def peek(self):
        if not self.is_empty():
            return self.head.data
        else:
            raise Exception("Stack is empty: Cannot peek from an empty stack.")
    def count(self):
        current = self.head
        count = 0
        while current:
            count += 1
            current = current.next
        return count
    def display(self):
        current = self.head
        stack_items = []
        while current:
            stack_items.append(current.data)
            current = current.next
        print("Stack:", stack_items)
# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.display()  # Output: Stack: [3, 2, 1]
print("Top element:", stack.peek())  # Output: Top element: 3
stack.pop()
stack.display()  # Output: Stack: [2, 1]
print("Is the stack empty?", stack.is_empty())  # Output: Is the stack empty? False
print("Number of elements in the stack:", stack.count())  # Output: Number of elements in the stack:
2
```

QUEUE

- A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, meaning the element that is inserted first will be first one to be removed. It behaves like a real-world queue or line, where the first person to join the line is the first one to be served and leave.
- In a queue, elements are added at the rear (also known as the "enqueue" operation) and removed from the front (also known as the "dequeue" operation).

❖ Queue Operations:

- **Enqueue (or insert):**
  - Enqueue operation adds an element to the rear (end) of the queue.
  - It is also known as "insert" or "push" operation in some contexts.
- **Dequeue (or delete):**
  - Dequeue operation removes and returns the front (first) element from the queue.
  - It is also known as "delete" or "pop" operation in some contexts.
- **Front (or Peek):**
  - Front operation returns the front (first) element from the queue without removing it.
  - It allows you to see the first element in the queue without modifying the queue.
- **Is Empty:**
  - Is Empty operation checks if the queue is empty or contains any elements.
  - It returns true if the queue is empty; otherwise, it returns false.
- **Size:**
  - Size operation returns the number of elements currently in the queue.
  - It provides the count of elements present in the queue.
- **Clear:**
  - The Clear operation removes all elements from the queue, making it empty.
  - This operation resets the queue, and after calling Clear, the queue will have no elements.
- **Search (or contains):**
  - The Search operation checks if a given element is present in the queue.
  - It returns true if the element is found; otherwise, it returns false.
- **Rear (or back):**
  - The Rear operation returns the rear (last) element from the queue without removing it.
  - This is the opposite of the Front operation, which returns the front (first) element.
- **Priority Queue Operations:**
  - Priority queues are a variation of standard queue where elements have assigned priorities.
  - In addition to Enqueue and Dequeue, priority queues often have operations like Insert with Priority and Get Highest Priority Element.

❖ Application of Queue:

1. **Task Scheduling:** Queues are used to manage tasks and processes that need to be executed in a specific order. For example, in an operating system, the task scheduler uses a queue to prioritize and manage running processes.
2. **Printer Spooling:** In printer queues, print jobs are enqueued based on their arrival time. The printer dequeues and processes print jobs in the order they were added, ensuring fair printing for all users.
3. **Breadth-First Search (BFS):** BFS algorithm uses a queue to explore all neighboring nodes of a graph in breadth-first order. It is commonly used in graph traversals, shortest path finding, and network routing.
4. **Simulation Systems:** Queues are used in simulations to model various scenarios where entities need to wait for resources or services. For example, in a bank simulation, customers wait in a queue to be served by tellers.
5. **Network Packet Queuing:** Routers and network switches use queues to manage incoming network packets. This ensures that packets are forwarded in the order they arrive and helps manage network congestion.
6. **Message Queues:** In message queuing systems, messages are stored in a queue until they are processed by the consumer. This pattern is used in inter-process communication and distributed systems.
7. **Task Processing:** In multithreaded applications, queues are used to manage tasks and balance the workload across multiple threads, ensuring efficient resource utilization.
8. **Call Center Management:** In call centers, customer calls are queued until a customer service representative becomes available to handle the call.
9. **Traffic Management:** Traffic lights at intersections can be managed using queues to ensure smooth traffic flow and reduce congestion.
10. **Print Job Management:** Printers can use queues to manage print jobs and prioritize certain print jobs over others.

❖ Types of Queues :

1. **Linear Queue:** A linear queue is a basic queue where elements are added at the rear (enqueue) and removed from the front (dequeue). It follows the First-In-First-Out (FIFO) principle.
2. **Circular Queue:** A circular queue is an extension of the linear queue where the rear and front pointers wrap around, forming a circular buffer. This allows efficient space utilization and avoids wastage of memory.
3. **Priority Queue:** A priority queue is a type of queue where elements have associated priorities. Elements with higher priorities are dequeued before elements with lower priorities.
4. **Double-Ended Queue (Deque):** A deque allows insertion and deletion of elements from both the front and rear. It supports operations like push front, push back, pop front, and pop back.
5. **Blocking Queue:** A blocking queue is a queue that blocks (waits) when attempting to dequeue from an empty queue or enqueue into a full queue. It is commonly used in concurrent programming to synchronize data between threads.
6. **Concurrent Queue:** A concurrent queue is designed to handle multiple concurrent accesses from multiple threads safely. It employs synchronization mechanisms like locks or atomic operations to ensure thread-safety.

7. **Delay Queue:** A delay queue is a special type of queue where elements are enqueued with a specified delay, and they are dequeued only after the delay has passed.
8. **D-Queue:** A D-queue is a generalized queue that can have more than two ends, allowing insertion and deletion from multiple sides. It is also known as a "double-ended priority queue."
9. **Double Queue:** A double queue is a combination of two queues, one for inserting elements and another for removing elements. It is also known as a "dual queue."
10. **Banker's Queue:** A banker's queue is a priority queue used in resource management algorithms to allocate resources efficiently.
11. **Randomized Queue:** A randomized queue is a data structure where elements are randomly removed when dequeued, rather than following a specific order.

Example in python:

1. Linear Queue (Basic Queue):

```
class LinearQueue:
    def __init__(self):
        self.queue = []
    def is_empty(self):
        return len(self.queue) == 0
    def enqueue(self, item):
        self.queue.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            raise Exception("Queue is empty: Cannot dequeue.")
# Example usage:
linear_queue = LinearQueue()
linear_queue.enqueue(1)
linear_queue.enqueue(2)
linear_queue.enqueue(3)
print(linear_queue.dequeue())  # Output: 1
print(linear_queue.dequeue())  # Output: 2
print(linear_queue.is_empty())  # Output: False
```

2. Circular Queue:

```
class CircularQueue:
    def __init__(self, max_size):
        self.max_size = max_size
        self.queue = [None] * max_size
        self.front = self.rear = -1
    def is_empty(self):
        return self.front == -1
    def is_full(self):
        return (self.rear + 1) % self.max_size == self.front
    def enqueue(self, item):
        if not self.is_full():
            if self.is_empty():
                self.front = self.rear = 0
```

```
            else:
                self.rear = (self.rear + 1) % self.max_size
            self.queue[self.rear] = item
        else:
            raise Exception("Queue is full: Cannot enqueue.")
    def dequeue(self):
        if not self.is_empty():
            item = self.queue[self.front]
            if self.front == self.rear:
                self.front = self.rear = -1
            else:
                self.front = (self.front + 1) % self.max_size
            return item
        else:
            raise Exception("Queue is empty: Cannot dequeue.")
# Example usage:
circular_queue = CircularQueue(max_size=3)
circular_queue.enqueue(1)
circular_queue.enqueue(2)
circular_queue.enqueue(3)
print(circular_queue.dequeue())  # Output: 1
print(circular_queue.dequeue())  # Output: 2
print(circular_queue.is_empty())  # Output: False
```

3. Priority Queue (Using Python's built-in `heapq` module):

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.queue = []
    def is_empty(self):
        return len(self.queue) == 0
    def enqueue(self, item, priority):
        heapq.heappush(self.queue, (priority, item))
    def dequeue(self):
        if not self.is_empty():
            return heapq.heappop(self.queue)[1]
        else:
            raise Exception("Priority queue is empty: Cannot dequeue.")
# Example usage:
priority_queue = PriorityQueue()
priority_queue.enqueue("Task 1", 2)
priority_queue.enqueue("Task 2", 1)
priority_queue.enqueue("Task 3", 3)
print(priority_queue.dequeue())  # Output: Task 2
print(priority_queue.dequeue())  # Output: Task 1
print(priority_queue.is_empty())  # Output: False
```

4. Double-Ended Queue (Deque) (Using Python's `collections.deque`):

```
from collections import deque
class Deque:
    def __init__(self):
        self.deque = deque()
    def is_empty(self):
        return len(self.deque) == 0
    def enqueue_front(self, item):
        self.deque.appendleft(item)
    def enqueue_rear(self, item):
        self.deque.append(item)
    def dequeue_front(self):
        if not self.is_empty():
            return self.deque.popleft()
        else:
            raise Exception("Deque is empty: Cannot dequeue from front.")
    def dequeue_rear(self):
        if not self.is_empty():
            return self.deque.pop()
        else:
            raise Exception("Deque is empty: Cannot dequeue from rear.")
# Example usage:
deque_example = Deque()
deque_example.enqueue_front(1)
deque_example.enqueue_rear(2)
deque_example.enqueue_front(3)
print(deque_example.dequeue_front())  # Output: 3
print(deque_example.dequeue_rear())  # Output: 2
print(deque_example.is_empty())  # Output: False
```

ARRAY REORIENTATION IN QUEUE

- In the array representation of a queue, we use a fixed-size array to store the elements of the queue. The front and rear pointers are used to keep track of the front and rear positions of the queue, respectively. As elements are enqueued and dequeued, the front and rear pointers are adjusted accordingly.

Program in Python:

```
class ArrayQueue:
    def __init__(self, max_size):
        self.max_size = max_size
        self.queue = [None] * max_size
        self.front = self.rear = -1
    def is_empty(self):
        return self.front == -1
    def is_full(self):
        return (self.rear + 1) % self.max_size == self.front
    def enqueue(self, item):
        if not self.is_full():
            if self.is_empty():
```

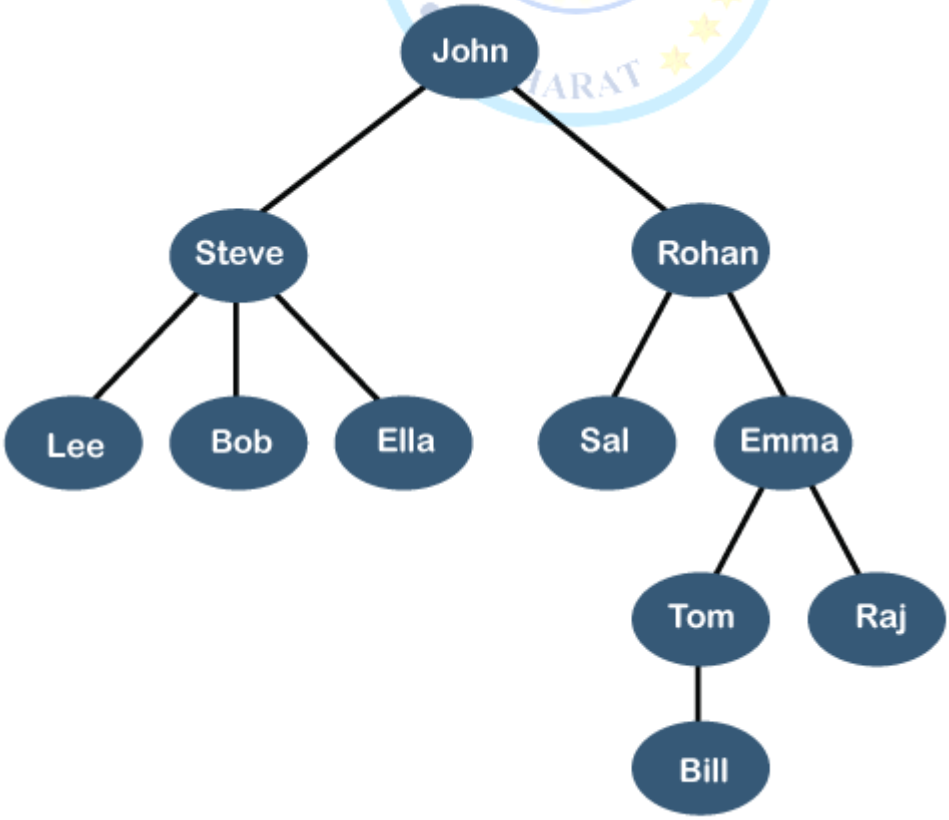
```
            self.front = self.rear = 0
        else:
            self.rear = (self.rear + 1) % self.max_size
        self.queue[self.rear] = item
    else:
        raise Exception("Queue is full: Cannot enqueue.")
  def dequeue(self):
    if not self.is_empty():
        item = self.queue[self.front]
        if self.front == self.rear:
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.max_size
        return item
    else:
        raise Exception("Queue is empty: Cannot dequeue.")
# Example usage:
queue = ArrayQueue(max_size=5)
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.dequeue())  # Output: 1
print(queue.dequeue())  # Output: 2
print(queue.is_empty())  # Output: False
Linked list implimation in queue
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None
class LinkedListQueue:
  def __init__(self):
    self.front = self.rear = None
  def is_empty(self):
    return self.front is None
  def enqueue(self, item):
    new_node = Node(item)
    if self.rear is None:
        self.front = self.rear = new_node
    else:
        self.rear.next = new_node
        self.rear = new_node
  def dequeue(self):
    if not self.is_empty():
        item = self.front.data
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        return item
```

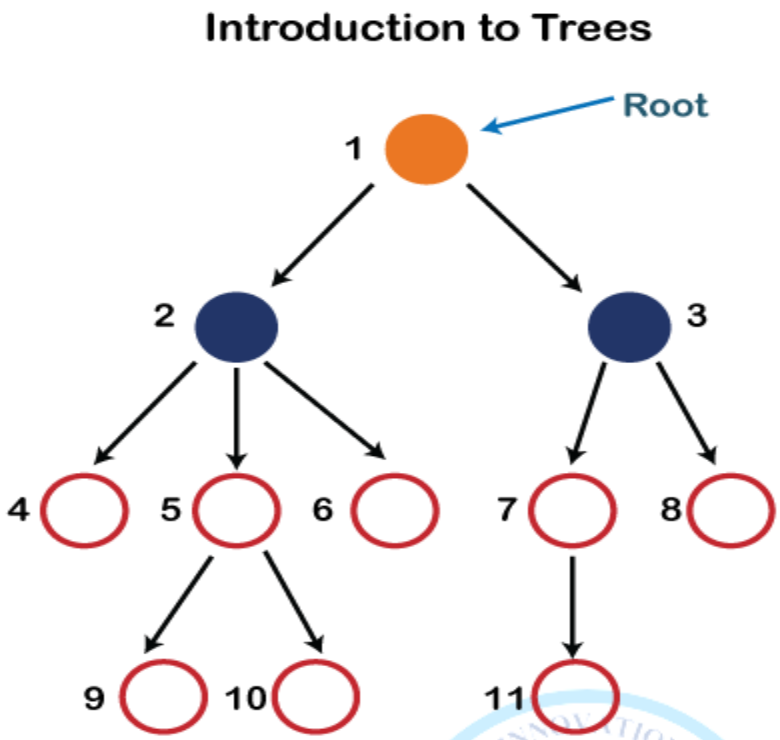
```
        else:
            raise Exception("Queue is empty: Cannot dequeue.")
# Example usage:
queue = LinkedListQueue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.dequeue())  # Output: 1
print(queue.dequeue())  # Output: 2
print(queue.is_empty())  # Output: False
```

TREE

- array, linked list, stack and queue in which all the elements are arranged in a sequential manner.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner.
- It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In a tree data structure, there is a special node called the "root" node that serves as the starting point of the tree. Each node in the tree can have zero or more child nodes, forming a parent-child relationship. Nodes with no children are known as "leaf" nodes.
- The topmost node (root) has no parent, and each child node has exactly one parent, except for the root node. Nodes with the same parent are called "siblings."

Example:





❖ **Some basic terms used in Tree data structure.**

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node: -** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree.
- **Internal nodes:** A node has at least one child node known as an internal
- **Ancestor node: -** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. (पूर्वज नोड: - नोड का पूर्वज रूट से उस नोड तक के पथ पर कोई भी पूर्ववर्ती नोड होता है। रूट नोड का कोई पूर्वज नहीं है.)
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. (वंशज: दिए गए नोड के तत्काल उत्तराधिकारी को नोड के वंशज के रूप में जाना जाता है।)

❖ Application of tree

➢ Trees have numerous applications in various fields due to their hierarchical and organized structure.

1. **File Systems:** Trees are commonly used to represent the hierarchical structure of file systems in operating systems. Directories and subdirectories are organized in a tree-like structure, with the root directory as the starting point.

2. **Data Organization:** Trees are used to organize data efficiently in databases and indexing systems. Binary Search Trees (BSTs) and Balanced Trees are employed for faster search, insertion, and deletion operations.
3. **Compiler Design:** Abstract Syntax Trees (ASTs) are used in compilers to represent the hierarchical structure of source code. ASTs aid in parsing and semantic analysis during the compilation process.
4. **Network Routing:** Trees are utilized in computer networks for routing packets efficiently. Hierarchical routing algorithms like OSPF (Open Shortest Path First) use trees to organize network nodes.
5. **Organizational Hierarchies:** Trees can represent organizational structures, with each node representing an employee and its children representing their subordinates. This allows easy navigation of the management hierarchy.
6. **AI and Game Theory:** Game trees are employed in artificial intelligence for decision-making in games and strategy planning. Algorithms like Minimax and Alpha-Beta Pruning are used for optimal game moves.
7. **Hierarchical Data Representation:** Trees are used to represent hierarchical data in various applications, such as XML and JSON formats in web development and data serialization.
8. **Family Trees and Genealogy:** Trees are used to represent family relationships and genealogy data, helping trace ancestry and relationships.
9. **Expression Parsing:** Trees are used in expression parsing, where mathematical expressions are represented as expression trees, aiding in evaluation and simplification.
10. **Decision Trees:** In machine learning, decision trees are used for classification and regression tasks. They help make decisions based on features or attributes of data.
11. **Huffman Encoding:** Huffman trees are used in data compression algorithms like Huffman coding to create efficient binary encodings for data.
12. **Binary Search:** Binary Search Trees (BSTs) are used for efficient searching and sorting operations in various applications.

❖ Types of tree :

1. **Binary Tree:** A tree in which each node can have at most two children.
2. **Binary Search Tree (BST):** A binary tree where the left child contains elements less than the node's value, and the right child contains elements greater than the node's value.
3. **AVL Tree:** A self-balancing binary search tree where the height difference between left and right subtrees of any node is at most 1.
4. **Red-Black Tree:** A self-balancing binary search tree with properties that ensure it remains balanced.
5. **B-Tree:** A multi-way search tree designed for efficient disk-based data storage and retrieval.
6. **Trie (Prefix Tree):** A tree-like data structure used for efficient retrieval of strings based on prefixes.
7. **Segment Tree:** A tree used for handling interval or range-based queries efficiently.
8. **Suffix Tree:** A specialized tree used to represent all the suffixes of a given string.
9. **Heap:** A binary tree-based data structure used to implement priority queues.
10. **Trie (Digital Tree):** A data structure used for fast and efficient retrieval of strings with common prefixes.

Example in Python:

1. Binary Tree:

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Example usage:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
```

2. Binary Search Tree (BST):

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def insert(self, key):
        self.root = self._insert_recursively(self.root, key)
    def _insert_recursively(self, node, key):
        if node is None:
            return TreeNode(key)
        if key < node.key:
            node.left = self._insert_recursively(node.left, key)
        else:
            node.right = self._insert_recursively(node.right, key)
        return node
    def in_order_traversal(self, node):
        if node is not None:
            self.in_order_traversal(node.left)
            print(node.key, end=" ")
            self.in_order_traversal(node.right)
# Example usage:
bst = BST()
bst.insert(5)
bst.insert(3)
bst.insert(7)
bst.in_order_traversal(bst.root)  # Output: 3 5 7
```

3. AVL Tree:

```
# AVL Tree implementation can be done by modifying the BST implementation with balancing functions.
# For simplicity, let's use a Python library.
from avl_tree import AVLTree
# Example usage:
avl_tree = AVLTree()
avl_tree.insert(5)
avl_tree.insert(3)
avl_tree.insert(7)

avl_tree.in_order_traversal()  # Output: 3 5 7
```

4. Red-Black Tree:

```
# Red-Black Tree implementation can be done by modifying the BST implementation with coloring functions.
# For simplicity, let's use a Python library.
from rbtree import RedBlackTree
# Example usage:
rb_tree = RedBlackTree()
rb_tree.insert(5)
rb_tree.insert(3)
rb_tree.insert(7)
rb_tree.in_order_traversal()  # Output: 3 5 7
```

5. B-Tree:

```
# B-Tree implementation can be done using external libraries or custom implementation based on specific requirements.
# Example usage of external library (bintrees):
from bintrees import FastBTree
b_tree = FastBTree()
b_tree[5] = 'data1'
b_tree[3] = 'data2'
b_tree[7] = 'data3'
print(b_tree.min_item())  # Output: (3, 'data2')
print(b_tree.max_item())  # Output: (7, 'data3')
```

6. Trie (Prefix Tree):

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
class Trie:
    def __init__(self):
        self.root = TrieNode()
    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
```

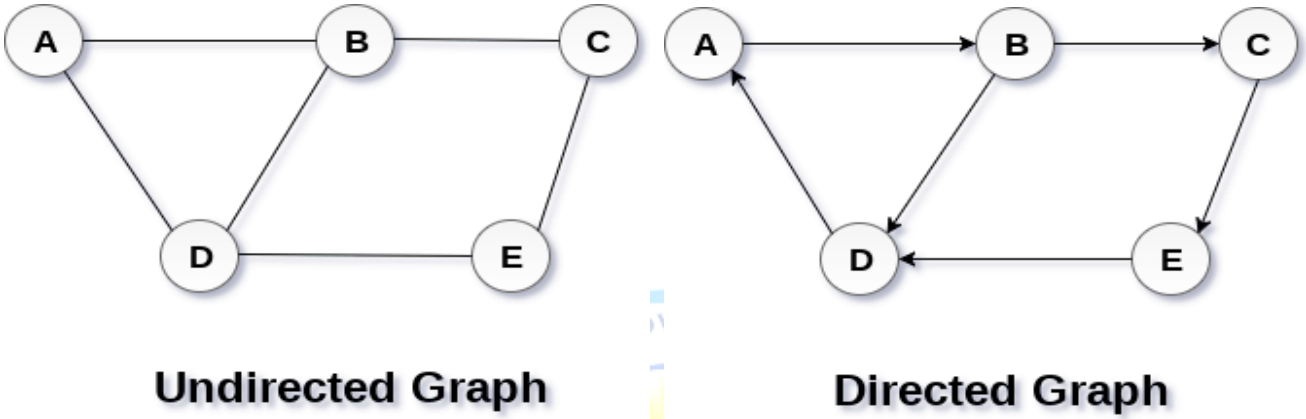
```
            node = node.children[char]
        node.is_end_of_word = True
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

# Example usage:
trie = Trie()
trie.insert("apple")
trie.insert("banana")
trie.insert("orange")
print(trie.search("apple"))  # Output: True
print(trie.search("grape"))  # Output: False
```

GRAPH

- A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.
- A graph is a collection of nodes (also known as vertices) connected by edges.
- A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.
- A graph can be directed or undirected.

Example:



Undirected Graph **Directed Graph**

Note: A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the above figure.

❖ Graph Terminology

- **Path:** A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
- **Closed Path:** A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.
- **Simple Path:** If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.
- **Cycle:** A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.
- **Connected Graph:** A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
- **Complete Graph:** A complete graph is the one in which every node is connected with all other nodes.
- **Weighted Graph:** In a weighted graph, each edge is assigned with some data such as length or weight.
- **Digraph:** A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
- **Loop:** An edge that is associated with the similar end points can be called as Loop.

- **Adjacent Nodes:** If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
- **Degree of the Node:** A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

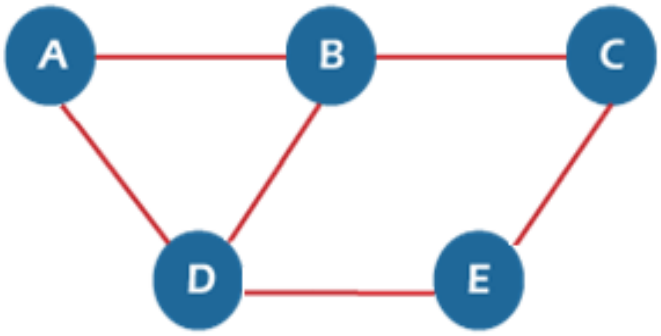
❖ Graph representation:

- A graph is a data structure that consist a set of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:
  1. Sequential representation (or, Adjacency matrix representation)
  2. Linked list representation (or, Adjacency list representation)

1. Sequential representation:

- In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph.
- We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

❖ adjacency matrix representation of an undirected graph:

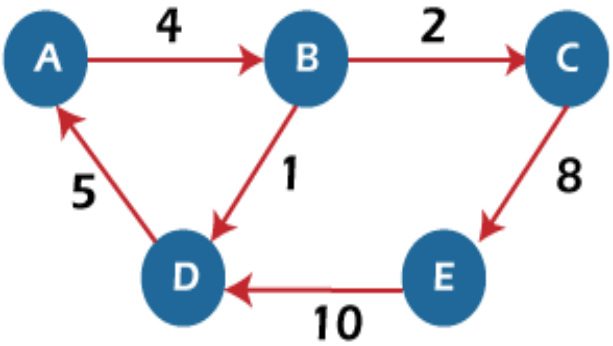


Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

❖ Adjacency matrix for a directed graph:

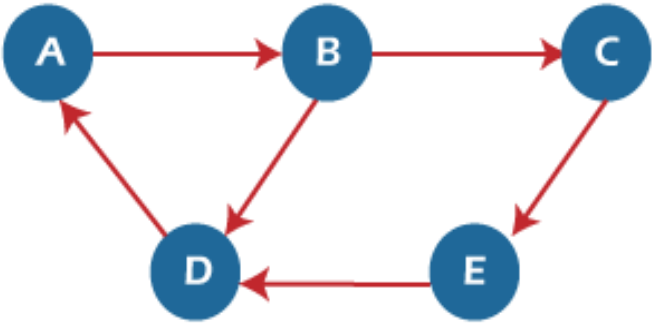


weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

❖ **Adjacency matrix for a directed graph:**



Directed Graph

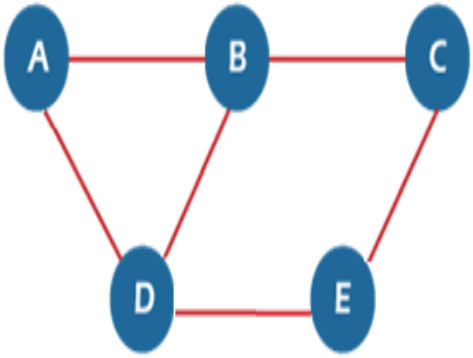
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

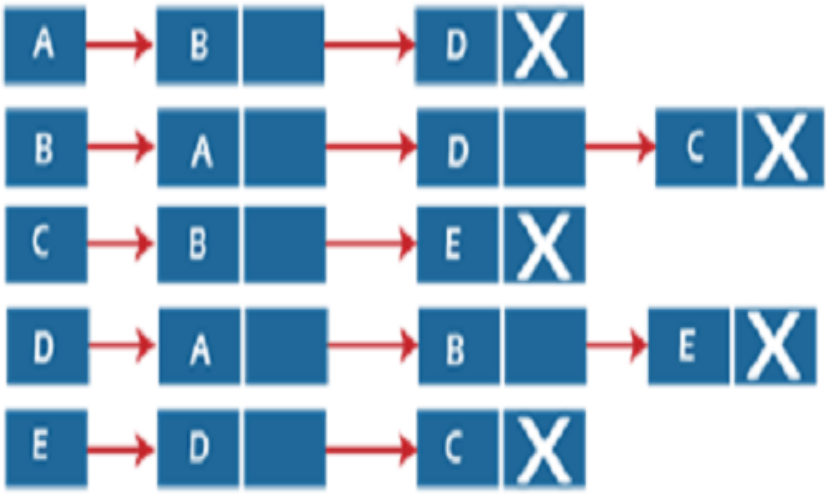
2. Linked list representation:

- An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

❖ **list representation of an undirected graph.**



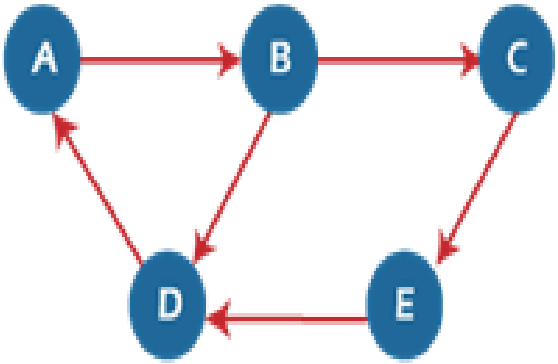
Undirected Graph



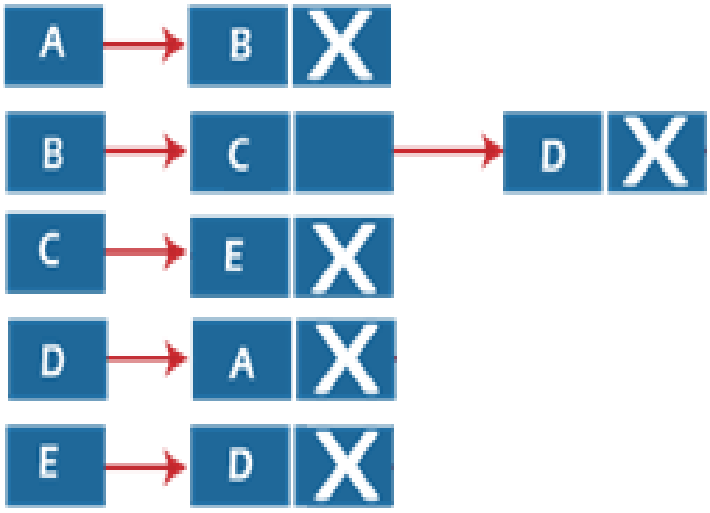
Adjacency List

- In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

❖ **directed graph, list representation :**

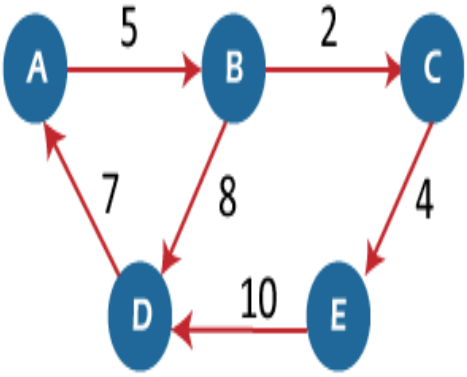


Directed Graph

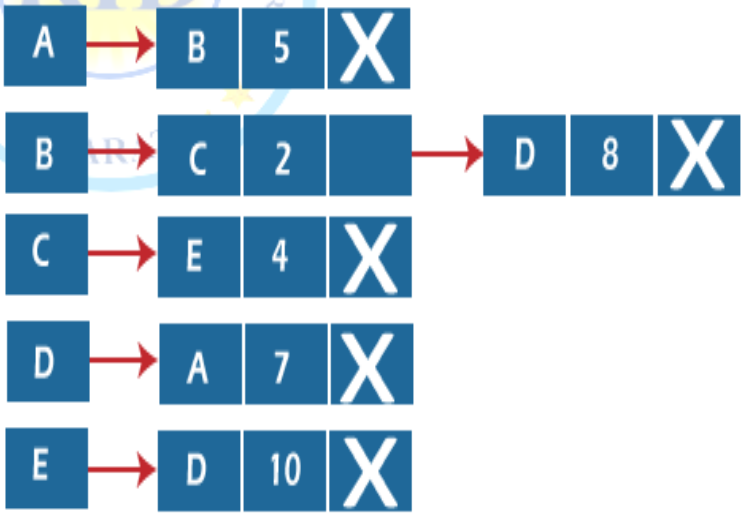


Adjacency List

❖ **weighted directed graph:**



Weighted Directed Graph



Adjacency List

❖ Application of graph:

1. **Social Networks:** Graphs are extensively used to model social networks like Facebook, Twitter, and LinkedIn, where users are represented as nodes, and friendships/connections are represented as edges.
2. **Computer Networks:** Graphs are used to model computer networks, with nodes representing network devices (routers, switches) and edges representing network connections.
3. **Route Planning and Navigation:** Graphs are used to represent road networks and transportation systems, helping with route planning and navigation.
4. **Internet and Web:** Graphs are used in web crawlers and search engines to index and represent hyperlinks between web pages.
5. **Dependency Management:** In software engineering, graphs are used to represent dependencies between software components, aiding in build and package management.
6. **Game Development:** Graphs are used in game development to represent game maps, character movements, and AI decision-making.
7. **Recommendation Systems:** Graphs can be used to build recommendation systems, suggesting products, movies, or content based on user preferences and relationships.
8. **Chemical Molecules:** Graphs are used to model chemical compounds and molecules, where nodes represent atoms and edges represent chemical bonds.
9. **Logistics and Supply Chain Management:** Graphs are used to optimize logistics and supply chain networks, helping with route planning and resource allocation.
10. **Network Analysis:** Graphs are used in social network analysis, traffic flow analysis, and other analytical tasks.
11. **Image Processing:** Graphs are used in image segmentation and object detection algorithms.
12. **Game Theory:** Graphs are used in game theory to model strategic interactions and outcomes in games and decision-making processes.

❖ Types of graphs:

1. **Undirected Graph:**

- An undirected graph is a graph in which the edges have no direction. The relationship between nodes is symmetric, and movement is bidirectional.

2. **Directed Graph (Digraph):**

- A directed graph is a graph in which each edge has a specific direction. The relationship between nodes is asymmetric, and movement is unidirectional.

3. Weighted Graph:

- A weighted graph is a graph where each edge has an associated weight or cost. These weights represent the distance, cost, or strength of the relationship between nodes.

4. Unweighted Graph:

- An unweighted graph is a graph where all edges have the same weight or no weight. The edges represent only the presence of a connection between nodes.

5. Cyclic Graph:

- A cyclic graph is a graph that contains at least one cycle (a path that starts and ends at the same node). In other words, there is a way to return to a node by following the edges.

6. Acyclic Graph:

- An acyclic graph is a graph that has no cycles. There is no way to return to a node by following the edges.

7.Connected Graph:

- A connected graph is a graph in which there is a path between every pair of nodes. There are no isolated nodes or disconnected components.

8. Disconnected Graph:

- A disconnected graph is a graph that has two or more connected components. There is no path between nodes in different components.

9. Complete Graph:

- A complete graph is a graph where every pair of distinct nodes is connected by a unique edge. It has the maximum possible number of edges for the given number of nodes.

10. Bipartite Graph:

- A bipartite graph is a graph whose nodes can be partitioned into two disjoint sets, such that all edges connect nodes from different sets.

11. Eulerian Graph:

- A Eulerian graph is a graph that contains a Eulerian cycle, which is a cycle that visits every edge exactly once.

12. Hamiltonian Graph:

- A Hamiltonian graph is a graph that contains a Hamiltonian cycle, which is a cycle that visits every node exactly once.

❖ Example in python:

- implementation of an undirected graph in Python:

```
class Graph:
    def __init__(self):
        self.graph = {}
    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)
    def remove_edge(self, u, v):
        if u in self.graph and v in self.graph:
            self.graph[u].remove(v)
            self.graph[v].remove(u)
    def get_neighbors(self, node):
        return self.graph.get(node, [])
# Example usage:
g = Graph()
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(4, 1)
g.add_edge(2, 4)
print("Neighbors of node 2:", g.get_neighbors(2))  # Output: Neighbors of node 2: [1, 3, 4]
g.remove_edge(1, 2)
print("Neighbors of node 1:", g.get_neighbors(1))  # Output: Neighbors of node 1: [4]
```

implementation of the Breadth-First Search (BFS) algorithm in Python for a graph represented using an adjacency list:

```
from collections import deque
class Graph:
    def __init__(self):
        self.graph = {}
    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
    def bfs(self, start_node):
        visited = set()
        queue = deque([start_node])
        visited.add(start_node)
        while queue:
            current_node = queue.popleft()
            print(current_node, end=" ")
            for neighbor in self.graph.get(current_node, []):
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)
# Example usage:
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
g.add_edge(2, 4)
g.add_edge(3, 4)
g.add_edge(4, 5)
print("BFS Traversal from node 1:")
g.bfs(1)  # Output: 1 2 3 4 5
```

Example:

- implementation of the Depth-First Search (DFS) algorithm in Python for a graph represented using an adjacency list:

```
class Graph:
    def __init__(self):
        self.graph = {}
    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
    def dfs(self, start_node):
        visited = set()
        self._dfs_recursive(start_node, visited)
    def _dfs_recursive(self, node, visited):
        visited.add(node)
```

```
        print(node, end=" ")
        for neighbor in self.graph.get(node, []):
            if neighbor not in visited:
                self._dfs_recursive(neighbor, visited)
# Example usage:
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
g.add_edge(2, 4)
g.add_edge(3, 4)
g.add_edge(4, 5)
print("DFS Traversal from node 1:")
g.dfs(1)  # Output: 1 2 3 4 5
```

SPANNING

- It is a subgraph of a connected, undirected graph that includes all the vertices of the original graph while forming a tree without any cycles. In other words, a spanning tree is a subset of the edges of the original graph that ensures all the nodes are reachable, and there are no redundant or extraneous edges.

❖ Application of spanning tree:

1. Network Design: Construct efficient communication networks by eliminating loops and ensuring connectivity.
2. Network Routing: Determine the shortest paths for data transmission in computer networks.
3. Electrical Distribution Networks: Design power distribution systems without forming redundant loops.
4. Cluster Analysis: Identify hierarchical relationships and patterns within datasets.
5. Approximation Algorithms: Provide efficient solutions to optimization problems, such as the minimum spanning tree problem.
6. Image Processing: Extract object shapes' major axes for skeletonization and representation.
7. Circuit Design: Reduce complexity and enhance efficiency in electronic circuit layouts.
8. Broadcast Algorithms: Efficiently deliver messages to all nodes in distributed systems.
9. Wireless Sensor Networks: Optimize data dissemination and energy consumption for extended network lifetime.
10. Geographic Information Systems (GIS): Design optimal transportation routes and utility networks.
11. Game Theory: Analyze strategies and outcomes in graph-based games using spanning trees.
12. Resource Management: Allocate resources optimally in various applications, like cloud computing and project scheduling.
13. Tree-Based Searching: Efficiently explore possibilities in search algorithms, such as spanning tree search in artificial intelligence.
14. Load Balancing: Distribute workloads evenly across a network for better performance.
15. Fault Tolerance: Create backup routes to maintain network connectivity in the event of failures

❖ Algorithms for Minimum spanning tree

1). Prim's Algorithm
2). Kruskal's Algorithm

1). Prim's Algorithm:

- Prim's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, weighted graph. The MST is a subset of the original graph's edges that connects all the vertices with the minimum total edge weight, without forming any cycles.

❖ Example in python:

```
import heapq
def prim(graph):
    mst = []
    visited = set()
    start_vertex = next(iter(graph))  # Start with an arbitrary vertex
    # Priority queue to keep track of edges with their weights
    pq = [(0, start_vertex)]
    while pq:
        weight, current_vertex = heapq.heappop(pq)
        if current_vertex not in visited:
            visited.add(current_vertex)
            if weight > 0:
                mst.append((current_vertex, weight))
            for neighbor, edge_weight in graph[current_vertex].items():
                if neighbor not in visited:
                    heapq.heappush(pq, (edge_weight, neighbor))
    return mst
```

2). Kruskal's Algorithm:

- Kruskal's algorithm is another greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, weighted graph. Like Prim's algorithm, the MST is a subset of the original graph's edges that connects all the vertices with the minimum total edge weight, without forming any cycles.

Example in python:

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
    def find(self, x):
        if x != self.parent[x]:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, x, y):
        root_x, root_y = self.find(x), self.find(y)
        if root_x == root_y:
            return False
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
```

```
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            self.parent[root_y] = root_x
            self.rank[root_x] += 1
        return True
def kruskal(graph):
    mst = []
    edges = []
    n = len(graph)
    # Flatten the graph's adjacency list into a list of edges
    for u in graph:
        for v, weight in graph[u].items():
            edges.append((weight, u, v))
    # Sort edges in non-decreasing order of weight
    edges.sort()
    # Create a Union-Find data structure to track connected components
    uf = UnionFind(n)
    for weight, u, v in edges:
        if uf.union(u, v):
            mst.append((u, v, weight))
    return mst
```

❖ Linear Search:

- Linear search, also known as sequential search, is a simple search algorithm that sequentially checks each element in a list until a match is found or the end of the list is reached.

❖ Applications of Linear Search:

1. **Unsorted Lists:** Linear search is suitable for searching elements in unsorted lists or arrays when the data is not organized in any specific order.
2. **Short Lists:** For small lists, linear search can be a reasonable choice as the overhead of sorting and implementing more complex algorithms (like binary search) may not be necessary.
3. **Sequential Access:** Linear search is often used in applications where elements are accessed sequentially, such as traversing linked lists or performing simple operations on small data sets.
4. **Searching for Multiple Occurrences:** Linear search can efficiently find all occurrences of an element in a list, as it will keep searching even after finding a match.

❖ Example in Python:

```
def linear_search(arr, target):
    for i, item in enumerate(arr):
        if item == target:
            return i
    return -1
```

❖ Binary Search:

- Binary search is an efficient search algorithm that works on a sorted list. It repeatedly divides the search space in half and compares the middle element with the target. It then eliminates half of the remaining search space based on the comparison.

❖ **Applications of Binary Search:**

1. Sorted Lists: Binary search is highly effective on sorted lists or arrays. It significantly reduces the search space and provides faster search times compared to linear search.
2. Large Datasets: For large datasets, binary search is preferred due to its logarithmic time complexity (O(log n)), which is much faster than linear search's linear time complexity (O(n)).
3. Numerical Data: Binary search is commonly used when searching for numeric values in sorted collections like arrays or databases.
4. Divide and Conquer Algorithms: Binary search is a fundamental component of many divide and conquer algorithms, such as searching, sorting, and various optimization problems.
5. Efficient Searching: In scenarios where time efficiency is crucial, binary search is an excellent choice for quickly locating elements.
6. Application with Pointers: Binary search is often implemented in memory-constrained environments where pointer manipulation and recursive algorithms are discouraged due to stack space limitations.

Example in python:

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

SORTING

1. Bubble Sort: Repeatedly swap adjacent elements until the list is sorted.
2. Bucket Sort: Divide input into buckets, sort each bucket, then merge.
3. Comb Sort: Improve bubble sort by reducing the gap between comparisons.
4. Counting Sort: Integer-based sort; counts occurrences and reconstructs.
5. Heap Sort: Uses a binary heap to extract the minimum element repeatedly.
6. Insertion Sort: Build the sorted list one element at a time by shifting.
7. Merge Sort: Divide and conquer; recursively sort and merge sublists.
8. Quick Sort: Divide and conquer; recursively partition and sort sublists.
9. Radix Sort: Sorts by digits from the least to the most significant.
10. Selection Sort: Repeatedly selects the minimum element and places it at the beginning.
11. Shell Sort: Extended insertion sort using a sequence of gaps.
12. Bitonic Sort: Creates bitonic sequences and merges them.
13. Cocktail Sort: Sorts in both directions using bidirectional bubble sort.
14. Cycle Sort: Minimizes memory writes by directly swapping elements.
15. Tim Sort: Hybrid algorithm combining merge sort and insertion sort.

1. Bubble Sort:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Bubble Sort Output:", arr)
```

Output:

Bubble Sort Output: [11, 12, 22, 25, 34, 64, 90]

2. Bucket Sort:

```
def bucket_sort(arr):
    buckets = [[] for _ in range(10)]
    for num in arr:
        index = int(num * 10)
        buckets[index].append(num)
    arr.clear()
    for bucket in buckets:
        arr.extend(sorted(bucket))
# Example usage:
arr = [0.42, 0.32, 0.33, 0.52, 0.37, 0.47, 0.51]
bucket_sort(arr)
print("Bucket Sort Output:", arr)
```

Output:

Bucket Sort Output: [0.32, 0.33, 0.37, 0.42, 0.47, 0.51, 0.52]

3. Comb Sort:

```
def comb_sort(arr):
    n = len(arr)
    gap = n
    shrink = 1.3
    swapped = True
    while gap > 1 or swapped:
        gap = int(gap / shrink)
        if gap < 1:
            gap = 1
        i = 0
        swapped = False
        while i + gap < n:
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
                swapped = True
            i += 1
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
comb_sort(arr)
print("Comb Sort Output:", arr)
```

Output:

Comb Sort Output: [11, 12, 22, 25, 34, 64, 90]

4. Counting Sort:

```
def counting_sort(arr):
    max_val = max(arr)
    min_val = min(arr)
    range_val = max_val - min_val + 1
    count = [0] * range_val
    output = [0] * len(arr)
    for num in arr:
        count[num - min_val] += 1
    for i in range(1, range_val):
        count[i] += count[i - 1]
    for num in reversed(arr):
        output[count[num - min_val] - 1] = num
        count[num - min_val] -= 1
    arr[:] = output
# Example usage:
arr = [4, 2, 2, 8, 3, 3, 1]
counting_sort(arr)
print("Counting Sort Output:", arr)
```

Output:

Counting Sort Output: [1, 2, 2, 3, 3, 4, 8]

5. Heap Sort:

```
def heap_sort(arr):
```

```
    def heapify(arr, n, i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2
        if left < n and arr[i] < arr[left]:
            largest = left
        if right < n and arr[largest] < arr[right]:
            largest = right
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
heap_sort(arr)
print("Heap Sort Output:", arr)
```

Output:

Heap Sort Output: [11, 12, 22, 25, 34, 64, 90]

6. Insertion Sort:

```
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
insertion_sort(arr)
print("Insertion Sort Output:", arr)
```

Output:

Insertion Sort Output: [11, 12, 22, 25, 34, 64, 90]

7. Merge Sort:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
```

```
    i = j = k = 0
    while i < len(left_half) and j < len(right_half):
      if left_half[i] < right_half[j]:
        arr[k] = left_half[i]
        i += 1
      else:
        arr[k] = right_half[j]
        j += 1
      k += 1
    while i < len(left_half):
      arr[k] = left_half[i]
      i += 1
      k +=1
    while j < len(right_half):
      arr[k] = right_half[j]
      j += 1
      k += 1
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
merge_sort(arr)
print("Merge Sort Output:", arr)
```

Output:

Merge Sort Output: [11, 12, 22, 25, 34, 64, 90]

8. Quick Sort:

```
def quick_sort(arr):
  def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
      if arr[j] <= pivot:
        i += 1
        arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
  def quick_sort_helper(arr, low, high):
    if low < high:
      pi = partition(arr, low, high)
      quick_sort_helper(arr, low, pi - 1)
      quick_sort_helper(arr, pi + 1, high)
  quick_sort_helper(arr, 0, len(arr) - 1)
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
quick_sort(arr)
print("Quick Sort Output:", arr)
```

Output:

Quick Sort Output: [11, 12, 22, 25, 34, 64, 90]

9. Radix Sort:

```
def radix_sort(arr):
    def counting_sort_for_radix(arr, exp):
        n = len(arr)
        count = [0] * 10
        output = [0] * n
        for num in arr:
            index = num // exp
            count[index % 10] += 1
        for i in range(1, 10):
            count[i] += count[i - 1]
        i = n - 1
        while i >= 0:
            index = arr[i] // exp
            output[count[index % 10] - 1] = arr[i]
            count[index % 10] -= 1
            i -= 1
        arr[:] = output
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:
        counting_sort_for_radix(arr, exp)
        exp *= 10
# Example usage:
arr = [170, 45, 75, 90, 802, 24, 2, 66]
radix_sort(arr)
print("Radix Sort Output:", arr)
```

Output:

Radix Sort Output: [2, 24, 45, 66, 75, 90, 170, 802]

10. Selection Sort:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
selection_sort(arr)
print("Selection Sort Output:", arr)
```

Output:

Selection Sort Output: [11, 12, 22, 25, 34, 64, 90]

11. Shell Sort:

```
def shell_sort(arr):
    n = len(arr)
    gap = n // 2
```

```
    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
arr = [64, 34, 25, 12, 22, 11, 90]
shell_sort(arr)
print("Shell Sort Output:", arr)
```

Output:

Shell Sort Output: [11, 12, 22, 25, 34, 64, 90]

12. Bitonic Sort:

Note: Bitonic Sort is usually implemented in parallel environments and is more complex to show in a single-line explanation.

13. Cocktail Sort:

```
def cocktail_sort(arr):
    n = len(arr)
    swapped = True
    start = 0
    end = n - 1
    while swapped:
        swapped = False
        for i in range(start, end):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end -= 1
        for i in range(end - 1, start - 1, -1):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start += 1
arr = [64, 34, 25, 12, 22, 11, 90]
cocktail_sort(arr)
print("Cocktail Sort Output:", arr)
```

Output:

Cocktail Sort Output: [11, 12, 22, 25, 34, 64, 90]

14. Cycle Sort:

```
def cycle_sort(arr):
    n = len(arr)
    writes = 0
```

```
    for cycleStart in range(n - 1):
        item = arr[cycleStart]
        pos = cycleStart
        for i in range(cycleStart + 1, n):
            if arr[i] < item:
                pos += 1
        if pos == cycleStart:
            continue
        while item == arr[pos]:
            pos += 1
        arr[pos], item = item, arr[pos]
        writes += 1
        while pos != cycleStart:
            pos = cycleStart
            for i in range(cycleStart + 1, n):
                if arr[i] < item:
                    pos += 1
            while item == arr[pos]:
                pos += 1
            arr[pos], item = item, arr[pos]
            writes += 1
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
cycle_sort(arr)
print("Cycle Sort Output:", arr)
```

Output:

Cycle Sort Output: [11, 12, 22, 25, 34, 64, 90]

❖ Research(अनुसंधान):

- अनुसंधान एक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिज़ाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

❖ Innovation(नवीनीकरण): -

- Innovation (इनोवेशन) एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

Note: अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

TWKSAA RID MISSION

(Research)	(Innovation)	(Discovery)
अनुसंधान करने के महत्वपूर्ण कारण:	**नवीनीकरण करने के महत्वपूर्ण कारण:**	**खोज करने के महत्वपूर्ण कारण:**
1. नई ज्ञान की प्राप्ति	1. प्रगति के लिए	1. नए ज्ञान की प्राप्ति
2. समस्याओं का समाधान	2. परिवर्तन के लिए	2. ज्ञान के विकास में योगदान
3. तकनीकी और व्यापार में उन्नति	3. उत्पादन में सुधार	3. अविष्कारों की खोज
4. विकास को बढ़ावा देना	4. प्रतिस्पर्धा में अग्रणी होने के लिए	4. समस्याओं का समाधान
5. सामाजिक प्रगति	5. समाज को लाभ	5. समाज के उन्नति का माध्यम
6. देश विज्ञान और प्रौद्योगिकी का विकास	6. देश विज्ञान और प्रौद्योगिकी के विकास।	6. देश विज्ञान और तकनीक के विकास

➢ जो लोग रिसर्च, इनोवेशन और डिस्कवरी करते हैं उन लोगों को ही हमें अपना नायक, प्रतीक एवं आदर्श मानना चाहिए क्योंकि यें लोग हमारे समाज, देश एवं विज्ञान के क्षेत्र में प्रगति, विकास और समस्याओं के समाधान में महत्वपूर्ण भूमिका निभाते हैं।

मैं राजेश प्रसाद एक वीणा उठाया हूँ Research, Innovation and Discovery का जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले New(RID, PMS & TLR) की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से ही हो |

"अगर आप भी Research, Innovation and Discovery के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें तो हमारे इस त्वक्सा रीड मिशन (TWKSAA RID MISSION) से जरुर जुड़ें" |

- **राजेश प्रसाद**