Date: ...../...../.....

1) Linked List:

Linked list elements are not stored at a contiguous locations.

* The elements are linked using pointers.

Why:

Arrays can be used to store linear data of similar types, but arrays have some limitations

1) The size of array is fixed

2) Insertion of a new element/ deletion of a existing element in array is expensive

Advantages over arrays:

1) Dynamic array
2) Ease of insertion / deletion.

Drawbacks:

1) Random access is not allowed.

* we have to access elements sequentially starting from the first node (head node).

* So, we cannot do binary search with linked list efficiently.

2) Extra memory space for pointer is required
3) Not cache friendly.

Date: ...../...../.....

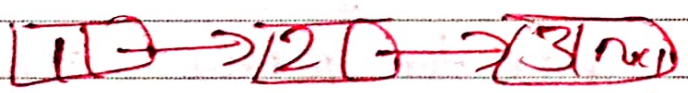
Linked List:

Representation:

(1)

Ex:

```
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    def __init__(self):
        self.head = None

if __name__ == "__main__":
    llist = LinkedList()
    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

    llist.head.next = second;
    second.next = third;
```



(1)

Date: ...../...../.....

② Traversal

Add print list function to above at (1)

(1)
```
def printList(self):
    temp = self.head
    while(temp):
        print(temp.data)
        temp = temp.next
```

Time Complexity

	worst	average
1) search	O(n)	O(n)
2) Insert	O(1)	O(1)
3) Delete	O(1)	O(1)

Auxiliary space = O(n)

Date: ...../...../.....

* 1) what is __init__ in Python?

It is a constructor used to initializing the object's state.

*Reserved method in python classes

* 2) what is self in python?

The keyword self represents the instance of class and binds the attributes with given arguments.

3) Lambda:

It is a small anonymous function.

* can take any number of arguments, but can only have one expression.

* x = lambda a,: a+10
x = lambda a,b: a*b } expressions.
↓
arguments

Date: ...../...../.....

④ super() : used to give access to methods and properties of a parent (or) sibling class

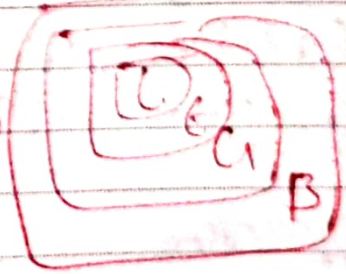
⑤	init	constructor
	init method is invoked when servlet is called and it adds values to servlet context.	constructor is called by container simply to create a POJO (plain old java object)

⑥ Name space :

A collection of currently defined symbolic names along with information about the object that each name references

Types
1) Built-in
2) Global
3) Enclosing
4) Local

⑦ Scope : LEGB



Local
Enclosing
Global
Built-in

Date: ...../...../.....

(8) Decorators:

It is a function that takes another function as its argument, and returns yet another function.

(9) map():

map(func, *iterables)

* map() function returns a map object (iterable) of the results after applying the given function to each item of a given iterable.

(10) filter():

filter(fnc, sequence)

It is used filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Applications:

* generally used with Lambda functions

Date: ...../...../.....

(11) *args and **kwargs

*args - Non keyword Arguments
**kwargs - keyword Arguments

we use these as an argument when we are unsure about no. of arguments to pass in functions.

(12) Pickling & Unpickling :

*pickling - It is the process where by a Python (object) hierarchy is converted into byte stream

*unpickling: Byte Stream is converted back into an object hierarchy

Date: ...../...../.....

(13) packing and unpacking:

unpack * (for tuples)
** (for dictionaries)

↓

L2 [1,2,3,4]

*L = (1,2,3,4) — Tuple

pack * used when we don't know how many arguments needed to be passed to a python function.

(14) 4 pillars - 1) Abstraction
2) encapsulation
3) Inheritance
4) Polymorphism

Consider 3 variables
Lo=0, mid, high

Date: ...../...../.....

(0's, 1's, 2's)

Sort the ARRAY IN Ascending order.

① 'Dutch National Flag problem'

```
def sort012(a,arr_size):

    lo=0
    hi=arr_size-1
    mid=0
    while mid<=hi:
        if a[mid]==0:
            a[lo],a[mid]=a[mid],a[lo]
            lo=lo+1
            mid=mid+1

        elif a[mid]==1:

            mid=mid+1

        else:
            a[mid],a[hi]=a[hi],a[mid]
            hi=hi-1
    return a
```

Date: ...../...../.......

a = [2,2,0,1,1,0]

mid=0, l=0, h=5

① while mid <= h:

0 . 5

a[0] == 2:

execute else and swap with mid, hi.

a[0], a[5] = a[5], a[0]

h = h-1 = 4

[0, 2, 0, 1, 1, 2]

② 0 <= 4:

a[0] == 0:

swap with (l)

a[0], a[0] = a[0], a[0]

mid = mid+1 → 1

l = l+1 → 1

[0, 2, 0, 1, 1, 2]

③ 1 <= 4

a[1] == 2

execute else and swap with (h, mid)

a[1], a[4] = a[4], a[1]

[0, 1, 0, 1, 2, 2]. h = h-1 = 4-1 = 3

④ 1 <= 4

a[1] == 1

~~swap with m~~

mid = mid+1

[0, 1, 0, 1, 2, 2] = 1 mid=2

l mid hi

Date: ...../...../.....

(5) 2<=3

a[2]==0

then swap with low

a[2], a[1] = a[1], a[2]

mid = mid+1 - 2+1=3

low = low+1 - 1+1=2

∴ [0,0,1,1,2,2]

(6) 3<=3

a[3]==1

then mid+1 = 4

4<=3 # ∴ Done

→ O(n)

& - O(1) - no extra space

Date: ...../...../.....

① 1st non repeat element

② Kth non repeat element

③ Linked list

1) find middle element (single traversal)
2) Kth ~~node~~ element of cinked list
3) linked list palindrome.
4) single linked list
5) double linked list

Date: ...../...../.....

(1) middle element in single linked list :-

① Double traversal

1) Traverse for length

2) Traverse till middle (length/2)

② Single traversal

2 pointers

1) slow (Increase by 1)

2) fast (Increase by 2)

(end)

while fast.next = none, then

slow reaches middle

Print (slow)

Date: ...../...../.....

(2) Kth element in linked list

O(n) - Time
O(1) - space

Algorithm

1) Initialize count = 0
2) Loop through linked list
   a) If the count is equal to passed index then return the current node
   b) Increment count
   c) change current to point to next of current

```
def getNth(self, index):
    current = self.head
    count = 0

    while (current):
        if count == index:
            return current.data
        count += 1
        current = current.next

    assert (false)
    return 0
```

Date: ...../...../.....

③ palindrome in Singly Linked list

(R) → (A) → (D) → (A) → (R)

Method 1: Use a stack

1) Traverse the given list from head to tail and push every visited node to stack.

2) Traverse the given list from head to tail (Traverse again)
for every visited node, pop a node from stack and

```
while head != None:
    i = stack.pop()
    if head.data == i
        is_palin = True
    else:
        is_palin = False
        break
    head = head.ptr
```

T - O(n)

S - O(n) - stack

Date: ...../...../.....

(4) nth from last: Time O(n) - Space of linked list O(1)

```
def printNthFromLast(self, n)
    temp = self.head
    length = 0
    while temp is not None:
        temp = temp.next
        length += 1
    if n > length:
        print("index exceeded")
        return

    temp = self.head
    for i in range(0, length-n):
        temp = temp.next
    print(temp.data)
```

Date: ...../...../.....

(4) Search of the node in linked list?

(1) Iterative Soln: Bool search(Node head, Info)

1) Initialize a node pointer, ~~head~~
current = head.

2) while current is not Null:

a) If current → info = key. return true

b) else: current = current → next

3) Return false.

```
def search (self, x):                    O(n)
    current = self.head                  O(1)

    while current != None:

        if current.data == x:
            return true # found
        current = current.next

    return false # not found
```

Date: ...../...../.....

② Recursive O(n)

bool search(head, x): O(n)

1) If head is NULL, return false.

2) If head's key is same as x, return tr

3) Else return search(head->next, x)

* def search(self, li, key):

if(not li):
return False

if(li.data == key):
return True
return self.search(li.next, key)

Date: ...../...../.....

Deleting a node:

1) Beginning

point head to next node.

Second node head = head -> next

2) Delete from end:

point head to previous element

* last second element, change next pointer to null.

```
struct node *temp = head;
struct node *prev = NULL;
while (temp -> next != NULL)
{
    prev = temp;
    temp = temp -> next
}
prev -> next = NULL;
```

Date: ...../...../.....

3) Delete from middle

Traverse to element before the element to be deleted change next pointer to exclude the node from chain

```
for(int i=2; i< position ;i++)
{
    if(temp->next != NULL)
    {
        temp= temp->next;
    }
}
temp->next = temp->next->next;
```

O(n)

Date: ...../...../.....

Linked list types

1) singly
2) Doubly
3) Circular
4) Singly - circular
5) Doubly circular
6) Header

Date: ...../...../.....

Searching and Sorting

Searching:

1) membership operators:
   in
   not in

2) Linear search: O(n)

Iterating over an array and returning the index of the first occurence of an item once it is found

3) Binary search: O(log n)

4) Ternary search

5) Jump

6) Interpolation

7) Exponential

Most optimal - cycle sort ttp

*Does less memory writes than any other

Date: ...../...../.....

1) Basic idea for heap sort

Sorting

2) Not stable

3) In-place.

① Selection Sort:

i) Initialize minimum value (min_idx) to location 0.

2) Traverse the array to find minimum element in array.

3) while traversing if any element smaller than min_idx is found then swap both the values.

4) Then increment min_idx to point to next elem

5) Repeat until sorted

```
for i in range(len(A)):
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j
    A[i], A[min_idx] = A[min_idx], A[i]
```

T - $O(n^2)$

S - $O(1)$

* Simple comparison based algorithm
* First pass first largest to last place
  Next " second " to second last place.

Date: ...../...../.....

② Bubble Sort :

```
n = len(arr)

for i in range(n):
    swapped = False
    for j in range(0, n-i-1):
        if arr[j] > arr[j+1]:
            arr[j], arr[j+1] = arr[j+1], arr[j]
            swapped = True
    if swapped == False:
        break
```

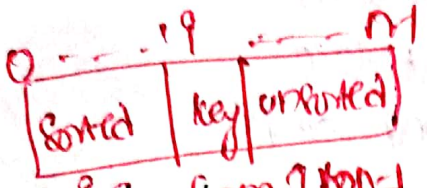
* T: $O(n^2)$
* S: $O(1)$

worst case:

Total no. of swaps = Total no. of comparisons = $n(n-1)/2$

worst case & avg case: $O(N^2)$ / Best = $O(N)$

auxiliary space - $O(1)$



T - $O(n^2)$

S - $O(1)$

Date: 9 Jan 2024

3) Insertion Sort

* $\Omega(n^2)$ - worst case → Reverse sorted.
* In-place - Stability
* Incremental approach
* Used in practice for small arrays.
  (Tim sort and Intro sort)
  - Python - merge sort (n log n)
    - Insertion
  - → Heap
  - → Quick
  - → Insertion.
* $\Omega(n)$ - Best case → already sorted
* Average - $O(n^2)$
* Used for already sorted (or) almost sorted.

```
# def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = j-1
        while j>=0 and key < arr[j]:
            arr[j+1] = arr[j]
            j-=1          # (j = j-1)
        arr[j+1] = key
```

7 - Worst Best Avg
O(n logn)

$T(n) = 2T(n/2) + \theta(n)$

8 - O(n)

Date: ..../......../..............

Divide and Conquer

(1) merge sort

Divide

```
def mergesort(arr):
    if len(arr)>1:
        mid = len(arr)//2
        L = arr[:mid]        # Divide
        R = arr[mid:]

        mergesort(L)
        mergesort(R)

        i = j = k = 0
        while i<len(L) and j<len(R):
            if L[i]<R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1
```

compare (or) merge

$T(n) = 2T(n/2) + \Theta(n)$

space : $O(n)$

Date: ...../...../.....

```
while i < len(L):
    arr[k] = L[i]
    i += 1
    k += 1

while j < len(R):
    arr[k] = R[j]
    j += 1
    k += 1
```

→ to check any element is left

print(mergeSort([12, 11, 13, 5, 6, 7]))

1) Divide and Conquer
2) No in a typical implementation
3) merge sort is stable

Scanned with CamScanner

Date: ...../...../.....

(5) Quick sort

1) Divide and Conquer algorithm

2) worst case Time : $O(n^2)$

3) Despite $O(n^2)$ worst case, it is considered faster, because of the following reasons.

   a) In-place
   b) cache friendly
   c) Avg case in $O(n \log n)$
   d) Tail recursive: (If a recurrence function does recursion as last step.)

4) Partition is key function (Naive, Lomuto, Hoare)

Naive — Stable

Lomuto & Hoare — not stable.

Date: ...../...../.....

Partition

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1
```

Quicksort

```
def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)
        quickSort(array, low, pi-1)
        quickSort(array, pi+1, high)
```

$\theta(n)$ → partition(array, low, high)

$T(k)$ → quickSort(array, low, pi-1)

$T(n-k)$ → quickSort(array, pi+1, high)

Date: ...../...../.....

Analysis

Recursive parts

* $T(N) = T(k) + T(n-k-1) + \Theta(n)$ — partition

1) worst case:

$T(n) = T(0) + T(n-1) + \Theta(n)$

$T(n) = T(n-1) + \Theta(n)$ → when first or last pivot

Recurrence — $O(n^2)$

2) Best case: (Recursion)

$T(n) = 2T(n/2) + \Theta(n)$ — partition

$O(n \log n)$

3) Avg case:

$T(n) = T(n/9) + T(9n/10) + \Theta(n)$

$O(n \log n)$

Date: ...../...../.....

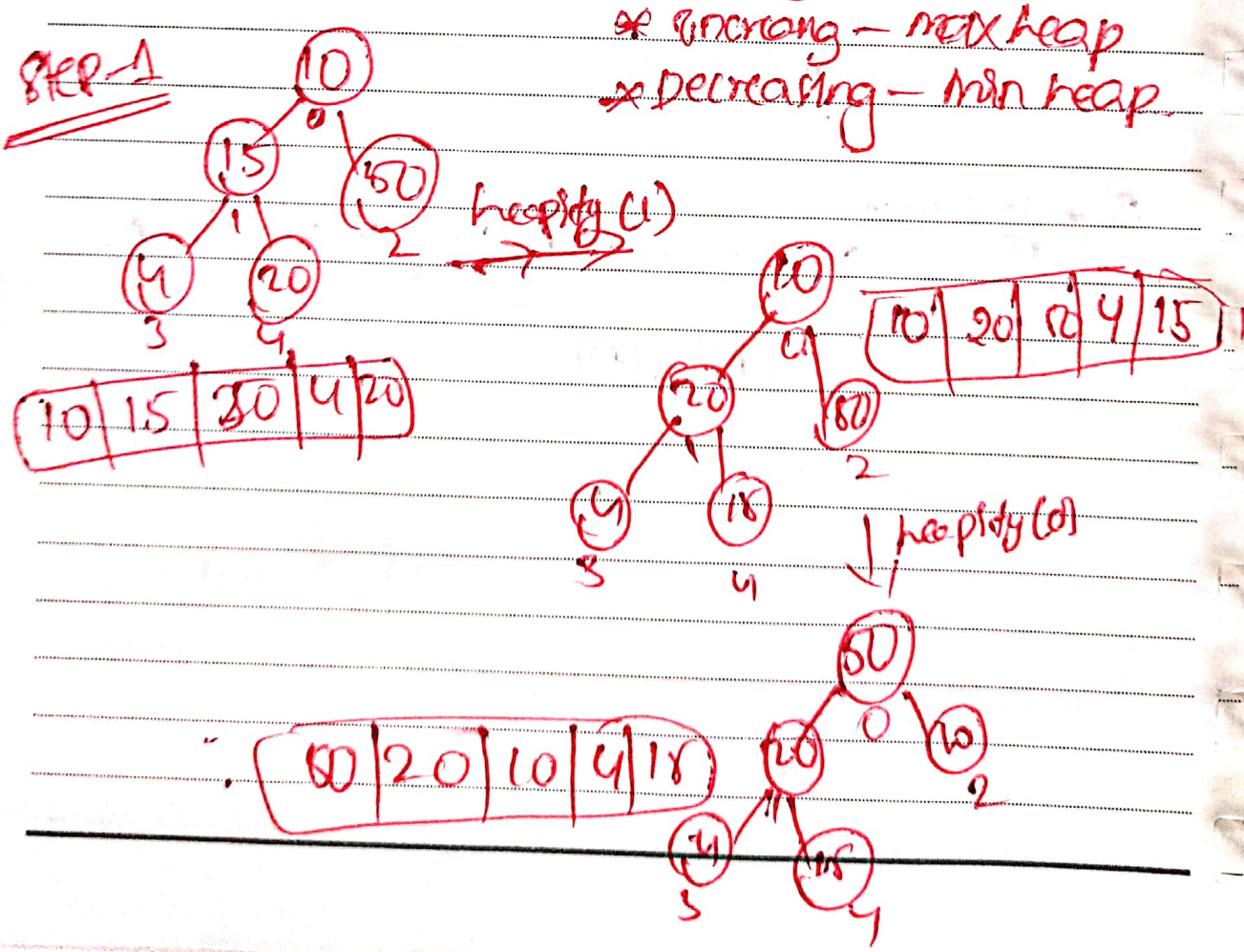
Heap Sort

* Based on selection sort
* In selection sort we use linear search to find the max element and swap with the last element [O(n2)]
* In heap sort, uses max heap data structure, to get O(n)
* Then do heapify.

e/p arr = [10, 15, 50, 4, 20]

* Increasing – max heap
* Decreasing – min heap

Step-1



Date: ...../...../.....

```
def heapify(arr, n, i):
    largest = i  # initialize largest as root
    l = 2*i+1    # left
    r = 2*i+2    # right

    if l<n and arr[largest] < arr[l]:
        largest = l

    if r<n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]

        heapify(arr, n, largest)
```

```
def heapsort(arr):
    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)
```

Date: ...../...../.....

1) T = O(nlogn)

O(logn) - Time comp of heapify.

O(n) - create and BuildHeap()

2) In place

3) not stable, but node stable

Advantages

1) efficiency
2) memory usage
3) simplicity.