1. factorial and Fibonacci series using recursion

```
def fact(n): return 1 if n<=1 else n*fact(n-1)
def fib(n): return n if n<=1 else fib(n-1)+fib(n-2)
n=int(input("n:"))
print("Factorial:",fact(n))
print("Fibonacci:",[fib(i) for i in range(n)])
```

output:

```
n:5
Factorial: 120
Fibonacci: [0, 1, 1, 2, 3]
```

Concepts:
Uses recursion — function calls itself.
Base case stops recursion (n <= 1).
Factorial: n! = n * (n-1)!
Fibonacci: F(n) = F(n-1) + F(n-2)
Time complexity of Fibonacci recursion is $O(2^n)$ — expensive for large n

2. analyze time complexity of recursive and iterative approaches.

```
import time
def fact_rec(n):
   return 1 if n==0 else n*fact_rec(n-1)
def fact_itr(n):
   res=1
   for i in range(1,n+1):
      res*=i
   return res
n=500
start=time.time()
fact_rec(n)
rec_time=time.time()-start
start=time.time()
fact_itr(n)
itr_time=time.time()-start
print("Recursive Time:",rec_time)
print("Iterative Time:",itr_time)
print("Recursive: O(n) calls, Iterative: O(n) loop")
```

Concepts:
Recursive and iterative methods give the same result but differ in performance.
Recursive: function calls itself repeatedly (uses stack memory).
Iterative: uses loops (more efficient).
Program measures execution time using time.time().
Typically, iterative is faster and uses less memory.

3. perform insertion, deletion, searching, and traversal in an array.

```
arr=[19.33]
arr.extend([10,20,30])
arr.insert(1,15)
arr.remove(20)  print("Search 30:",30 in arr)
print("Traversal:")
for i in arr: print(i)
```

output:

```
Search 30: True
Traversal:
10 15 19 30 33
```

Concepts:
Demonstrates basic array (list) operations:
Insertion → insert(index, value)
Deletion → remove(value)
Search → value in arr
Traversal → loop through array elements
Python’s list is dynamic, unlike static arrays in C.
Operations like insertion/deletion are O(n) (linear time).

4. to implement Singly Linked List with all operations.

```
class Node:
   def __init__(s,d): s.d=d; s.n=None
class SLL:
   def __init__(s): s.h=None
   def insert(s,d):
      n=Node(d)
      if not s.h: s.h=n; return
      t=s.h
      while t.n: t=t.n
      t.n=n
   def delete(s,k):
      t=s.h
      if not t: return
      if t.d==k: s.h=t.n; return
      while t.n and t.n.d!=k: t=t.n
      if t.n: t.n=t.n.n
   def display(s):
      t=s.h
      while t: print(t.d,end=" "); t=t.n
L=SLL()
L.insert(10);L.insert(20);L.insert(30)
print("List:");L.display();print()
L.delete(20)
print("After Deletion:");L.display()
```

output:

```
List:
10 20 30
After Deletion: 10
10 30
```

Single linked list Concepts:
Each node has data and link (next pointer).
Head points to the first node.
Traversal uses a while loop until None.
Insertion adds new node at the end.
Deletion removes node by linking previous to next.
Efficient for insertion/deletion, but slower for searching (O(n)).

==

5. Program to implement Doubly Linked List.

```
class Node:
   def __init__(s,d): s.d=d; s.p=s.n=None
class DLL:
   def __init__(s): s.h=None
   def insert(s,d):
      n=Node(d)
      if not s.h: s.h=n; return
      t=s.h
      while t.n: t=t.n
      t.n=n; n.p=t
   def delete(s,k):
      t=s.h
      while t and t.d!=k: t=t.n
      if not t: return
      if t.p: t.p.n=t.n
      else: s.h=t.n
      if t.n: t.n.p=t.p
   def display(s):
      t=s.h
      while t: print(t.d,end=" "); t=t.n
L=DLL()
L.insert(10);L.insert(20);L.insert(30)
print("List:");L.display();print()
L.delete(20)
print("After Deletion:");L.display()
```

output:
List:
10 20 30
After Deletion:
10 30

Concepts:
Each node has data, previous, and next pointers.
Allows bidirectional traversal.
Easier deletion (no need to track previous manually).
Extra memory needed for storing previous pointer.
Useful when traversal in both directions is required.

6. implement Circular Linked List.

```
class Node:
   def __init__(s,d): s.d=d; s.n=None
class CLL:
   def __init__(s): s.l=None
   def insert(s,d):
      n=Node(d)
      if not s.l: s.l=n; n.n=n; return
      t=s.l
      while t.n!=s.l: t=t.n
      t.n=n; n.n=s.l
   def display(s):
      if not s.l: return
      t=s.l
      while True:
         print(t.d,end=" ")
         t=t.n
         if t==s.l: break
L=CLL()
L.insert(10);L.insert(20);L.insert(30)
print("Circular Linked List:")
L.display()
```

Output:
Circular Linked List:
10 20 30

Important Points:
Last node points back to the head, forming a loop.
Traversal stops when you come back to the first node.
Useful in round-robin scheduling.

======================================

7. implement Stack using arrays.

```
stack=[]
stack.append(10)
stack.append(20)
stack.append(30)
print("Stack:",stack)
stack.pop()
print("After Pop:",stack)
```

output:
Stack: [10, 20, 30]
After Pop: [10, 20]

Important Points:
LIFO (Last In First Out) principle.
Push → append(), Pop → pop().
Simple and efficient using Python lists.

8.implement Stack using linked list

```
class Node:
   def __init__(s,d): s.d=d; s.n=None
class Stack:
   def __init__(s): s.t=None
   def push(s,d):
      n=Node(d); n.n=s.t; s.t=n
   def pop(s):
      if not s.t: return
      s.t=s.t.n
   def display(s):
      t=s.t
      while t: print(t.d,end=" "); t=t.n
S=Stack()
S.push(10);S.push(20);S.push(30)
print("Stack:")
S.display();print()
S.pop()
print("After Pop:")
S.display()
```

Output:

```
Stack:
30 20 10
After Pop:
20 10
```

Important Points:
Uses linked nodes instead of arrays.
Top pointer (t) stores latest element.
Avoids overflow (dynamic memory)..

11.implement Queue using arrays.

```
q=[]
q.append(10)
q.append(20)
q.append(30)
print("Queue:",q)
q.pop(0)
print("After Dequeue:",q)
```

Output:

```
Queue: [10, 20, 30]
After Dequeue: [20, 30]
```

Important Points:
FIFO (First In First Out) structure.
Enqueue → append(), Dequeue → pop(0).
Simple but not memory efficient (shifts elements).

12. implement Circular Queue

```
size=3
q=[None]*size
f=r=-1
def enqueue(v):
   global f,r
   if (r+1)%size==f: return
   if f==-1: f=0
   r=(r+1)%size; q[r]=v
def dequeue():
   global f,r
   if f==-1: return
   if f==r: f=r=-1
   else: f=(f+1)%size
enqueue(10);enqueue(20);enqueue(30)
print("Queue:",q)
dequeue()
enqueue(40)
print("After Enqueue:",q)
```

output:

```
Queue: [10, 20, 30]
After Enqueue: [40, 20, 30]
```

important Points:
Uses modulus (%) for circular connection.
Avoids wasted space at the front.
Efficient for fixed-size queues.

13. implement Queue using linked list.

```
class Node:
   def __init__(s,d): s.d=d; s.n=None
class Queue:
   def __init__(s): s.f=s.r=None
   def enqueue(s,d):
      n=Node(d)
      if not s.r: s.f=s.r=n; return
      s.r.n=n; s.r=n
   def dequeue(s):
      if not s.f: return
      s.f=s.f.n
   def display(s):
      t=s.f
      while t: print(t.d,end=" "); t=t.n
Q=Queue()
Q.enqueue(10);Q.enqueue(20);Q.enqueue(30)
print("Queue:")
Q.display();print()
Q.dequeue()
print("After Dequeue:")
Q.display()
```

Output:

Queue:
10 20 30
After Dequeue:
20 30

Important Points:

Uses front and rear pointers.
Efficient enqueue/dequeue in O(1) time.
Dynamic size (no overflow).

---

14. Binary Tree using Linked Representation

```
class Node:
   def __init__(s,d): s.d=d; s.l=s.r=None
r=Node(1)
r.l=Node(2)
r.r=Node(3)
r.l.l=Node(4)
r.l.r=Node(5)
def inorder(n):
   if n:
      inorder(n.l)
      print(n.d,end=" ")
      inorder(n.r)
print("Inorder Traversal:")
inorder(r)
```

Output:

Inorder Traversal:
4 2 5 1 3

Important Points:

Each node stores data, left, and right references.
Linked representation → dynamic memory allocation.
Useful base for tree traversals and BST.

---

15. Inorder, Preorder, and Postorder Traversals

```
class Node:
   def __init__(s,d): s.d=d; s.l=s.r=None
r=Node(1)
r.l=Node(2)
r.r=Node(3)
r.l.l=Node(4)
r.l.r=Node(5)
def inorder(n):
   if n: inorder(n.l);print(n.d,end=" ");inorder(n.r)
def preorder(n):
   if n: print(n.d,end="
");preorder(n.l);preorder(n.r)
def postorder(n):
   if n: postorder(n.l);postorder(n.r);print(n.d,end="
")
print("Inorder:");inorder(r)
print("\nPreorder:");preorder(r)
print("\nPostorder:");postorder(r)
```

Output:

Inorder:
4 2 5 1 3
Preorder:
1 2 4 5 3
Postorder:
4 5 2 3 1

Important Points:

Inorder (LNR): Left → Node → Right
Preorder (NLR): Node → Left → Right
Postorder (LRN): Left → Right → Node
Recursive traversal is simple and clear.

16. Binary Search Tree (Insert & Search)

```
class Node:
   def __init__(s,d): s.d=d; s.l=s.r=None
def insert(r,v):
   if not r: return Node(v)
   if v<r.d: r.l=insert(r.l,v)
   else: r.r=insert(r.r,v)
   return r
def search(r,v):
   if not r or r.d==v: return r
   return search(r.l,v) if v<r.d else search(r.r,v)
def inorder(r):
   if r: inorder(r.l);print(r.d,end=" ");inorder(r.r)
r=None
for x in [50,30,70,20,40,60,80]: r=insert(r,x)
print("Inorder:");inorder(r)
print("\nSearch 60:", "Found" if search(r,60) else
"Not Found")
```

Output:

Inorder:

20 30 40 50 60 70 80

Search 60: Found

Important Points:

Left < Root < Right property.

Insertion and searching both O(h) (height of tree).

Inorder traversal gives sorted order.

Program to delete a node from a BST.

```
class Node:
   def __init__(self,key):
      self.key=key
      self.left=None
      self.right=None
def insert(root,key):
   if not root:
      return Node(key)
   if key<root.key:
      root.left=insert(root.left,key)
   else:
      root.right=insert(root.right,key)
   return root
def minValueNode(node):
   curr=node
   while curr.left:
      curr=curr.left
   return curr
def deleteNode(root,key):
   if not root:
      return root
   if key<root.key:
      root.left=deleteNode(root.left,key)
elif key>root.key:
   root.right=deleteNode(root.right,key)
else:
      if not root.left:
         return root.right
      elif not root.right:
         return root.left
      temp=minValueNode(root.right)
      root.key=temp.key
      root.right=deleteNode(root.right,temp.key)
   return root
def inorder(root):
   if root:
      inorder(root.left)
      print(root.key,end=" ")
      inorder(root.right)
root=None
for x in [50,30,20,40,70,60,80]:
   root=insert(root,x)
print("Inorder before deletion:")
inorder(root)
root=deleteNode(root,50)
print("\nInorder after deleting 50:")
inorder(root)
```

Inorder before deletion:

20 30 40 50 60 70 80

Inorder after deleting 50:

20 30 40 60 70 80

Important Points:

Handles three cases:

Node with no child → delete directly

Node with one child → link to child

Node with two children → replace with inorder successor

Maintains BST property after deletion.

---

18. Priority Queue using Heap

```
import heapq
pq=[]
heapq.heappush(pq,3)
heapq.heappush(pq,1)
heapq.heappush(pq,2)
print("Min-Heap:",pq)
print("Deleted:",heapq.heappop(pq))
print("After Deletion:",pq)
```

Output:

Min-Heap: [1, 3, 2]

Deleted: 1

After Deletion: [2, 3]

Important Points:

Implemented using heapq (min-heap by default).

Fast O(log n) insertion & deletion.

Always returns smallest (highest priority) element first.

Used in schedulers, Dijkstra’s, task queues.

1. Factorial and Fibonacci using Recursion
Recursion is a method where a function calls itself until a base condition is met.
Recursion uses function call stack; every call adds a new frame.
Advantage: simple and clear logic.
Disadvantage: consumes more memory and time for large inputs.
Applications: Divide and Conquer algorithms, sorting (Quick sort, Merge sort), and dynamic programming.

2. Time Complexity of Recursive and Iterative Approaches
Time complexity measures how fast an algorithm executes based on input size n.
Recursive approach: divides the problem into smaller subproblems.
Iterative approach: uses loops and variables for repetition.
Recursive methods use extra memory (stack frames) and may be slower due to repeated calls.
Iterative methods are faster and memory efficient.
Example: Fibonacci recursion has $O(2^n)$, iterative has $O(n)$.
Used to choose optimal solutions when designing efficient algorithms.

3Array Operations (Insertion, Deletion, Searching, Traversal)
An array stores elements of the same type in contiguous memory locations.
Insertion: adding an element at any index — requires shifting elements.
Deletion: removing an element — also requires shifting.
Searching:
Linear search — O(n)
Binary search — O(log n) for sorted arrays.
Traversal: accessing all elements sequentially.
Advantage: fast random access (O(1)).
Disadvantage: fixed size, expensive insertion/deletion.

4.Singly Linked List
A linked list is a dynamic structure made of nodes, each containing data and a pointer to the next node.
Singly linked list connects nodes in one direction.
Operations:
Insertion (beginning, end, position)
Deletion
Searching
Traversal
Advantages: dynamic memory allocation, efficient insertion/deletion.
Disadvantages: no backward traversal, more memory for pointers.

5. Doubly Linked List
Each node has three fields: data, next pointer, and previous pointer.
Allows traversal in both directions.
Easier deletion since the previous node is easily accessible.
Operations: insertion, deletion, searching, traversal (forward/backward).
Uses more memory but improves flexibility.
Example: browser history navigation, music playlists.

6.Circular Linked List
The last node points back to the first node, forming a circle.
Can be singly or doubly linked.
Traversal starts at any node and continues until it loops back.
Used in round-robin scheduling, buffering, and queue implementations.
Advantage: continuous traversal, no NULL at end.
Disadvantage: more complex insertion/deletion logic.

7.Stack using Arrays
A stack is a LIFO (Last In, First Out) data structure.
Operations:
Push: insert element at top.
Pop: remove top element.
Peek: view top element.
Implemented using arrays with a top pointer.
Overflow occurs when stack is full; underflow when empty.
Used in expression evaluation, backtracking, recursion, undo operations.

8.Stack using Linked List
Each node contains data and a pointer to the next node.
The top points to the most recent element.
Push: insert node at top; Pop: remove top node.
Dynamic size — no overflow unless memory full.
More memory due to pointers.
Suitable when size of data is unknown.

11.Queue using Arrays
A queue is FIFO (First In, First Out).
Elements are inserted at rear and removed from front.
Implemented using arrays with front and rear pointers.
When rear reaches end, can't insert even if front has moved → solved using circular queue.
Used in task scheduling, printers, call centers.

12.Circular Queue
The last position connects to the first position to make it circular.
Uses modulo operation % to wrap around indices.
Prevents unused space in array implementation.
Efficient for fixed-size memory.
Used in buffering and real-time systems.

13.Queue using Linked List
Implemented with nodes having data and next pointer.
Front points to first node; Rear points to last.
Enqueue: insert at rear.
Dequeue: delete from front.
Dynamic in size, no overflow.
Used in dynamic memory management and CPU scheduling.

14.Binary Tree using Linked Representation
A binary tree is a non-linear structure where each node has at most two children.
Each node contains data, left pointer, and right pointer.
The root is the topmost node.
Used in hierarchical data storage.
Basis for advanced trees like BST, AVL, and heaps.
Applications: expression trees, parsing, searching.

15. Inorder, Preorder, and Postorder Traversals
Traversal means visiting all nodes in a tree.
Inorder (LNR): Left → Node → Right
For BST, gives sorted order.
Preorder (NLR): Node → Left → Right
Used to copy or create trees.
Postorder (LRN): Left → Right → Node
Used to delete or free memory.
Traversal can be done recursively or iteratively using stack.

16. Binary Search Tree (BST) with Insert and Search
A BST is a binary tree where:
Left child < parent node
Right child > parent node
Insertion: place new node at correct position following order rule.
Search: traverse left or right depending on value.
Time complexity:
Best/Average: O(log n)
Worst (skewed tree): O(n).
Used for fast searching and sorting.

17.Deletion in BST
Deletion depends on node type:
Leaf node: delete directly.
One child: connect parent to child.
Two children: find inorder successor or predecessor and replace value.
Maintains BST property after deletion.
Balancing helps keep time complexity efficient.
Used in database indexing and symbol tables.

18.Priority Queue using Heap
A priority queue assigns each element a priority.
Implemented using a heap (usually a binary heap)
Min-heap: smallest value at root.
Max-heap: largest value at root.
Operations:
Insert: add at end, then heapify-up.
Delete: remove root, replace with last element, then heapify-down.
Time complexity: O(log n).
Used in Dijkstra's algorithm, CPU scheduling, and simulation systems.