**Data Structures -Python**

Way to store and organize the data so that it can be used effectively.

**Built-In** -List, tuple, set, dictionary

**User defined**- stack, queue, linked list, tree, graph

**List- mutable and ordered**

append() Adds an element at the end of the list

clear() Removes all the elements from the list

copy() Returns a copy of the list

count() Returns the number of elements with the specified value

extend() Add the elements of a list to the end of the current list

index() Returns the index of the first element with the specified value

insert() Adds an element at the specified position

pop() Removes the element at the specified position

remove() Removes the item with the specified value

reverse() Reverses the order of the list

sort() Sorts the list

**Tuple-ordered, immutable**

count() Returns the number of times a specified value occurs in a tuple

index() Searches the tuple for a specified value and returns the position of where it was found

Concatenation: concatenate two or more tuples to create a new tuple using the + operator.

**Dictionary- ordered, mutable**

keys-immutable only, values-mutable or immutable

clear() Removes all the elements from the dictionary

copy() Returns a copy of the dictionary

get() Returns the value of the specified key

items() Returns a list containing a tuple for each key value pair

keys() Returns a list containing the dictionary's keys

pop() Removes the element with the specified key

popitem() Removes the last inserted key-value pair

update() Updates the dictionary with the specified key-value pairs

values() Returns a list of all the values in the dictionary

**Sets-No duplicate elements, unordered, mutable,**

immutable objects as element only

set cannot be nested

loop, in, not in used for accessing the elements in set

union=|, intersection=&, difference=-, symmetric difference=^

add() Adds an element to the set

clear() Removes all the elements from the set

copy() Returns a copy of the set

difference() Returns a set containing the difference between two or more sets

difference\_update() Removes the items in this set that are also included in another, specified set

discard() Remove the specified item

intersection() Returns a set, that is the intersection of two other sets

intersection\_update() Removes the items in this set that are not present in other, specified set(s)

pop() Removes an element from the set

remove() Removes the specified element

symmetric\_difference() Returns a set with the symmetric differences of two sets

symmetric\_difference\_update() inserts the symmetric differences from this set and another

union() Return a set containing the union of sets

update()Update the set with the union of this set and others

**Stack-Last in first out, first in last out**

**Application**s: Reverse a string, Expression evaluation(infix to postfix,…)**,**forward and backward features in browser,

**Implement stack using list**- push-append, pop-pop

stack=[]

def push():

if len(stack)==n:

print("stack full")

else:

ele=int(input())

stack.append(ele)

print(stack)

def pop():

if len(stack)==0:

print("stack empty")

else:

e=stack.pop()

print("Removed element",e)

print(stack)

print("Enter the length of stack")

n=int(input())

while True:

print("Choose operation 1.push 2.pop 3.quit")

ch=int(input())

if ch==1:

push()

elif ch==2:

pop()

elif ch==3:

break

else:

print("Enter correct operation")

**Implement stack using deque(Double ended queue) push-append, pop-pop**

import collections

stack=collections.deque()

def push():

if len(stack)==n:

print("stack full")

else:

ele=int(input())

stack.append(ele)

print(stack)

def pop():

if len(stack)==0:

print("stack empty")

else:

e=stack.pop()

print("Removed element",e)

print(stack)

print("Enter the length of stack")

n=int(input())

while True:

print("Choose operation 1.push 2.pop 3.quit")

ch=int(input())

if ch==1:

push()

elif ch==2:

pop()

elif ch==3:

break

else:

print("Enter correct operation")

**Implement stack using lifoqueue push-put, pop- get**

import queue

stack=queue.LifoQueue()

def push():

if stack.qsize()==n:

print("stack full")

else:

ele=int(input())

stack.put(ele)

def pop():

if stack.qsize()==0:

print("stack empty")

else:

print("Removed element",stack.get())

print("Enter the length of stack")

n=int(input())

while True:

print("Choose operation 1.push 2.pop 3.quit")

ch=int(input())

if ch==1:

push()

elif ch==2:

pop()

elif ch==3:

break

else:

print("Enter correct operation")

**Queue- First in first out, last in last out, enqueue , dequeue, isfull, isempty**

**Queue implementation using list-Enqueue- append, Dequeue- pop(0)**

queue=[]

def enqueue():

if len(queue)==n:

print("queue full")

else:

ele=int(input())

queue.append(ele)

def dequeue():

if len(queue)==0:

print("queue empty")

else:

print("Removed element",queue.pop(0))

print("Enter the length of queue")

n=int(input())

while True:

print("Choose operation 1.push 2.pop 3.quit")

ch=int(input())

if ch==1:

enqueue()

elif ch==2:

dequeue()

elif ch==3:

break

else:

print("Enter correct operation")

**Queue implementation using list-Enqueue- insert, Dequeue- pop()**

queue=[]

def enqueue():

if len(queue)==n:

print("queue full")

else:

ele=int(input())

queue.insert(0,ele)

print(queue)

def dequeue():

if len(queue)==0:

print("queue empty")

else:

print("Removed element",queue.pop())

print(queue)

print("Enter the length of queue")

n=int(input())

while True:

print("Choose operation 1.push 2.pop 3.quit")

ch=int(input())

if ch==1:

enqueue()

elif ch==2:

dequeue()

elif ch==3:

break

else:

print("Enter correct operation")

**Implement queue using deque(Double ended queue) push-appendleft , pop-pop**

import collections

queue=collections.deque()

def enqueue():

if len(queue)==n:

print("queue full")

else:

ele=int(input())

queue.appendleft(ele)

print(queue)

def dequeue():

if len(queue)==0:

print("queue empty")

else:

print("Removed element",queue.pop())

print(queue)

print("Enter the length of queue")

n=int(input())

while True:

print("Choose operation 1.push 2.pop 3.quit")

ch=int(input())

if ch==1:

enqueue()

elif ch==2:

dequeue()

elif ch==3:

break

else:

print("Enter correct operation")

**Implement queue using deque(Double ended queue) push-append , pop-popleft**

import collections

queue=collections.deque()

def enqueue():

if len(queue)==n:

print("queue full")

else:

ele=int(input())

queue.append(ele)

print(queue)

def dequeue():

if len(queue)==0:

print("queue empty")

else:

print("Removed element",queue.popleft())

print(queue)

print("Enter the length of queue")

n=int(input())

while True:

print("Choose operation 1.push 2.pop 3.quit")

ch=int(input())

if ch==1:

enqueue()

elif ch==2:

dequeue()

elif ch==3:

break

else:

print("Enter correct operation")

**Implement queue using queue push-put , pop- get**

import queue

queue=queue.Queue()

def push():

if queue.qsize()==n:

print("queue full")

else:

ele=int(input())

queue.put(ele)

def pop():

if queue.qsize()==0:

print("queue empty")

else:

print("Removed element",queue.get())

print("Enter the length of queue")

n=int(input())

while True:

print("Choose operation 1.push 2.pop 3.quit")

ch=int(input())

if ch==1:

push()

elif ch==2:

pop()

elif ch==3:

break

else:

print("Enter correct operation")

**Implement Priority Queue using list (lowest number as high priority)**

pq=[]

pq.append(1000)

pq.append(100)

pq.append(5)

pq.sort()

print(pq) //5,100,1000

print(pq.pop(0)) //5

**Implement Priority Queue using list (highest number as high priority)**

pq=[]

pq.append(1000)

pq.append(100)

pq.append(5)

pq.sort(reverse=True)

print(pq) //1000,100,5

print(pq.pop(0)) //1000

**Implement Priority Queue using classes. (It will take lowest value as high priority)**

import queue

pq=queue.PriorityQueue()

pq.put(1)

pq.put(3)

pq.put(0)

print(pq.get())

**Implement Priority Queue using list object as tuple (It will take highest value as high priority)**

pq=[]

pq.append((1,"gokul"))

pq.append((4,"ram"))

pq.append((2,"raju"))

pq.sort(reverse=True)

print(pq.pop(0))

**Implement Priority Queue using list object as tuple (It will take lowest value as high priority)**

pq=[]

pq.append((1,"gokul"))

pq.append((4,"ram"))

pq.append((2,"raju"))

pq.sort()

print(pq.pop(0))

**Linked List**

- chain of nodes

**Advantages:**

-dynamic data structure

-insertion and deletion in easy

-Implement stack, queue ,graph

-Web browser, music player, image viewer

**Disadvantages**

**-** Needs extra memory

-Random access not possible

**Applications:**

Treasure hunt game, Relay run

**Singly linked list:**

**Traversing singly linked list:**

def print\_ll(self):

if self.head is None:

print("Linked list is empty")

else:

n=self.head

while n is not None:

print(n.data)

n=n.ref

**Insertion at beginning:**

class Node:

def \_\_init\_\_(self,data):

self.data=data

self.ref=None

class LinkedList:

def \_\_init\_\_(self):

self.head=None

def print\_ll(self):

if self.head is None:

print("Linked list is empty")

else:

n=self.head

while n is not None:

print(n.data)

n=n.ref

def add\_begin(self,data):

new\_node=Node(data)

new\_node.ref=self.head

self.head=new\_node

ll=LinkedList()

ll.add\_begin(10)

ll.add\_begin(20)

ll.print\_ll()

**Insertion at end:**

class Node:

def \_\_init\_\_(self,data):

self.data=data

self.ref=None

class LinkedList:

def \_\_init\_\_(self):

self.head=None

def print\_ll(self):

if self.head is None:

print("Linked list is empty")

else:

n=self.head

while n is not None:

print(n.data)

n=n.ref

def add\_end(self,data):

new\_node=Node(data)

if self.head is None:

self.head=new\_node

else:

t=self.head

while t.ref is not None:

t=t.ref

t.ref=new\_node

ll=LinkedList()

ll.add\_end(10)

ll.add\_end(20)

ll.print\_ll()

**Insertion after the particular data:**

def add\_after(self,data,x):

nt=self.head

while nt is not None:

if nt.data==x:

break

nt=nt.ref

if nt is None:

print("Element is not found in ll")

else:

new\_node=Node(data)

new\_node.ref=nt.ref

nt.ref=new\_node

**Insertion before the particular data:**

def add\_before(self,data,x):

if self.head is None:

print("ll is empty")

if self.head.data==x:

new\_node=Node(data)

new\_node.ref=self.head

self.head=new\_node

return

f=self.head

while f.ref is not None:

if f.ref.data==x:

break

f=f.ref

if f.ref is None:

print("Element not found in ll")

else:

new\_node=Node(data)

new\_node.ref=f.ref

f.ref=new\_node

**Insertion when singly linked list is empty**

def insert\_empty(self,data):

if self.head is None:

new\_node=Node(data)

self.head=new\_node

else:

print("LL is not empty")

**Deletion at beg:**

def delete\_beg(self):

if self.head is None:

print("ll is empty")

else:

self.head=self.head.ref

**Deletion at end:**

def delete\_end(self):

if self.head is None:

print("ll is empty")

else:

e=self.head

while e.ref.ref is not None:

e=e.ref

e.ref=None

**Deletion at any position:**

def delete\_at\_any\_position(self,x):

if self.head is None:

print("ll is empty")

return

if self.head.data==x:

self.head=self.head.ref

return

p=self.head

while p.ref is not None:

if p.ref.data==x:

break

p=p.ref

if p.ref is None:

print("element not found in ll")

else:

p.ref=p.ref.ref

**Doubly linked list:**

Easy to move forward or backward -advantages

Disadavantages- extra memory to store previous link

**Traversing doubly linked list:**

class Node:

def \_\_init\_\_(self,data):

self.data=data

self.pref=None

self.nref=None

class LinkedList:

def \_\_init\_\_(self):

self.head=None

def print\_dll(self):

if self.head is None:

print("Linked list is empty")

else:

n=self.head

while n is not None:

print(n.data)

n=n.nref

def print\_rev\_dll(self):

if self.head is None:

print("Linked list is empty")

else:

n=self.head

while n.nref is not None:

n=n.nref

while n is not None:

print(n.data)

n=n.pref

dll=LinkedList()

dll.print\_dll()

dll.print\_rev\_dll()

**Insertion when doubly linked list is empty**

def insert\_empty(self,data):

if self.head is None:

new\_node=Node(data)

self.head=new\_node

else:

print("LL is not empty")

**Insertion at front :**

def insert\_beg(self,data):

new\_node=Node(data)

if self.head is None:

self.head=new\_node

else:

new\_node.nref=self.head

self.head=new\_node

**Insertion at end:**

def insert\_end(self,data):

new\_node=Node(data)

if self.head is None:

self.head=new\_node

else:

t=self.head

while t.nref is not None:

t=t.nref

t.nref=new\_node

new\_node.pref=t

**Insertion after particular data:**

def insert\_after(self,data,x):

if self.head is None:

print("dll. is empty")

else:

p=self.head

while p is not None:

if p.data==x:

break

p=p.nref

if p is None:

print("Element not found in dll")

else:

new\_node=Node(data)

new\_node.nref=p.nref

new\_node.pref=p

if p.nref is not None:

p.nref.pref=new\_node

p.nref=new\_node

**Insertion before particular data:**

def insert\_before(self,data,x):

if self.head is None:

print("dll is empty")

else:

f=self.head

while f is not None:

if f.data==x:

break

f=f.nref

if f is None:

print("Element not found in dll")

else:

new\_node=Node(data)

new\_node.nref=f

new\_node.pref=f.pref

if f.pref is not None:

f.pref.nref=new\_node

else:

self.head=new\_node

f.pref=new\_node

**Deletion at front:**

def delete\_beg(self):

if self.head is None:

print("dll is empty")

return

if self.head.nref is None:

self.head=None

print("dll is empty after deleting the node")

else:

self.head=self.head.nref

self.head.pref=None

**Deletion at end:**

def delete\_end(self):

if self.head is None:

print("dll is empty")

return

if self.head.nref is None:

self.head=None

print("dll is empty after deleting the node")

else:

k=self.head

while k.nref is not None:

k=k.nref

k.pref.nref=None

**Deletion by value:**

def delete\_value(self,x):

if self.head is None:

print("dll is empty")

return

if self.head.nref is None:

if self.head.data==x:

self.head=None

else:

print("element not found in dll")

return

if self.head.data==x:

self.head=self.head.nref

self.head.pref=None

return

q=self.head

while q.nref is not None:

if q.data==x:

break

q=q.nref

if q.nref is not None:

q.pref.nref=q.nref

q.nref.pref=q.pref

else:

if q.data==x:

q.pref.nref=None

else:

print("Element not found in dll")

**Circular linked list: -1 linked list**

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.ref = None

class CircularLinkedList:

def \_\_init\_\_(self):

self.head = None

def print\_ll(self):

if self.head is None:

print("Circular Linked list is empty")

return

current = self.head

while True:

print(current.data, end=" -> ")

current = current.ref

if current == self.head:

break

print()

def add\_begin(self, data):

new\_node = Node(data)

if not self.head:

new\_node.ref = new\_node # Make it circular if the list is empty

else:

new\_node.ref = self.head

current = self.head

while current.ref != self.head:

current = current.ref

current.ref = new\_node

self.head = new\_node

def add\_end(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

new\_node.ref = new\_node # Make it circular if the list is empty

else:

current = self.head

while current.ref != self.head:

current = current.ref

current.ref = new\_node

new\_node.ref = self.head

def add\_after(self, data, x):

if self.head is None:

print("Circular Linked list is empty")

return

current = self.head

while True:

if current.data == x:

break

current = current.ref

if current == self.head:

print("Element is not found in the circular linked list")

return

new\_node = Node(data)

new\_node.ref = current.ref

current.ref = new\_node

def add\_before(self, data, x):

if self.head is None:

print("Circular Linked list is empty")

return

if self.head.data == x:

new\_node = Node(data)

new\_node.ref = self.head

current = self.head

while current.ref != self.head:

current = current.ref

current.ref = new\_node

self.head = new\_node

return

current = self.head

while current.ref != self.head:

if current.ref.data == x:

break

current = current.ref

if current.ref == self.head:

print("Element is not found in the circular linked list")

return

new\_node = Node(data)

new\_node.ref = current.ref

current.ref = new\_node

def insert\_empty(self, data):

if self.head is None:

new\_node = Node(data)

self.head = new\_node

new\_node.ref = new\_node # Make it circular if the list is empty

else:

print("Circular Linked list is not empty")

def delete\_beg(self):

if self.head is None:

print("Circular Linked list is empty")

else:

current = self.head

while current.ref != self.head:

current = current.ref

current.ref = self.head.ref

self.head = self.head.ref

def delete\_end(self):

if self.head is None:

print("Circular Linked list is empty")

else:

current = self.head

while current.ref.ref != self.head:

current = current.ref

current.ref = self.head

def delete\_at\_any\_position(self, x):

if self.head is None:

print("Circular Linked list is empty")

return

if self.head.data == x:

current = self.head

while current.ref != self.head:

current = current.ref

current.ref = self.head.ref

self.head = self.head.ref

return

current = self.head

while current.ref != self.head:

if current.ref.data == x:

break

current = current.ref

if current.ref == self.head:

print("Element is not found in the circular linked list")

return

current.ref = current.ref.ref

cll = CircularLinkedList()

cll.add\_begin(10)

cll.add\_end(20)

cll.add\_after(30, 20)

cll.add\_before(5, 10)

cll.insert\_empty(10)

cll.insert\_empty(20)

cll.print\_ll()

cll.delete\_beg()

cll.delete\_end()

cll.delete\_at\_any\_position(10)

cll.print\_ll()

**circular linked list- 2 linked list**

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.pref = None

self.nref = None

class CircularDoublyLinkedList:

def \_\_init\_\_(self):

self.head = None

def print\_cdll(self):

if self.head is None:

print("Circular Doubly Linked list is empty")

return

current = self.head

while True:

print(current.data, end=" -> ")

current = current.nref

if current == self.head:

break

print()

def print\_rev\_cdll(self):

if self.head is None:

print("Circular Doubly Linked list is empty")

return

current = self.head

while current.nref != self.head:

current = current.nref

while True:

print(current.data, end=" -> ")

current = current.pref

if current == self.head:

break

print()

def insert\_beg(self, data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

new\_node.nref = new\_node

new\_node.pref = new\_node

else:

new\_node.nref = self.head

new\_node.pref = self.head.pref

self.head.pref.nref = new\_node

self.head.pref = new\_node

self.head = new\_node

def insert\_end(self, data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

new\_node.nref = new\_node

new\_node.pref = new\_node

else:

new\_node.nref = self.head

new\_node.pref = self.head.pref

self.head.pref.nref = new\_node

self.head.pref = new\_node

def insert\_after(self, data, x):

if self.head is None:

print("Circular Doubly Linked list is empty")

return

current = self.head

while True:

if current.data == x:

break

current = current.nref

if current == self.head:

print(f"Element {x} is not found in the circular doubly linked list")

return

new\_node = Node(data)

new\_node.nref = current.nref

new\_node.pref = current

current.nref.pref = new\_node

current.nref = new\_node

def insert\_before(self, data, x):

if self.head is None:

print("Circular Doubly Linked list is empty")

return

current = self.head

while True:

if current.data == x:

break

current = current.nref

if current == self.head:

print(f"Element {x} is not found in the circular doubly linked list")

return

new\_node = Node(data)

new\_node.nref = current

new\_node.pref = current.pref

current.pref.nref = new\_node

current.pref = new\_node

def delete\_beg(self):

if self.head is None:

print("Circular Linked list is empty")

return

if self.head.nref == self.head: # If there is only one node in the list

self.head = None

else:

current = self.head

while current.nref != self.head:

current = current.nref

current.nref = self.head.nref

self.head = self.head.nref

self.head.pref = current

def delete\_end(self):

if self.head is None:

print("Circular Linked list is empty")

return

if self.head.nref == self.head: # If there is only one node in the list

self.head = None

else:

current = self.head

while current.nref.nref != self.head:

current = current.nref

current.nref = self.head

self.head.pref = current

def delete\_value(self, x):

if self.head is None:

print("Circular Doubly Linked list is empty")

return

current = self.head

while True:

if current.data == x:

break

current = current.nref

if current == self.head:

print(f"Element {x} is not found in the circular doubly linked list")

return

current.pref.nref = current.nref

current.nref.pref = current.pref

if current == self.head:

self.head = current.nref

cdll = CircularDoublyLinkedList()

cdll.insert\_beg(10)

cdll.insert\_beg(1)

cdll.insert\_end(30)

cdll.insert\_after(15, 30)

cdll.insert\_before(5, 30)

cdll.print\_cdll()

cdll.delete\_beg()

cdll.delete\_end()

cdll.delete\_value(23)

cdll.print\_cdll()

cdll.print\_rev\_cdll()

**Linear data structure:**

Statck,queue,linked list

**Non linear data structure:**

Tree,graph

**Tree**

**Applications:** Family tree, organization structure

Represents relationship between nodes.

Collection of entities called nodes.

Nodes are connected by edges.

**Node**- individual element of tree

**Root**- topmost node of the tree, origin of the data structure, a tree can contain only one root node

**Edge/link**- connection between two nodes

**Parent node** - node which has child

**Child node**- a node which has its link from parent node. In a tree all the nodes except root is child node.

**Siblings**: nodes which belong to same parents

**Leaf node/External node/Terminal node**: node which does not have a child .

**Internal node/Non terminal node** -node which has atleast 1 child, other than leaf node all other are internal nodes

**Path:** sequence of nodes and edges from 1 node to another node is called path between 2 nodes.

In a tree if we have **N** nodes then we will have **N-1** edges

Every child will have only one parent but parent can have multiple child

Tree is a recursive data structure.

Degree of a node -total number of children of that node.

Degree of tree- a node which contains highest degree which is degree of tree.

Level – In a tree, each step from top to bottom is called level of tree. starts from 0.

Height of node – total no of edges that lies on the longest path from any leave node to particular node.

Height of tree- height of root node.

Depth of a node- total no of edges from root node to particular node.

Height of leaf node- 0

Depth of root node -0

**General tree:** each node can have any no of child nodes. There are no restriction or limitation of how many child node each node can have.

**Binary tree:** each node have atmost 2 children. There are limitation of how many child node each node can have. each node can have 0,1,2 not more than 2.

**Types of binary tree:**

Full binary tree, complete binary tree, perfect binary tree, balanced binary tree, pathological binary tree.