
POST

Algorithms

Frequency Count Method:

A = [8 | 3 | 4 | 7 | 2] n = 5

```
Algorithm Sum(A, n)
{
    s = 0;                      → 1
    for (i=0; i<n; i++)         → n+1
    {                           (i=0: 1, i<n: n+1, i++: n)
        s = s + A[i]            → n
    }
    return s;                   → 1
}
```

$f(n) = 2n + 3$

$O(n)$

i < n

i = 0 already i = 0 — In total 6 comparisons

i = 1

i = 2

i = 3

i = 4 — 4 times true

i = 5 ✗ → 1 time false

n + 1

Space

A — n

n — 1

s — 1

i — 1

$S(n) = n + 3$

$O(n) = n + 3$

```
Algorithm Add(A, B, n)
{
    for (i=0; i<n; i++)                 → n+1
    {
        for (j=0; j<n; j++)             → n × n+1
        {
            C[i][j] = A[i][j] + B[i][j]; → n × n
        }
    }
}
```

$f(n) = 2n^2 + 2n + 1$

$O(n^2)$

Space

A → n^2

B — n^2

C — n^2

n — 1

i — 1

j — 1

$S(n) = 3n^2 + 3$

$O(n^2)$

```
Algorithm Multiply(A, B, n)
{
  for (i=0; i<n; i++)          — n+1
  {
    for (j=0; j<n; j++)        — n×n+1
    {
      C[i,j] = 0               — n×n
      for (k=0; k<n; k++)      — n×n×n+1
      {
        C[i,j] = C[i,j] + A[i,k] * B[k,j];   — n×n×n
      }
    }
  }
}
```

$f(n) = 2n^3 + 3n^2 + 2n + 1$

$O(n^3)$

Space

A — n^2

B — n^2

C — n^2

n — 1

i — 1

j — 1

k — 1

$S(n) = 3n^2 + 4$

$O(n^2)$

Time Complexity:

① • for (i=0; i<n; i++) ⟶ n+1
{
  Stmt; — n
}
O(n)

• for (i=n, i>0 ; i--) ⟶ n+1
{
  Stmt; — n
}
O(n)

• for (i=n, i>0; i=i+2) — n+1
{
  Stmt; — n/2
}
O(n)

② for (i=0; i<n; i++) — n+1
{
  for (j=0; j<n; j++) — n×n+1
  {
    Stmt; — n×n
  }
}
O(n²)

③
```
for (i=0; i<n; i++)
{
  for (j=0; j<i ; j++)
  {
    Stmt;
  }
}
```

total no of times = n(n+1)/2

O(n²)

i	j	no of times
0	0	0
1	0 ✓	1
	1 ×	
2	0 ✓	2
	1 ✓	
	2 ×	
3	0 ✓	3
	1 ✓	
	2 ✓	
	3 ×	
⋮	⋮	⋮
n		n

total = 1+2+3+⋯n = n(n+1)/2

④

```
p=0
for(i=1; p<=n; i++)
{
   p= p+i;
}
```

i	p
1	0+1 = 1
2	1+2 = 3
3	1+2+3
4	1+2+3+4
⋮	
K times	$1+2+3+\cdots K = \frac{K(K+1)}{2}$

Assume $p>n$, it'll stop.

$\because p = \frac{K(K+1)}{2}$

$\frac{K(K+1)}{2} > n$

$\simeq K^2 > n$

$K > \sqrt{n} \Rightarrow$ **$O(\sqrt{n})$**

⑤

```
for(i=1; i<n; i=i*2)
{
   Stmt;
}
```

i
1
$1\times2 = 2$
$2\times2 = 2^2$
$2^2\times2 = 2^3$
⋮
2^K

Assume $i>=n$, it'll stop.

$2^k >= n$

$2^k >= n$

$K = \log_2 n \Rightarrow$ **$O(\log_2 n)$**

↳ take ceil value.

Let n=8

i
1 < 8 ✓
2 < 8 ✓
4 < 8 ✓
8 < 8 ✗

3 times

$\log 8 = 3$

n = 10

i
1 < 10 ✓
2 < 10 ✓
4 < 10 ✓
8 < 10 ✓
16 < 10 ✗

4 times

$\log_2 10 = 3.2$

⑥

```
for(i=n; i>=1; i=i/2)
{
   Stmt;
}
```

i
n
$\frac{n}{2}$
$\frac{n}{2^2}$
⋮
$\frac{n}{2^k}$

Assume $i<1$.

$\frac{n}{2^k} < 1$

$n = 2^k$

$k = \log_2 n$ ↘ **$O(\log_2 n)$**

⑦
```
for(i=0; i*i<n; i++)
{
  stmt;
}
```

$i*i < n$

Assume $i*i >= n$

$i^2 = n$

$i = \sqrt{n}$

$O(\sqrt{n})$

⑧
```
for (i=0; i<n; i++)
{
  stmt;
}
for (j=0; j<n; j++)
{
  stmt;
}
```

stmt; —— n

stmt; —— n

independent loops

$O(n)$

⑨
```
p=0
for (i=1; i<n; i=i*2)
{
  p++;
}
for(j=1; j<p; j=j*2)
{
  stmt;
}
```

p++; —— log n

stmt; —— log p

$\log(\log(n))$

$O(\log \log n)$

(10)
```
for(i=0; i<n; i++)          ——— n
{
    for(j=1; j<n; j=j*2)    ——— n x logn
    {
        Stmt;               ——— n x logn
    }
}
```

O(nlogn)

```
for(i=0; i<n; i++)          ——— O(n)
for(i=0; i<n; i=i+2)        ——— O(n)  (n/2)
for(i=n; i>1; i--)          ——— O(n)
for(i=1; i<n; i=i*2)        ——— O(log2 n)
for(i=1; i<n; i=i*3)        ——— O(log3 n)
for(i=n; i>1; i=i/2)        ——— O(log2 n)
for(i=0; i*i<n; i++)        ——— O(√n)
```

Analysis of if and while:

- `i=0` —— 1

```
while (i<n)      —— n+1
{
  Stmt;          —— n
  i=i+1;         —— n
}
```

$O(n)$

Similar $\equiv$ to

```
for (i=0; i<n; i++)   —— n+1
{
  Stmt;               —— n
}
```

$O(n)$

-

```
a=1
while (a<b)
{
  Stmt;
  a=a*2;
}
```

a
1
1×2
2×2
$2^2 \times 2$
$2^3 \times 2$
⋮
2^k

$\Rightarrow$

```
for (a=1; a<b; a=a*2)
{
  Stmt;
}
```

$O(n)$

$2^k >= b$

$2^k = b$

$k = \log_2 b \Rightarrow O(\log_2 n)$

-

```
i=1;
k=1;
while (k<n)
{
  Stmt;
  k=k+i;
  i++;
}
```

i	k
1	1
2	1+1
3	2+2
4	2+2+3
5	2+2+3+4
⋮	⋮
m	$2+2+3+4+\dots+m \simeq 1+\frac{m(m+1)}{2} \simeq \frac{m(m+1)}{2}$

Assume $k >= n$

$\frac{m(m+1)}{2} \geq n$

$m = \sqrt{n}$

$\Rightarrow O(\sqrt{n})$

- While (m!=n)

```
{
  if (m>n)
      m=m-n;
  else
      n=n-m;
}
```

O(n)

min O(1)

- Algorithm Test(n)

```
{
  if (n<5)
  {
     printf ("%d", n);   ---> O(1)
  }
  else
  {
     for(i=0; i<n; i++)
     {
       printf("%d", i);   ---> n   O(n)
     }
  }
}
```

Best case : O(1)

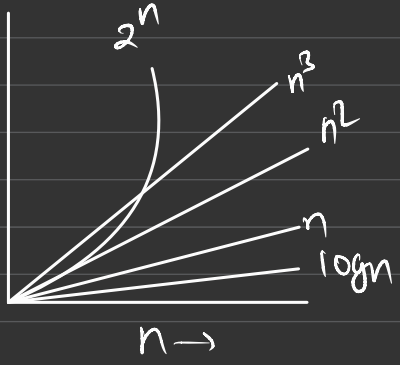
Worst case: O(n)

Types of time functions:

- $O(1)$ → constant
- $O(\log n)$ → logarithmic
- $O(n)$ → linear
- $O(n^2)$ → Quadratic
- $O(n^3)$ → Cubic
- $O(2^n)$ → exponential

Weightage:

$$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n \cdots < n^n$$



ASYMPTOTIC NOTATIONS

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

O - big Oh upper bound
Ω - big Omega lower bound
Θ - theta Average bound

Big-Oh:

The fn $f(n) = O(g(n))$ iff $\exists$ +ve constants C and n_0 such that

$$f(n) \leq C * g(n) \quad \forall \; n \geq n_0$$

Eg: $f(n) = 2n+3$

$2n+3 \leq 10n \quad n \geq 1$

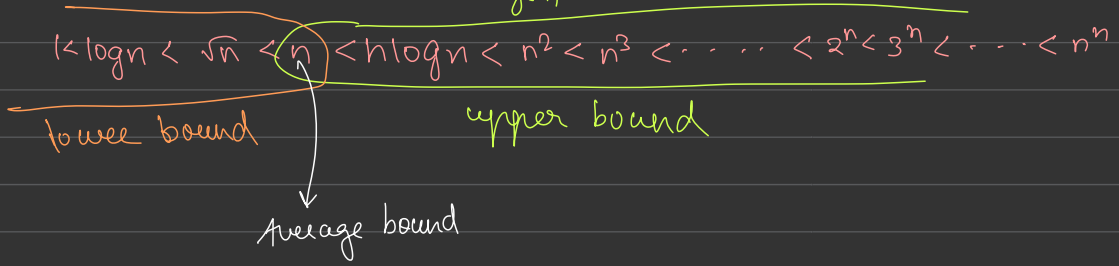
write whatever you want but make sure LHS $\leq$ RHS.

Simple trick

$2n+3 \leq 2n+3n$

$2n+3 \leq 5n \quad n \geq 1$

$g(n)$



Omega:

The fn $f(n) = \Omega(g(n))$ iff $\exists$ +ve constants c and n_0 such that

$$f(n) \geq c * g(n) \quad \forall \; n \geq n_0$$

Eg: $f(n) = 2n+3$

$2n+3 \geq 1n \quad \forall n \geq 1$

$f(n) \geq c\,g(n)$

Write $\log n$ (or) 1 (or) $\sqrt{n}$

$$f(n) = \Omega(n)$$

Theta:

The fn $f(n) = \Theta(g(n))$ iff $\exists$ +ve constants c_1, c_2 and n_0 such that

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall \; n \geq n_0$$

Eg: $f(n) = 2n+3$

$$1 \times n \leq 2n+3 \leq 5 \times n$$

$$f(n) = \Theta(n)$$

Unit 3 Sorting Algorithms

4,3,7,1,2 → SORTING → 1,2,3,4,7

		Time Complexity
1. Bubble Sort	Exhaustive Search	$O(n^2)$
2. Selection Sort		$O(n^2)$
3. Insertion Sort	Divide and Conquer Techniques.	$O(n^2)$
4. Merge Sort		$O(n \log n)$
5. Quick Sort		$O(n \log n)$

BUBBLE SORT → After first pass, largest element bubbles to last position.

	0	1	2	3	4	
first pass:	11	77	42	37	56	⇒ list A
	11	42	37	56	77	⇒ 4 comparisons ⇒ $\sum_{i=0}^{n-i-1}$
second pass:	11	37	42	56	77	⇒ 3 comparisons ⇒ $\sum_{i=1}^{n-i-1}$
third pass:	11	37	42	56	77	⇒ 2 comparisons ⇒ $\sum_{i=1}^{n-i-1}$
fourth pass:	11	37	42	56	77	⇒ 1 comparisons ⇒ $\sum_{i=1}^{n-i-1}$

Here $n = 5$

- no. of pass = 4 ⇒ $n-1$
- no. of comparisons = $n-i-1$

So basically i should from 0, i.e. $n-1$ (4) & totally 4 pass, so till (0, 1, 2, 3)

Pseudo Code in Python:

```
def Bubblesort(A):
    n = len(A)
    for i in range(0, n-1):
        for j in range(0, n-i-1):
            if A[j] > A[j+1]:
                A[j], A[j+1] = A[j+1], A[j]
    Return
```

for i in range(0, n): → unnecessary extra comparison.

j → Observe the pass → i should start from 0 goes till $n-1$ (4)

Start i with 0

Time Complexity:

$$T(n) = \sum_{i=0}^{n-1-1} \sum_{j=0}^{n-i-1} (1) = \sum_{i=0}^{n-2} \sum_{j=0}^{(n-i-2)} (1)$$

(upper limit − lower limit + 1) = total no. of times

$$= \sum_{i=0}^{n-2} (n-i-2-0+1)(1) = \sum_{i=0}^{n-2} (n-i-1)$$

$$= n-1 + n-2 + n-3 + \dots$$

$$= n \times n - \frac{n(n+1)}{2} = \frac{n^2 - n}{2} = \frac{n(n-1)}{2}$$

$T(n) = \Theta(n^2)$

HW:

- Selection Sort
  - ↳ given a list of elements, it'll find the minm element & swap the minm element with 0th index.
  - ↳ note down no of comparisons
  - ↳ second minm element swapped with 1st index element
  - ↳ continue

Inner loop: finding the minm element index
swap outside inner loop

Outer loop:

Just write the iteration & find out no. of pass,
no. of comparisons.

Selection Sort: After the first pass, smallest element is swapped with first index.

Always start writing inner loop first.

	0	1	2	3	4	5	6
A =	14	3	22	1	7	-1	88

```
n = len(A)
for i in range(0, n-1):
    min = A[i]
    minindex = i
    for j in range(i+1, n):
        if (A[j] < min):
            min = A[j]
            minindex = j
    A[i], A[minIndex] = A[minIndex], A[i]
```

i = 0

j = 0+1 to n-1

A[j] < min

min = A[j]

minIndex = j

	0	1	2	3	4	5	6
Pass 1	14	3	22	1	7	-1	88
Pass 2	-1	3	22	1	7	14	88
Pass 3	-1	1	22	3	7	14	88
Pass 4	-1	1	3	22	7	14	88
Pass 5	-1	1	3	7	22	14	88
Pass 6	-1	1	3	7	14	22	88

Sorted!!

Time Complexity:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} (1)$$

$$= \sum_{i=0}^{n-1} (n-1-i-1+1)$$

$$= \sum_{i=0}^{n-1} (n-i-1)$$

$$= n-1 + n-2 + \dots + 3 + 2 + 1$$

$$= \frac{n(n-1)}{2}.$$

$$T(n) = \frac{n^2-n}{2}.$$

$\therefore$ $O(n^2)$ is time complexity.

HW

l = | 4 | 8 | 6 | 2 | 5 | 0 |

Bubble Sort:

```
n = len(l)
for i in range(n-1):
    for j in range(0, n-i-1):
        if l[j] > l[j+1]:
            swap [l[j], l[j+1]]
```

4 8 6 2 5 0

1st: 4 6 2 5 0 | 8

2nd: 4 2 5 0 | 6 8

3rd: 2 4 0 | 5 6 8

4th: 2 0 | 4 5 6 8

5th: | 0 | 2 | 4 | 5 | 6 | 8 |

Sorted

Selection Sort:

4 8 6 2 5 0

1st: 0 | 8 6 2 5 4

2nd: 0 2 | 6 8 5 4

3rd: 0 2 4 | 8 5 6

4th: 0 2 4 5 | 8 6

5th: | 0 | 2 | 4 | 5 | 6 | 8 |

Sorted

Insertion Sort

A =

14	3	2	1	7	-1	88

0

14

Pass ①

0 1

14	3

Temp = 3

i = 1, Temp = 3

i-1 = 0

A[0] > Temp:

A[1] = A[0]

j = j-1

A[j+1] = Temp

3	14

②

3	14	2

i = 2, j = i-1 = 1

Temp = 2

A[i] > Temp:

(A[1] > 2):

A[j+1] = A[j]

A[0] > temp

A[1] = A[0]

Pseudocode:

```
for i in range(1,n):
    temp = A[i]
    j = i-1
    while (j>=0 and A[j]>temp):
        A[j+1] = A[j]
        j = j-1
    A[j+1] = Temp
```

Insertion Sort

0	1	2	3	4	5	6	7
97	15	24	16	72	53	11	102

Pass 1: $i = 1$, $j = 0$

temp = 15.
A[0] = 97 > temp
A[1] = 97
j = -1
A[0] = temp = 15

15	97	24	16	72	53	11	102

Pass 2: $i = 2$, $j = 1$

temp = 24
A[1] > 24
97 > 24 ✓
A[2] = A[1] = 97
j = 0
A[0] > 24 ✗
A[1] = 24

→

15	24	97	16	72	53	11	102
0	1	2	3	4	5	6	7

Pseudocode:

```
for i in range(1,n):
    temp = A[i]
    j = i-1
    while (j >= 0 and A[j] > temp):
        A[j+1] = A[j]
        j = j-1
    A[j+1] = Temp
```

Pass 3: $i = 3$, $j = 2$

temp = 16
A[2] > temp
97 > 16 ✓
A[3] = A[2] = 97
j = 1
A[1] > temp
24 > 16 ✓
A[2] = A[1] = 24
j = 0
A[0] > temp
15 > 16 ✗
A[1] = 16

⇒

15	16	24	97	72	53	11	102
0	1	2	3	4	5	6	7

Pass 4: $i = 4$, $j = 3$

temp = 72
A[3] > temp
97 > 72 ✗ ✓
A[4] = A[3] = 97
j = 2
A[2] > temp
24 > 72 ✗
A[3] = 72.

15	16	24	72	97	53	11	102
0	1	2	3	4	5	6	7

Pass 5: $i = 5$, $j = 4$

temp = 53
A[4] > temp
97 > 53 ✓
A[5] = A[4] = 97
j = 3
A[3] > temp
72 > 53 ✓
A[4] = A[3] = 72
j = 2
A[2] > temp ✗
A[3] = 53

15	16	24	53	72	97	11	102

Pass 6: $i = 6$, $j = 5$

temp = 11
A[5] > temp ✓
A[6] = A[5] = 97
j = 4
A[5] = A[4] = 72
j = 3
A[4] = A[3] = 53
j = 2
A[3] = A[2] = 24
j = 1
A[2] = A[1] = 16
j = 0
A[1] = A[0] = 15
j = -1
A[0] = 11

11	15	16	24	53	72	97	102

Pass 7: $i = 7$, $j = 6$

temp = 102
A[6] > 102 ✗
A[7] = 102

11	15	16	24	53	72	97	102

Sorted!!

Time complexity of Insertion Sort:

```
A = [ . . . . . ]
n = len(A)
for i in range(1,n):
    temp = A[i]
    j = i-1
    while (j >= 0 and A[j] > temp):
        A[j+1] = A[j]
        j = j-1
    A[j+1] = temp
```

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} (1)$$

$$= \sum_{i=1}^{n-1} (i-1+1) = \sum_{i=1}^{n-1} i$$

$$= 1 + 2 + \cdots + n-1$$

$$= \frac{n(n-1)}{2}$$

$$T(n) = \frac{n^2 - n}{2}$$

$\Theta(n^2)$ is the time complexity

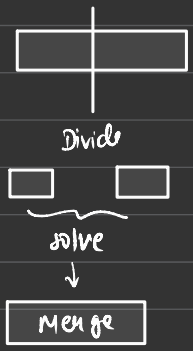
- Insertion Sort is better because less no. of comparisons.

Merge Sort

↳ Divide and Conquer Algorithm.

↓

Recursive Way.



Recursive function:

Eg: Factorial

factorial(5)

1 x 2 x 3 x 4 x 5 → repeated multiplication } so iteration OR RECURSION

5 x 4!, 5 x 4 x 3!, 5 x 4 x 3 x 2!, 5 x 4 x 3 x 2 x 1!

5! → 5 x 4! → 5 x 4 x 3! → 5 x 4 x 3 x 2! → 5 x 4 x 3 x 2 x 1!

Dividing & Conquering

Base condition: where to stop the recursion.

```
def factorial(n):
    if n==0:
        return 1          } → Base condition
    else:
        return n * factorial(n-1)   → Recursive call
```

Memory:

Stack:

Space complexity O(n)

Last In First Out

fact(5)
↓
5 * fact(4)
↓
4 * fact(3)
↓
3 * fact(2)
↓
2 * fact(1)

```
def factorial(n):
    if n==0:
        return 1          } → Base condition
    else:
        return n * factorial(n-1)   } → Recursive call
```

Time Complexity:

$$T(n) = \begin{cases} d, & \text{if } n < 1 \\ T(n-1) + c, & \text{if } n > 1 \end{cases}$$

→ Recurrence equation

$T(n-1)$: Time complexity of $(n-1)$

→ Solve mathematically to find Time Complexity.

Expansion Method:

$T(n) = T(n-1) + c$

$= T(n-2) + 2c$

$= T(n-3) + 3c$

$= T(n-i) + ic$ ⇒ $T(n-(n-1)) + (n-1)c$

$= T(1)$ ⇒ $T(1) + (n-1)c$

⇒ $n - i = 1$

$i = n-1$

$T(n) = d + (n-1)c$

$= cn + d - c$

∴ $O(n)$ is the time complexity.

REFER HOD SIR'S PPT.

Tail recursion:

↳ Special case of recursion where a recursive call appears as the last individual statement & not in an expression.

- We can reduce space complexity of O(n) to O(1) in this recursion.
  ↳ Tail Recursion!!

```
def factorial(n, a=1):
    if n<2:
        return a
    else:
        return factorial(n-1, n*a)
```

Changes from original code is color coded.

5! factorial(5)
a=1

↓

factorial(4, 5×1)
a=5

↓

factorial(3, 4×5)
a=4×5

↓

factorial(2, 3×4×5)
a=3×4×5

↓

factorial(1, 2×3×4×5) → Ans: 2×3×4×5 = 120

n<2 ✓
return a

Refer HOD's ppt.

Search Algorithms

BINARY SEARCH:

- Time Complexity: $O(\log_2 n)$
- Should be Sorted order.

Linear Search
↳ O(n)

CAT 1:
↳ unit 1 fully
↳ unit 3 → except hashing.

```
def search(list, srele):
    """
    Return the Index of the search element
    """
```

① low = 0
high = 6 = len(l) − 1
mid = (low + high) // 2

0	1	2	3	4	5	6
1	1	5	9	11	12	15

l = [1, 1, 5, 9, 11, 12, 15]
srele = 5

while low <= high:

l[mid] = l[6] = 9

② Check srele == l[mid] ⇒ return mid
srele < l[mid] ⇒ 5 < 9 ✓ ⇒ high = mid − 1
srele > l[mid] ⇒ 5 > 9 ⇒ low = mid + 1

repeat

Recursion:

```
def binarySearch(L, low, high, srele):
    if low > high:  # Base condition
        return -1
    else:
        mid = (low + high) // 2
        if srele == l[mid]:
            return mid
        elif srele < l[mid]:
            return binarySearch(L, low, mid-1, srele)
        elif srele > l[mid]:
            return binarySearch(L, mid+1, high, srele)
```

Time Complexity:

Recurrance Equation:

$$T(n) = \begin{cases} 1, & n<1 \\ T(n/2)+1, & n \geq 1 \end{cases}$$

$$T(n) = T(n/2) + 1$$
$$= T(n/4) + 2$$
$$= T(n/8) + 3$$
$$\vdots$$
$$= T(n/2^i) + i$$
$$\vdots$$
$$= T(1)$$

$$\frac{n}{2^i} = 1$$
$$n = 2^i$$
$$i = \log_2 n$$

$$T(n) = T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n$$

$$T(n) = T(1) + \log_2 n$$

$$\therefore T(n) = \log_2 n + \text{const}$$

$$\therefore \boxed{\theta(n) = \log_2 n}$$

MERGE SORT: → Dope Sorting Algorithm !!

Merge: size: m A | 11 | 22 | 71 | 666 | 972 (i → sorted !!) size: n B | 5 | 32 | 42 (j → sorted !!)

Given 2 sorted list

↳ merge two lists & return completely sorted list.

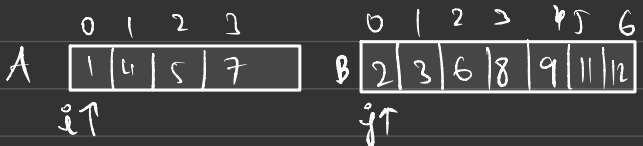
C | | (K=0)

Compare i & j, whichever is less, copy to C.

if one list gets over, copy the remaining elements to this of other list

5 | 11 | 22 | 33 | 42 | 71 | 666 | 972

```
def merge(A,B):
    C = []
    i = j = k = 0
    m = len(A)
    n = len(B)
    while(i<m & j<n):
        if A[i] < B[j]:
            C[k] = A[i]
            i = i+1
        else:
            C[k] = B[j]
            j = j+1
        k = k+1
    while(i<m):
        C.append(A[i])
        i = i+1
    while(j<n):
        C.append(B[j])
        j = j+1
```



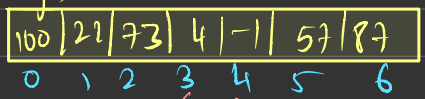

```
def merge(A,B):
    C=[]
    i=j= k =0
    m=len(A)
    n= len(B)
    while (i< m & j< n):
        if A[i] < B[j]:
            C.append (A(i))
            i=i+1
        else:
            C.append (B[j])
            j=j+1
        k=k+1
    if (i<m):
        C.extend(A[i:])
    elif (j< n):
        C.extend (B[j:])
    else:
        Return result
```

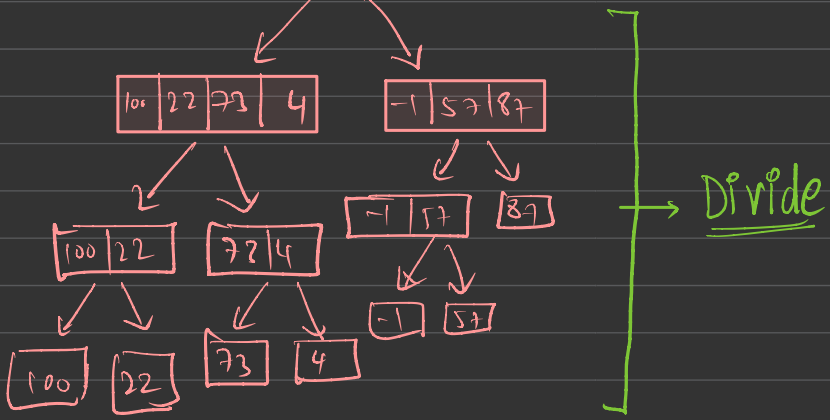
Merge sort:

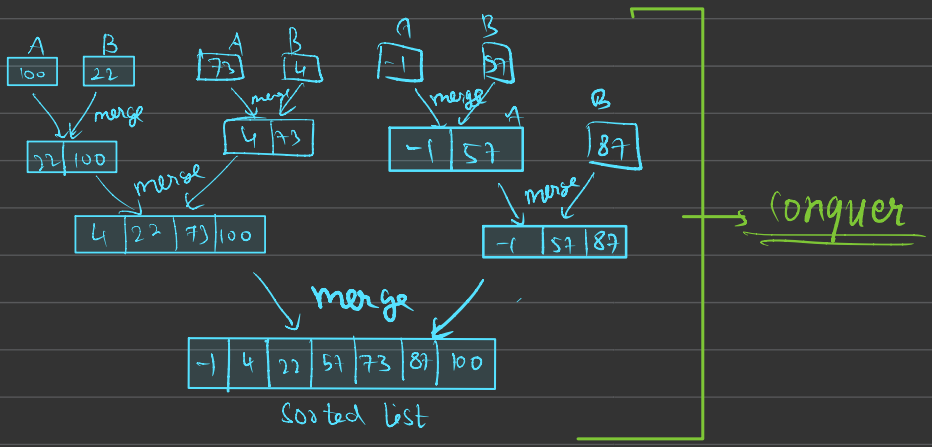
↓

Divide and conquer.

eg:







Pseudo code:

```
def mergesort(list):
    length = len(list)
    # Base condition
    if(length < 2):
        return (list[:])
    else: # divide
        mid = length//2
        return (msort(list[:mid]), msort(list[mid:]))
```

Time Complexity:

$$T(n) = \begin{cases} d, & \text{if } n < 2 \\ 2T(n/2) + n, & \text{if } n \geq 2. \end{cases}$$

$$T(n) = 2T(n/2) + n$$
$$= 2\left(2T(n/4) + \frac{n}{2}\right) + n$$
$$= 4T(n/4) + 2n.$$
$$= 8T(n/8) + 3n$$

$$2^k T\left(\frac{n}{2^k}\right) + kn.$$

$$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$$

$$T(n) = n(1) + (\log_2 n)n.$$
$$\Rightarrow n + n\log_2 n.$$

$$\boxed{\Theta(n) = n \log n}$$

Quick Sort

↳ $O(n \log_2 n)$

Divide and Conquer.

11, 5, 6, 7, 1, 99, 15

Pivot = 15
(Can be any value)

After round 1,

(nos less than 15) 15 (nos greater than 15)

basically it is placed at the right position

11, 5, 6, 7, 1 15 , 99

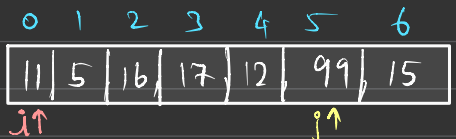
new list
new pivot = 1

1 | 11, 5, 6, 7

new list
new pivot = 7

1 5 6 7 11

1 5 6 7 11 15 99 ⟹ Pivot.



begin = 0

end = 6

pivot = a[end] #15

'i' → initialised to begin

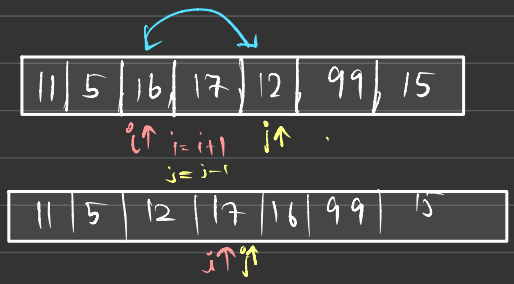
'j' → initialised to end

Objective of i:

↳ find num < 15

Objective of j:

↳ find num > 15



Pseudo code:

```
def partition(list, begin, end):
    pivot = list[end]
    i = begin
    j = end-1
    While (i <= j):
        while (list[i] < pivot):
            i = i+1
        while (list[j] > pivot):
            j = j-1
        if (i<j):
            list[i], list[j] = list[j], list[i]
            i = i+1
            j = j-1
    if (i<end):
        list[i], list[end] = list[end], list[i]
    return i
```

Example:

16, 7, 9, -1, 10, 11, 15, 14, 22, 13

0	1	2	3	4	5	6	7	8	9
16	7	9	-1	10	11	15	14	22	13

begin = 0

end = 9

pivot = 13

i = 0

j = 8

.

```
def partition(lst,begin,end):
    pvt = lst[end]
    i = begin
    j = end-1
    while(1<=j):
        while( lst[i]< pvt):
            i=i+1
        while(lst[j] > pvt):
            j=j-1
        if (i<=j):
            lst[i],lst[j] = lst[j],lst[i]
            i=i+1
            j=j-1
    if i<=end:
        lst[i],lst[end] = lst[end],lst[i]
    return i
```

Example:

-2, 44, 1, 9, 111, 12, 33, -4, 5, 77, 52, 43, 25

0 1 2 3 4 5 6 7 8 9 10 11 12

begin = 0
end = 12
pivot = 25
i = 0
j = 11

```
def partition(list, begin, end):
    pivot = list[end]
    i = begin
    j = end-1
    while (i <= j):
        while (list[i] < pivot & i <= end):
            i = i+1
        while (list[j] > pivot & j >= begin):
            j = j-1
        if (i<j):
            list[i], list[j] = list[j], list[i]
            i = i+1
            j = j-1
    if (i<end):
        list[i], list[end] = list[end], list[i]
    return i
```

①

0	1	2	3	4	5	6	7	8	9	10	11	12
-2	44	1	9	111	12	33	-4	5	77	52	43	25

i=1, j=8

i<j ✓ ⇒ 44, 5

i=2, j=7

0	1	2	3	4	5	6	7	8	9	10	11	12
-2	5	1	9	111	12	33	-4	44	77	52	43	25

i=4, j=7

i<j ✓ ⇒ swap (111, -4) & i=5, j=6

0	1	2	3	4	5	6	7	8	9	10	11	12
-2	5	1	9	-4	12	33	111	44	77	52	43	25

i ≯ j ⇒ swapping pivot & i

0	1	2	3	4	5	6	7	8	9	10	11	12
-2	5	1	9	-4	12	25	111	44	77	52	43	33

Return i

return 6.

Quick Sort Code:

```
def partition(list, begin, end):
    pivot = list[end]
    i = begin
    j = end-1
    while(i <= j):
        while(list[i] < pivot & i <= end):
            i = i+1
        while(list[j] > pivot & j >= begin):
            j = j-1
        if (i<j):
            list[i], list[j] = list[j], list[i]
            i=i+1
            j = j-1
    if (i<end):
        list[i], list[end] = list[end], list[i]
    return i
```

```
def qsort(list, begin, end):
    if (begin < end):
        k = partition(list, begin, end)
        qsort(list, begin, k-1)
        qsort(list, k+1, end)
    return
```

```
def partition(list, begin, end):
    pivot = list[end]
    i = begin
    j = end-1
    while(i<=j):
        while ( list[i] < pivot & i <= end):
            i = i+1
        while ( list[j] > pivot & j >= begin):
            j = j+1
        if (i<j):
            list[i], list[j] = list[j], list[i]
            i = i+1
            j = j-1
    if (i<end):
        list[i], list[end] = list[end], list[i]
    return i

def qsort( list, begin, end):
    if (begin < end):
        k = partition(list, begin, end)
        qsort (list, begin, k-1)
        qsort (list, k+1, end)
        return
```

goodrich

```
def find_median(begin, end, list)
    mid = (begin + end) // 2
    if list[begin] > list[mid]:
        swap(&(begin), &(mid))
    if list[mid] > list[end]:
        swap
    if list[begin] > list[end]:
        swap
    swap(mid, end)
```

begin > mid
mid > end
begin > end
* swap(mid, end)

QuickSort:

```
def median(L):
    n = len(L)
    low, mid, high = L[0], L[n//2], L[n-1]
    if low > high:
        L[0], L[n-1] = L[n-1], L[0]
    if high > mid:
        L[n-1], L[n//2] = L[n//2], L[n-1]
    if low > mid:
        L[0], L[n//2] = L[n//2], L[0]
    return L
def partition(L, begin, end):
    pivot = L[end]
    i, j = begin, end-1
    while (i <= j):
        while (L[i] <= pivot and i < end):
            i += 1
        while (L[j] > pivot and j >= begin):
            j -= 1
        if i < j:
            L[i], L[j] = L[j], L[i]
    L[i], L[end] = L[end], L[i]
    return i, L[i:i+1]
def qsort(L):
    if len(L) < 2:
        return L
    else:
        L = median(L)
        sorted_pos, mid = partition(L, 0, len(L)-1)
        return qsort(L[:sorted_pos]) + mid + qsort(L[sorted_pos+1:])
```

PDS Assignment 1

1) (a) $T(n) = T(n-1) + n$

$T(n) = T(n-1) + n$

$\vdots$

$= T(n-k) + kn - k$ ①

$n - k = 1$

$k = n-1$

↳ Put in ①

$= T(n-(n-1)) + (n-1)n - n + 1$

$= T(1) + n^2 - n - n + 1$

$= T(1) + n^2 - 2n + 1$

$T(n-1) = T(n-2) + n - 1$

$T(n) = T(n-2) + 2n - 1$

$T(n-2) = T(n-3) + n - 2$

$T(n) = T(n-3) + 3n - 3$

$\therefore$ Time complexity is $\Theta(n^2)$

(b) $T(n) = T(n-1) + \log n$.

$T(n) = T(n-1) + \log n$

$\vdots$

$T(n-k) + \log(n(n-1)\cdots(n-k))$ ①

$\vdots$

$T(1)$

$T(n-1) = T(n-2) + \log(n-1)$

$\therefore T(n) = T(n-2) + \log n + \log(n-1)$

$T(n-2) = T(n-3) + \log(n-2)$

$\therefore T(n) = T(n-3) + \log n + \log(n-1) + \log(n-2)$

k times

$n - k = 1 \Rightarrow k = n - 1 \longrightarrow$ Put in ①

$T(n-(n-1)) + \log(n(n-1)\cdots(n-(n-1)))$

$T(1) + \log(n^n)$

$\therefore$ Time complexity is $\Theta(n \log n)$

(c) $T(n) = T(n/2) + n$

$T(n) = T(n/2) + n$ $\quad T(n/2) = T(n/4) + n/2$

$= T(n/4) + \frac{n}{2}$ $\quad T(n) = T(n/4) + 3n/2$

$= T(n/8) + \frac{n}{2^2}$ $\quad T(n/4) = T(n/8) + n/4$

$\vdots$

$= T\left(\frac{n}{2^i}\right) + n\left(\sum_{i=0}^{k} \frac{1}{2^i} + 1\right)$

$\sum_{i=0}^{k} \frac{1}{2^i} \approx 1$

$T\left(\frac{n}{2^i}\right) + n(1+1)$

$\frac{n}{2^i} = 1$

$i = \log_2 n$

$T(1) + 2n.$ $\quad \therefore$ Time complexity is $\Theta(n)$

(d) $T(n) = T(n/2) + n\log n.$

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

$T(n) = T\left(\frac{n}{2^2}\right) + \frac{n}{2}\log\frac{n}{2} + n\log(n)$

$= T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\log\frac{n}{2^2} + \frac{n}{2}\log\frac{n}{2} + n\log n$

$\vdots$

$T\left(\frac{n}{2^i}\right) + \frac{n}{2^{i-1}}\log\frac{n}{2^{i-1}} + \cdots + n\log n.$

$T(1) = O(1)$

$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$

$= T(1) + \frac{n}{2^{\log_2 n - 1}}\log\left(\frac{n}{2^{\log_2 n - 1}}\right) + \cdots + n\log n$

Highest degree: $n\log n$ $\Rightarrow$ Time complexity is $\Theta(n\log n)$

2)

a)

$$\sum_{k=0}^{r-1} \sum_{j=0}^{v-1} \sum_{i=0}^{v-1} (1)$$

$v = n$

$$\sum_{k=0}^{n-1} \left[\sum_{j=0}^{n-1} \left[\sum_{i=0}^{n-1} (1) \right] \right]$$

$$\sum_{k=0}^{n-1} \left[\sum_{i=0}^{n-1} (n-1+1) \right]$$

$$\sum_{k=0}^{n-1} \sum_{j=0}^{n-1} (n)$$

$$\sum_{k=0}^{n-1} [n(n-1+1) - 1 + 1]$$

$$\sum_{k=0}^{n-1} [n(n)]$$

$$= n(n(n-1+1) - 1 + 1) - 1 + 1$$

$$= n^3$$

$\therefore$ Time Complexity is $O(n^3)$

b)
```
def order(x):
    n=0
    while (x!=0):
        n=n+1
        x = x//10
    return n
```
→ as it divides the number by 10 ⇒ it is of function O(log n).

is Armstrong function depends on order function.

∴ Time Complexity = O(log n)

c)

$$T(n) = \begin{cases} d, & root.val == key \\ T(n/2) + d, & root.val != key \end{cases}$$

$T(n) = T(n/2) + d$

$= T(n/4) + 2d$

$= T(n/8) + 3d$

$\vdots$

$T\left(\frac{n}{2^i}\right) + id$

$\frac{n}{2^i} = 1$

$n = 2^i \Rightarrow i = \log_2 n.$

$T(1) + d\log_2 n$

∴ Time complexity is O(log n)

3) (a)

Bubble Sort:

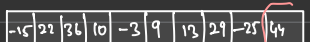
```
def BubbleSort(list):
    n = len(list)
    for i in range(0, n-1):
        for j in range(0, n-i-1):
            if list[j] > list[j+1]:
                list[j], list[j+1] = list[j+1], list[j]
    return list.
```

Example:

i=0

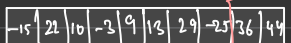
9 comparisons ⇒ n-i-1
10-0-1 = 9
i+=1

first pass:



8 comparisons ⇒ n-i-1
10-1-1 = 8
i+=1

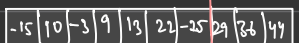
second pass:



i=1

7 comparisons ⇒ n-i-1
10-2-1 = 7
i+=1

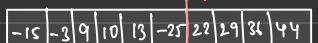
third pass:



i=2

6 comparisons ⇒ n-i-1
10-3-1 = 6

fourth pass:



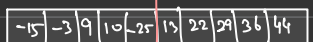
i=3

5 comparisons ⇒ n-i-1
10-4-1 = 5

i+=1

i=4

Fifth pass:

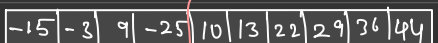


i+=1

4 comparisons ⇒ n-i-1
10-5-1 ⇒ 4

i=5

Sixth pass:

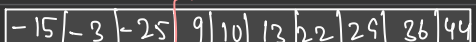


i+=1

3 comparisons ⇒ n-i-1
10-6-1 = 3

i=6

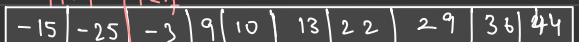
seventh pass:



i+=1 i=2

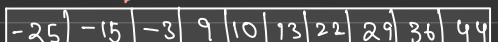
2 comparisons ⇒ n-i-1
10-7-1 = 2

eighth pass:



1 comparison
⇒ n-i-1
⇒ 10-8-1 = 1

ninth pass:



Sorted list

b) Selection sort:

Shallow and Deep Copy

Sequences: Linear data Structures

List, tuples, String

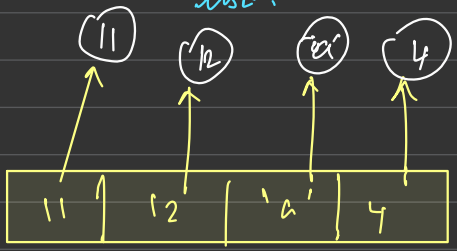
l = [11, 12, 13, 14] — sequential arrays

0	1	2	3
11	12	13	14

Some reference is given to each index

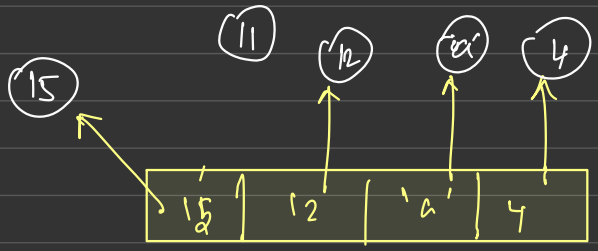
```
l = [1]
import sys
print(sys.sizeof(l))
i = 1
print(sys.sizeof(i))
```

⟶ small memory size than a list.

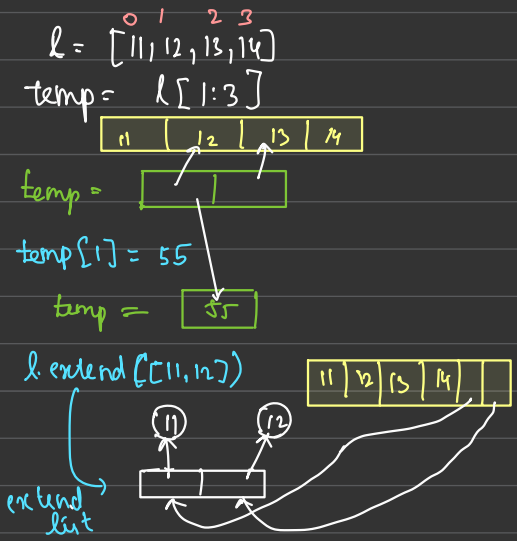


→ some address in Ram.

l[0] = 15



Points to new location.

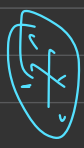


ARRAYS:

Import array

A = array.array(":", print(a[0])) [1,2,3,4,5]

11	12	13	14	15
0	1	2	3	4



DYNAMIC ARRAYS

- not static
  - (changes during runtime)
    - shrinking / expanding.

Stack, Queues, lists

Inheritance in Python

Base class Derived class

Eg: Cricket → players → Batsman

↓

Object: instance of a class

Class: encapsulation of data members & member funcs.

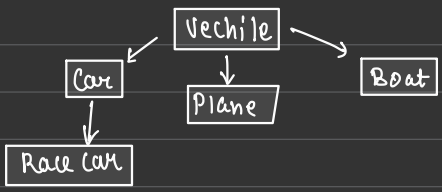
SINGLE INHERITANCE

Vehicle class → Base class Car class → derived class.

```
class Vechicle:
    def __init__(self, wheels):
        self.wheel = wheels
    def info():
        print('Inside Vehicle class')

class Car(Vehicle):
    def __init__(self, brand):
        self.brand = brand
        super().__init__(wheels)
    def car_info(self):
        print('Inside car class')
        print(self.wheel)

# Driver code:
c = Car('Maruti')
c.car_info()  (or)  c.info()
c = Car('Maruti', 4)
```

Inherited

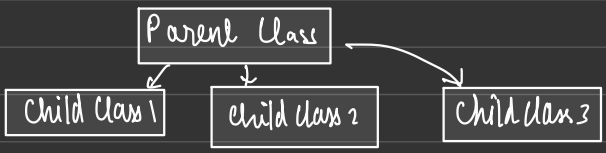
MULTILEVEL INHERITANCE



Syntax:

```
class Vehicle:
    ...

class Car(Vehicle):
    ...

class RaceCar(Car):
    ...
```

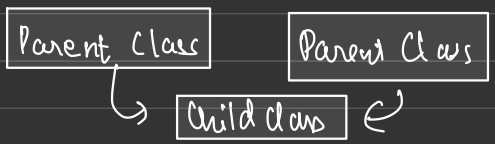
HIERARCHIAL INHERITANCE



Syntax:

```
class Vehicle
    ...

class Car(Vehicle):
    ...

class Truck(Vehicle):
    ...
```

MULTIPLE INHERITANCE:



Syntax:

```
class Person:
    ...

class Company:
    ...

class Emp(Person, Company):
    ...
```

super() function:

↳ Child class la Base class oda function ah invoke pannanum. That time we use super().__fnname__()

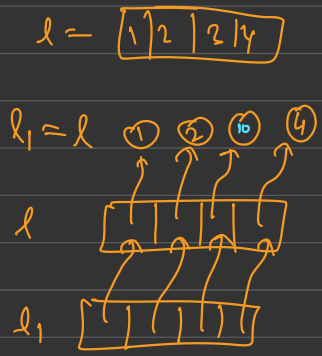
issubclass(): → returns Boolean

↳ syntax: issubclass(class, classinfo)

ABSTRACT BASE CLASSES:

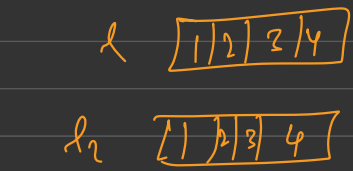
```
from abc import ABCMetaclass

class student(metaclass = ABCMetaclass): #creating abc.

    def __init__(self, name, regno):
        self.name = name
        self.regno = regno
    @abc.abstractmethod
    def set_dept(self):
        pass

class IT student(Student):

    def set_dept(self, dept, name, regno):
        self.dept = dept
        super().__init__(name, regno)
```

Aliasing



Shallow copy

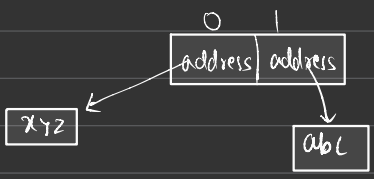
l2 = copy.copy(l)



24/5/23

Array Based Sequences

list = ['xyz', 'abc'] → Storing both address and data.

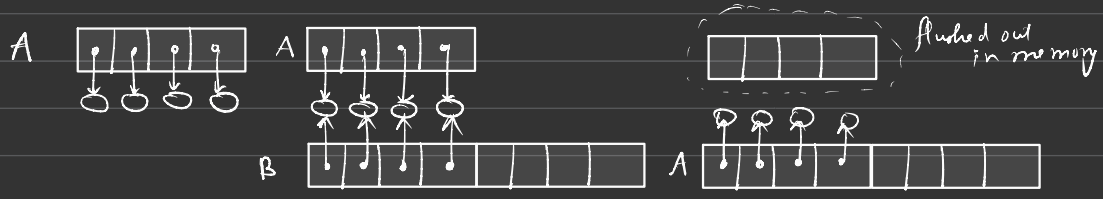


Step ①: fetch the address
Step ②: Then find the value
} Time consuming
2 steps for fetching an element.

To find the size of the list :
↳ import sys → sys.getsizeof (list) → size in bytes.

• Even though a list is empty ⇒ its size is 72 bytes
↳ occupies space.
↳ some amount of memory is consumed.
(Integer → 8 bytes)

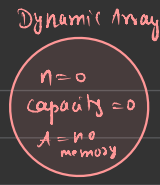
Dynamic Array:



- n → No of actual elements stored in the list.
- capacity → Maximum no of elements that could be stored
- A → reference to the currently allocated array.

Dynamic Array For lab

List ADT using low level arrays.




```
import ctypes

class DynamicArray:
    def __init__(self, val):          # val → size
        """create an empty array"""
        self._n = 0
        self._capacity = val
        self._A = self._make_array(self._capacity)
    if isinstance(val, int):
    else:
        self._n = 0
        self._capacity = len(val)
        self._A = val

    def _make_array(self, cap):
        temp = (cap * ctypes.py_object)()
        return temp
```

Stored here

(during run time it is allocated)

```
def append(self, ele):
    if self._n == self._capacity:
        self._resize(2 * self._capacity)
    self._A[self._n] = ele
    self._n += 1

def _resize(self, cap):
    B = self._make_array(cap)
    for i in range(self._n):
        B[i] = self._A[i]
    self._A = B
    self._capacity = cap

def __len__(self):
    return self._n

def __getitem__(self, index):
    if 0 <= k < self._n:
        raise IndexError("invalid index")
    else:
        return self._A[index]
```

25/5/2023

List ADT:

↳ using array.

Insert & Append:

```
Class Dynamic array:
    def append(self, ele):
        if (self.n == self.cap):
            self.resize(2* self.cap)
            self.A[self.n] = ele
            self.n += 1
    def insert(self, index, ele):
        if not(index <= self.n):
            raise IndexError("Index out of range")
        if self.n == self.capacity:
            self.resize(2* self.capacity)
        for i in range(self.n, index, -1):
            self.A[i] = self.A[i-1]
        self.A[index] = ele
        self.n += 1
```

A = | 3 | 4 | 5 | n=3, c=3

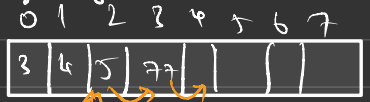
resize: something

| 3 | 4 | 5 | | | |

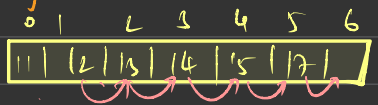
n=4, c=6

Insert:

↳ ele is inserted within 'n'.



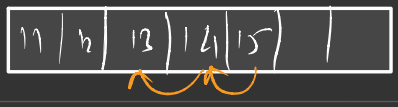
Example:



Index = 1

i = { self.n = 6, 5, 4, 3, 2
index += 1

Shrink only if $n < c//4$ (25%)



$A[i] = A[i+1]$

Deleting:

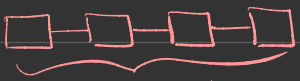
```
def delete(self, index):
    if not (0 <= index < self.n):
        raise IndexError("Index out of range")
    for i in range(index, self.n-1, 1):
        self.A[i] = self.A[i+1]
    self.n -= 1
    if (self.n < self.capacity//4):
        self.resize(self.capacity//2)
```

→ Try extend + __contains__

→ Amortized analysis of Dynamic Arrays

List ADT

List



search
Find
delete
min
max
Insertion.

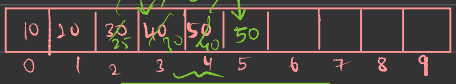
Array Implementation.

Array

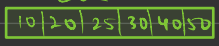
int a[10]; ⇒ 20 bytes memory.

int ⇒ 1 int = 2 bytes

max capacity = 10
n = 4



Insert a[2] = 100;



size = 10
n = 5

```
insert (int a[ ], int index, int val):
{
   for (i=n-1; i >= index; i--):
    { a[i+1] = a[i]
    }
    a[index] = key;
    n = n+1;
}
```

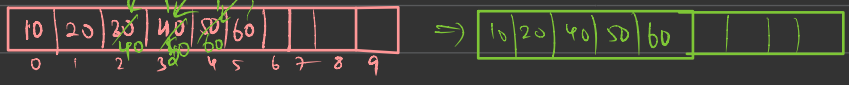
It 1: i = 4 — i >= 2, 4 >= 2, a[5] = a[4]

It 2: i = 3 — 3 >= 2, a[4] = a[3]

It 3: i = 2 — 2 >= 2, a[3] = a[2]

It 4: i = 1 — a[2] = key.

Delete:



n = 6

```
deletion(a[], int x)
{
  for (i=x; i<n-1; i++)
  {
    a[i] = a[i+1];
  }
  n=n-1
}
```

x=2,

It 1: i=2, i<5,
a[2] = a[3]
n=5

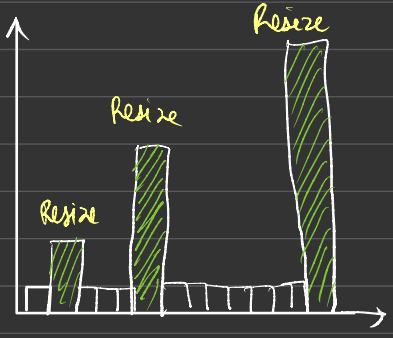
It 2: i=3, 3<5
a[3] = a[4]

It 3: i=4, 4<5
a[4] = a[5]

It 4: i=5, 5<5

n=5

Amortized Analysis of Dynamic Arrays.




```
from time import time
def avg(n):
    data = []
    start = time()
    for k in range(n):
        data.append(None)
    end = time()
    return (end-start)/n
```

$T(n) = 1+2+1+4+1+1+1+8+1+1+1+\dots+16$

$= \frac{2n}{16}$ (finding average basically)

↳ Time complexity

Arithmetic Increase in Capacity:

↳ Adding new empty spaces by addition.

↳ $\Theta(n^2)$

$1+2+3+\dots+n$

$= \frac{n(n+1)}{2}$

Geometric Increase in Capacity:

↳ Adding new empty space by multiplication

↳ $\Theta(n)$

$1 \times 2 \times \dots \times n$

$= n$

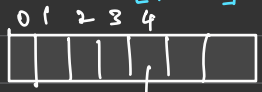
overcharging.

Array Based List ADT

Stack ADT → Last In first Out [LIFO]

Ex: placing notebooks one after the other

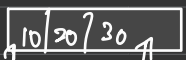
Queue ADT ⟹ Linear ⟹ sequential & referential → First In First Out (FIFO)



top

↓

Int variable which tells where to perform insertion (aka push) & deletion (aka pop)



A[r] = ele
r = r+1

front → deletion (dequeue)

rear → insertion (enqueue) → 2 variables

front = rear = 0

del = A[f]
f = f+1

1) top = 0

2) A[ ]

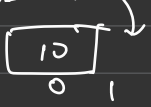
cap = 1, top = 0

3) push(10):

A[top] = 10 | 10

top = top+1

→ top = 1



pop():

del_ele = A[top − 1]

top −= 1

return del_ele

} no. of elements = top

push(20):

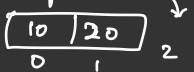
If top == cap:

resize the array

⇒ cap = 2, top = 1

⇒ top = top+1

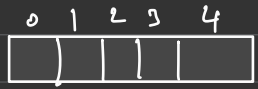
top = 2



Stack ADT:

↳ Linear Data Structure.

↳ Last In first out.



top ⇒ integer.

S = Stack()

Class Stack →
- S.isempty() ⇒ returns boolean
- S.isfull() ⇒ returns boolean
- S.push(ele) ⇒ insertion
- S.pop() ⇒ returns popped ele.
- len(S) ⇒ no. of elements
- print(S)
- S.topelement()

3 WAYS
- Wrapper approach
  - (Already Irukkadha vechi seiradhu)
  - using pre existing python list & making it a stack.
- Dynamic array
- linked.

Wrapper Method:

```
Class Stack:
    def __init__(self):
        self.item = [ ]

    def isempty(self):
        return len(self.item) == 0:

    def push(self, ele):
        self.item.append(ele)

    def pop(self):
        return self.item.pop()

    def __len__(self):
        return len(self.item)

    def __str__(self):
        return str(self.item).

    def topelement(self):
        return self.item[-1]
```

```
def makearray(self, cap):
    temp = (cap * ctypes.py_object)()
    return temp
```

Dynamic Array:

```
class Stack:
    def __init__(self):
        self.cap = cap
        self.top = 0
        self.item = makearray(cap)

    def isempty(self):
        return self.top == 0

    def push(self, ele):
        if self.top == self.cap
            self.resize(2 * self.cap)
        self.item[self.top] = ele
        self.top += 1

    def pop(self):
        if(self.isempty()):
            raise Empty('Stack is empty')
        ele = self.item[self.top - 1]
        self.item[self.top-1] = None
        self.top -= 1
        if (self.top < self.cap//4):
            self.resize(self.cap//2)
        return ele
```

user defined exception

```
class Empty(Exception):
    pass
```

```
def __len__(self):
    return self.top
```

```
def __str__(self):
    return str(self.item)
```

```
def topelement(self):
    return self.item[self.top-1]
```

```
def resize(self, cap):
    B = self.makearray(cap)
    for i in range(self.cap-1):
        B[i] = self.item[i]
    self.item = B
    self.cap = cap
```

↳ First in First Out

↳ Front, Rear.

↳ Integer

↳ Front → deletion

Rear → Insertion

↳ Initially, front = rear = 0

Class Queue:

↳ Q = Queue()

↳ Q.isempty() ⇒ front = rear

↳ Q.enqueue(ele)

↳ Q.dequeue()

↳ len(Q) ⇒ (rear - front)

↳ print(Q)

↙ Wrapper method

↓ Dynamic Array

↘ Linked.

Lab ex - 9

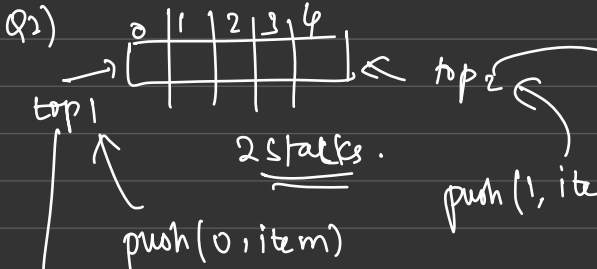
Palindrome:

```
n = 121
temp = n
rn = 0
while n > 0:
    r = n % 10
    rn = rn x 10 + r
    n = n // 10
if rn == temp:
    print("Palindrome")
```

→ Basic method.


```
while:
    if pop == deque:
        "palindrome"
```

Q2)



Compact arrays

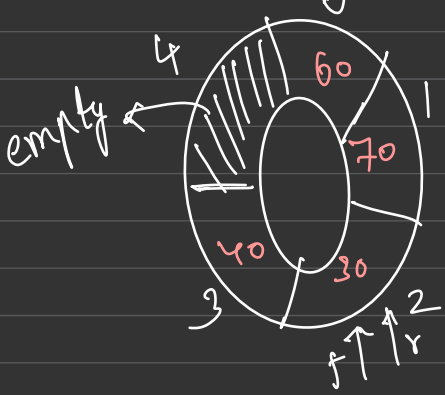
if top1 > top2 or top2 < top1:
Stack full

Queue.

Circular Queue

↓

compact array implementation.



cap = 5

front = rear = 0

enqueue(10)

rear = 1

dequeue(10)

front = 1

∴ cap = 5 ⇒ only 4 elements can be enqueued into circular queue.

```
class CircularQueue:
    def __init__(self, cap):
        self.cap = cap
        self.item = makearray(cap)
        self.rear = self.front = 0

    def next(self, p):
        return (p+1) % self.cap

    def isEmpty(self):
        return (self.front == self.rear)

    def isFull(self):
        return (self.rear+1) % self.cap == self.front

    def enqueue(self, ele):
        if (self.isFull()):
            raise Full
        else:
            self.item[self.rear] = ele
            rear = next(rear)

    def dequeue(self):
        if isempty():
            raise empty:
        else:
            del self.item(self.front)
            self.front = next(front).
```

Instead of rear = rear + 1
use rear = (rear+1) % cap

class Full(Exception):
    pass

Infix and Postfix evaluation:

Infix: a + b * c

↳ operators b/w operand.

a + b * c

convert to postfix expression

a(bc*)+

Infix: a * b / c

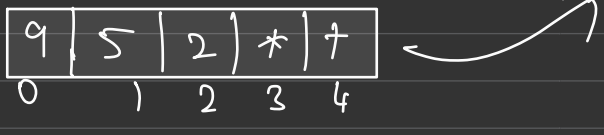
(ab*)c/

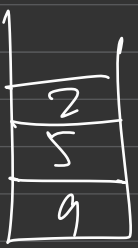
Implement

How to evaluate:

a(bc*)+

a = 9, b = 5, c = 2, *, +





operand: push into stack

operator: pop 2 from stack.

b = 2 pop 1

a = 5 pop 1

a * b = 10

c = 10

⇒ 10 + 9 ⇒ 19

pop = 19

INFIX TO POSTFIX CONVERSION RULES:

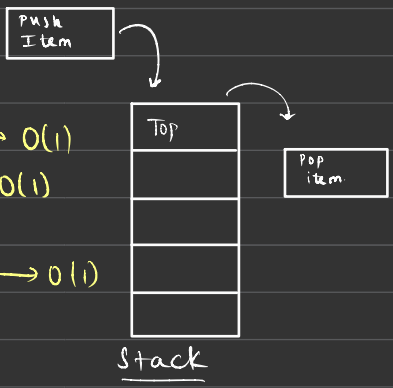
1. Print the operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If incoming symbol is '(', push it onto stack.
4. If incoming symbol is ')', pop the stack & print the operators until left parenthesis is found.
5. If incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If incoming symbol has lower precedence than the top of the stack, pop and print the top. Then test the incoming operator against the new top of the stack.
7. If incoming operator has equal precedance with the top of the stack, use associativity rule.
8. At the end of the expression, pop & print all operators of stack.

⇒ associativity L to R then pop & print the top of the stack & then push the incoming operator

⇒ R to L then push the incoming operator

Stacks

- Insertion and deletion → same element
- Last In First Out (LIFO)



Basic Operations:

- push() → Insert an element into the stack → O(1)
- pop() → remove an element from the stack → O(1)
- top() → returns the top of an element
- isEmpty() → returns True if empty else False → O(1)
- size() → returns size of stack → O(1)

ALGORITHM For Basic Operations:

- push():

```
if stack is full:
    return
else:
    increment top
    stack[top] assign value
```

- pop():

```
if stack is empty:
    return
else:
    store value of stack[top]
    decrement top
    return value
```

- top():

```
return stack[top]
```

- isEmpty():

```
if top < 1:
    return true
else:
    return false
```

- **Infix Expression to Postfix expression:**

→ Infix Expression: The expression of the form "a operator b" (a+b)
"when an operator is in btwn every pair of operands"

→ Postfix Expression: The expression of the form "a b operator" (ab+)
"when every pair of operands is followed by operator"

Example: Input: A+B*C+D
output: A(BC*)+D+

How to convert Infix Expression to Postfix expression?
Using Stacks.

Linked List:

Seq list: 0 | 1 | 2 — 10 | 20 | 30

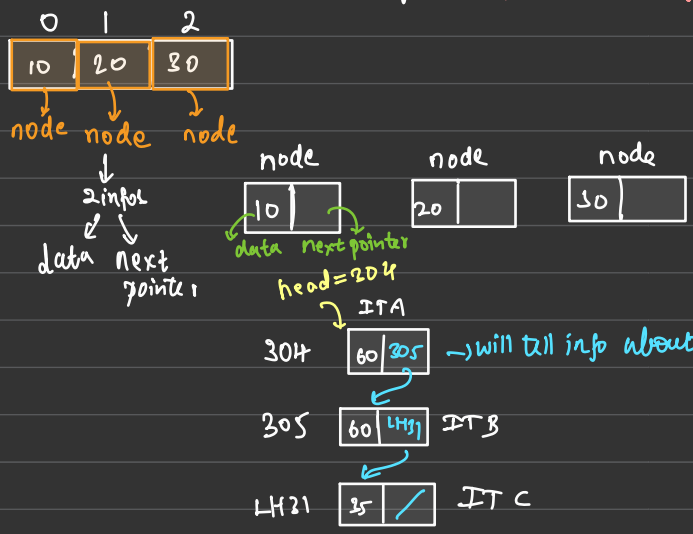
↳ Compact array

↓

Dynamic Array

Drawback:

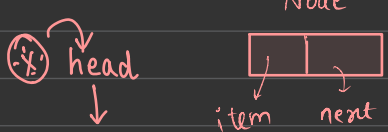
- Wastage of memory.
- Continous Memory allocation not possible
- Inserting / Deleting. (Lot of data shifting)



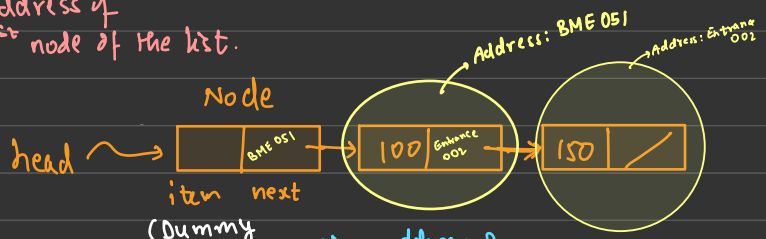
- You can't do indexing in Linked List.

Linked List

Node is created during runtime
↓
(biggest advantage)



Address of 1st node of the list.



(Dummy node) → gives address of first node

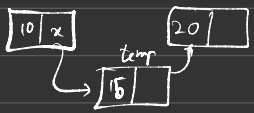
Driver code:

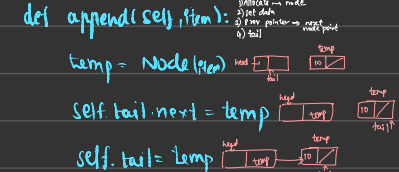
```
S = Singlylinkedlist()
S.append(100)
S.append(150)
S.display()
S.find(150)
```

```
Class Node:
    __slots__ = ['item', 'next']
    def __init__(self, item=None, next=None):
        self.item = item
        self.next = next
```

⇒ nodecreation.py

```
from nodecreation import Node
class Singlylinkedlist:
    def __init__(self):
        self.head = self.tail = Node()
    def isempty(self):
        return (self.head == self.tail)
    def append(self, item):
        temp = Node(item)
        self.tail.next = temp
        self.tail = temp
```



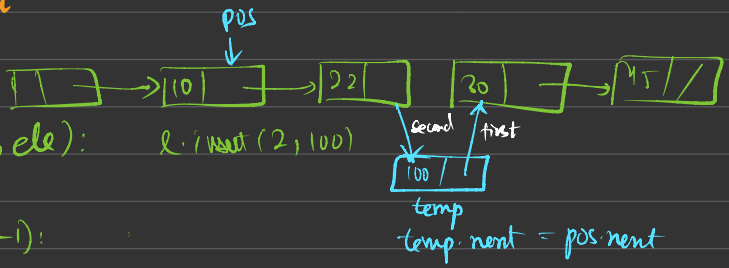


```
def display(self):
    pos = self.head.next
    while (pos != None):
        print(pos.item)
        pos = pos.next
```

Explanation:
Since we have dummy node, we are starting from address of next node and assigning it as pos. pos.item will give the data. So we are printing (pos.item) till the pos becomes None (last node).

```
def find(self, ele):
    pos = self.head.next
    while (pos != None):
        if (pos.item == ele):
            return pos
        pos = pos.next
    return None
```

can perform only linear search (Drawback)




```
def insert(self, position, ele):
    pos = self.head.next
    for i in range(position-1):
        pos = pos.next
    temp = Node(ele, pos.next)
    pos.next = temp
```

l.insert(2, 100)

temp.next = pos.next

```
def remove(self, ele):
    prev = self.find_prev(data)
    delnode = prev.next
    prev.next = delnode.next
```
(won't work for last node)

```
def find_prev(self, data):
    pos = self.head.next
    while (pos.next is not None):
        if (pos.next.item == data)
            return pos
        pos = pos.next
    return None
```

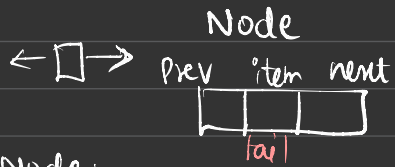
```
def __str__(self):
    st = '['
    pos = self.head.next
    while pos != None:
        str += ',' + str(pos.item)
        pos = pos.next
    st += ']'
    return st
```

```
def __init__(self):
    self.size = 0   along with self.head = self.tail = 0
```

increment in insertion

decrement in deletion

Doubly Linked List:



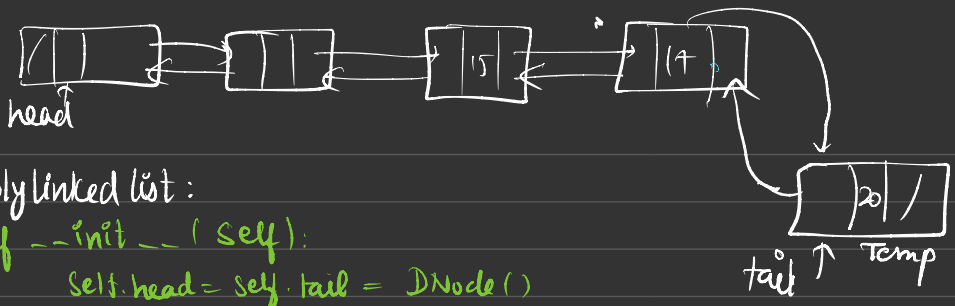

```
def __init__(self, item=None, next=None):
    self.item = item
    self.next = next
```

inheriting

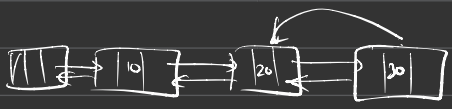
```
class DNode(Node):
    __slots__ = ['prev']
    def __init__(self, item, next, prev=None):
        super().__init__(item, next)
        self.prev = prev
```

Methods:

1) isempty
2) append(ele)
3) Display
4) Reverse display
5) Find
6) Insert
7) Remove



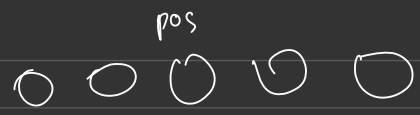

```
class Doubly linked list :
    def __init__ ( self ):
        self.head = self.tail = DNode()
        self.size = 0

    def isempty (self):
        return (self.head == self.tail)

    def append (self, ele):
        temp = DNode(ele)
        self.tail.next = temp
        temp.prev = self.tail
        self.tail = temp
        self.size += 1

    def display(self):
        pos = self.head.next
        while (pos != None):
            print(pos.item)
            pos = pos.next
```

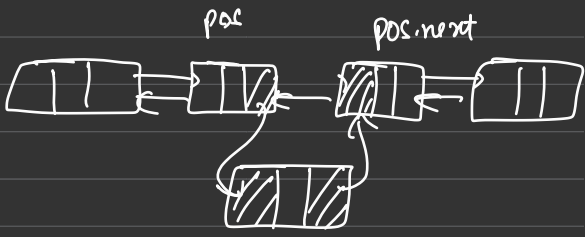


pos = self.tail

```
    def Reverse_display (self):
        st = '['
        pos = self.tail
        while pos != None :
            st += str(pos.item) + ','
            pos = pos.prev
        st += ']'
        return st
```

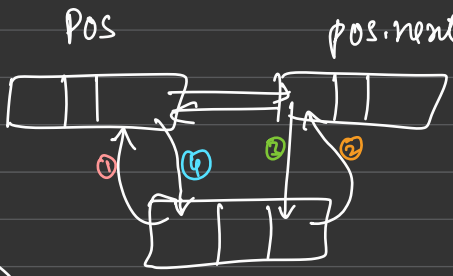
```
def find(self, ele):
    pos = self.head.next
    while (pos != None):
        if (pos.item == ele):
            return pos
        pos = pos.next
    return None
```





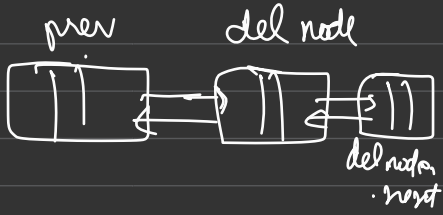

```
def insert(self, position, ele):
    if pos.next == None:
        self.append(ele)
    else:
```

```
temp.prev = pos
temp.next = pos.next
pos.next.prev = temp
pos.next = temp
```



① temp.prev = pos
② temp.next = pos.next
③ pos.next.prev = temp
④ pos.next = temp

```
def remove (self):
    prev = self. find_prev_data(data)
    del node = prev. next
    prev. next = del node. next
    del node. next. prev = prev
    self. size -= 1
```

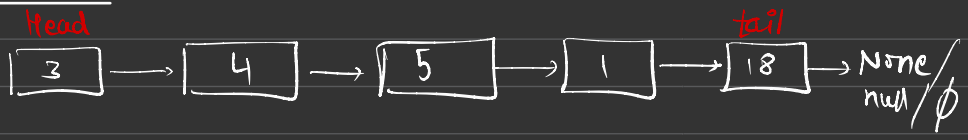



```
def reverse(self):
    curr = self. head. next
    prev_node = None
    while curr:
        next_node = curr. next
        curr. next = prev_node
        curr. prev = next_node
        prev_node = curr
        curr = next_node
```

```
self. head, self. tail = self. tail, self. head
```

Kunal Kushwaha

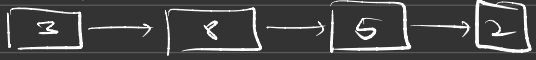
Linked List:




```
Class Node {
    int val;
    Node next;
}
```

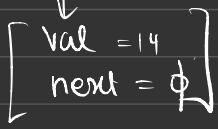
- head

Insert At First

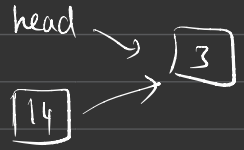


Add 14 in first place

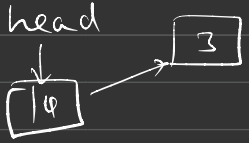
node



node.next = head



head = node




```
if tail == None:
    tail = head
```

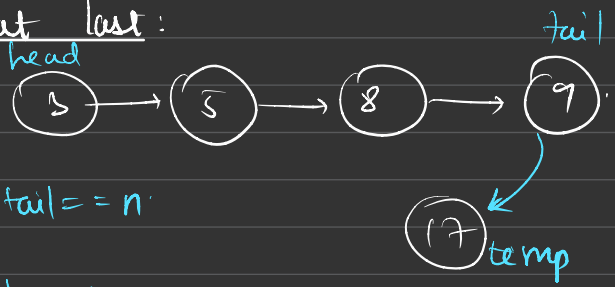
- To display:

```
while(head != ϕ):
    print(head.val)
    head = head.next
```

(temp = head) ⇒ because head shouldn't move.

like pos = self.head.next

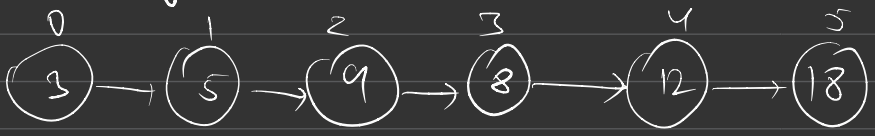
- **Insert at last:**



if tail == n:

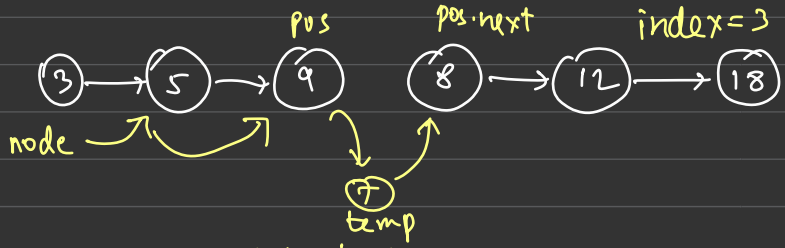
tail.next = temp
tail = temp
size += 1

- **Insert anywhere:**



if index == 0 ⇒ call insertAtFirst()
if index == last ⇒ call insertAtLast()

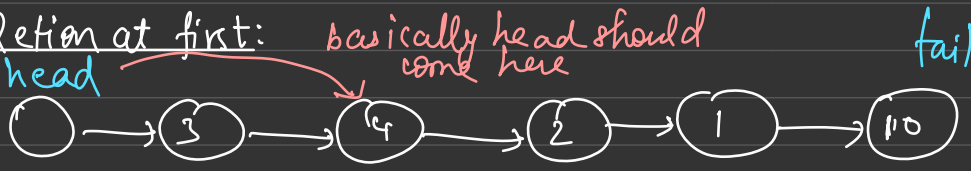
if in btwn:
↳ insert at 3 (for example)



pos.next = temp
temp.next = pos.next

Deletion:

- **Deletion at first:**



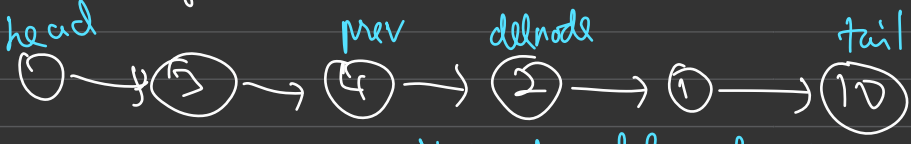

```
remnode = self.head.next
self.head.next = remnode.next
```

- **Deletion at last:**


```
pos = self.head
for i in range ( self.size -1):
    pos = pos.next
pos.next = None
self.tail = pos
```

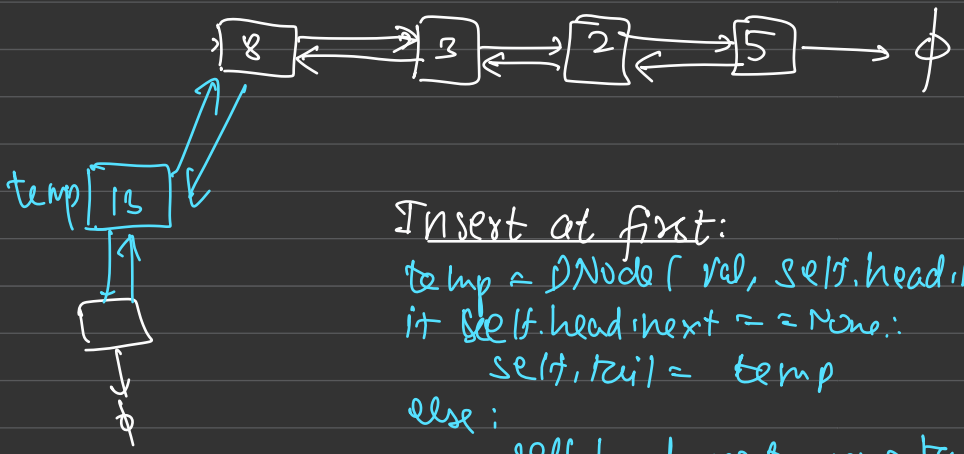
- **Deletion anywhere:**

delete 2.



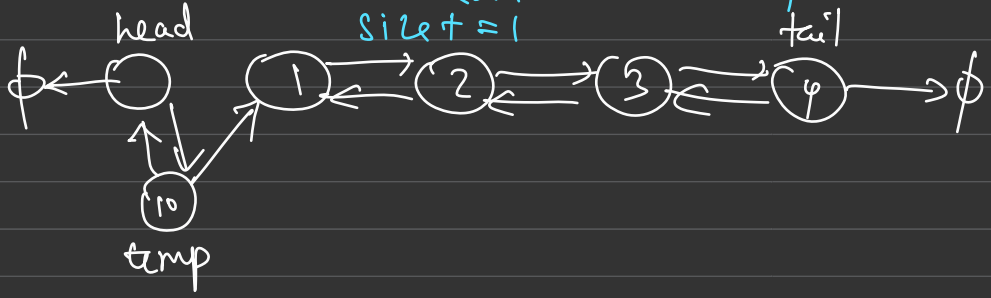

```
prev.next = delnode.next
```

Doubly linked list:



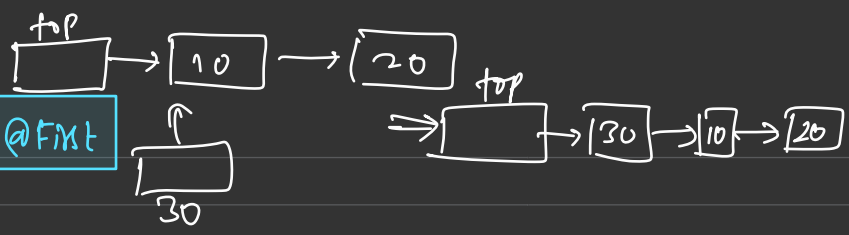
Insert at first:

```
temp = DNode( val, self.head.next, self.head)
if self.head.next == None:
    self.tail = temp
else:
    self.head.next.prev = tail
self.head.next = temp
```



- ### Reverse display:


```
pos = self.tail
while ( pos != None ):
    print(pos.item)
    pos = pos.prev
```



Insert @ First Delete @ First

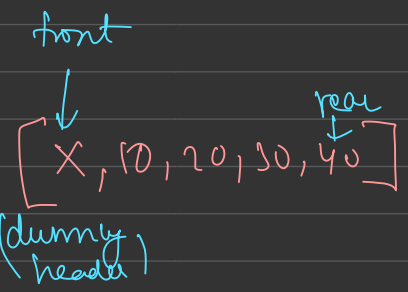
Linked Stack:

```
class Node:
    __slots__ = ['item', 'next']
    def __init__(self, item=None, next=None):
        self.item = item
        self.next = next

class LinkedStack:
    __slots__ = ['top']
    def __init__(self):
        self.top = Node()
        self.size = 0
    def isempty(self):
        return (self.top.next == None)
    def top(self):
        if self.isempty():
            raise Empty("stack empty")
        return self.top.next.item
    def push(self, ele):
        temp = Node(ele)
        temp.next = self.top.next
        self.top.next = temp
        self.size += 1
    def pop(self):
        delnode = self.top.next
        self.top.next = delnode.next
        self.size -= 1
    def __str__(self):
        out = ""
        pos = self.top.next
        while (pos != None):
            out += str(pos.item) + "->"
            pos = pos.next
        out += "END"
        return out
```

Insert at Last, delete at First

Linked Queue:

```
__slots__ = ['front', 'rear']
def __init__(self):
    self.rear = self.front = Node()
    self.size = 0
def isempty(self):
    return self.front == self.rear
def enqueue(self, val):
    temp = Node(val)
    self.rear.next = temp
    self.rear = temp
    self.size += 1
def dequeue(self):
    delNode = self.front.next
    Self.front.next = delNode.next
    self.size -= 1
def front(self):
    return self.rear.item
def __str__(self):
    out = ""
    pos = self.front.next
    while (pos != None):
        out += str(pos.item) + " → "
        pos = pos.next
    out += "END"
    return out
```

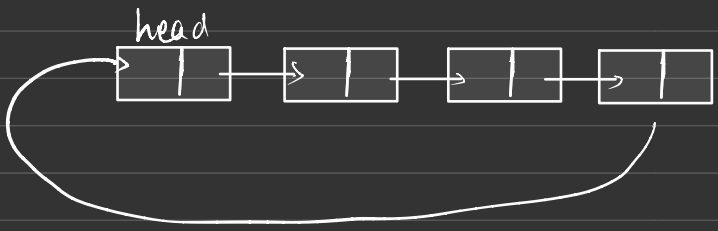


Time Complexity for Linked List:

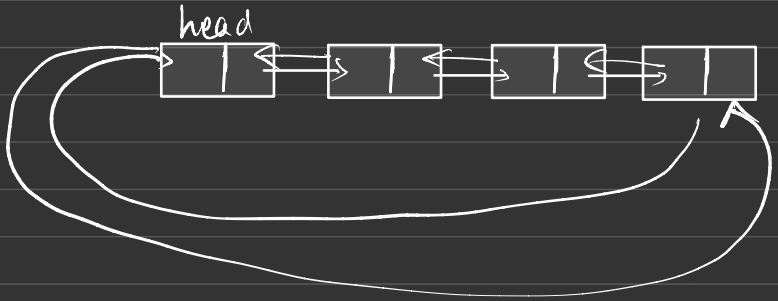
OPERATIONS	SINGLY LINKED LIST	DOUBLY LINKED LIST
• append	O(1)	O(1)
• insert	O(n)	O(n)
• remove	O(n)	O(n)
• display	O(n)	O(n)
• isempty	O(1)	O(1)
• find	O(n)	O(n)
• pop	O(1)	O(1)

Circular Linked List:

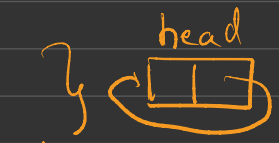
Circular Singly Linked List:

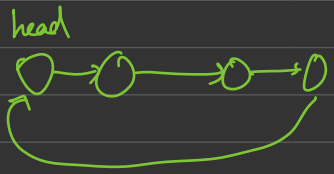


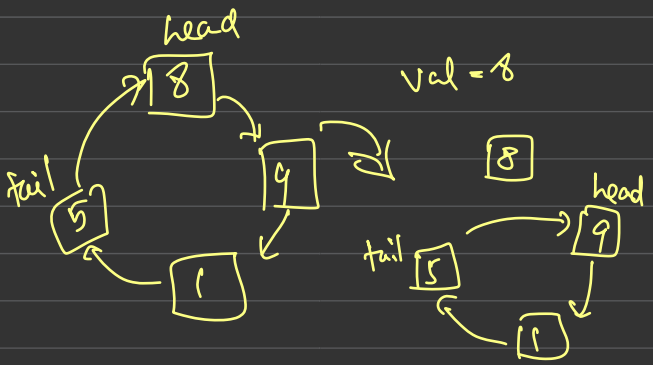
Circular Doubly Linked List:



```
def __init__(self):
    self.head = Node()
    self.tail = self.head
    self.head.next = self.head
    self.size = 0
def isempty(self):
    return self.head.next == self.head
def append(self, ele):
    temp = Node(ele, self.head)
    self.tail.next = temp
    self.size += 1
def display(self):
    pos = self.head.next
    first = str(pos.item)
    out = " "
    while (pos != self.head):
        out += str(pos.item) + "<->"
        pos = pos.next
    out += first
    return out
def delete(self, val):
    curr = self.head.next
    n = curr.next
    if curr.item == val:
        head = head.next
        tail.next = head.
        return
    for i in range(self.size):
        if curr.item == val:
            curr.next = n.next
```



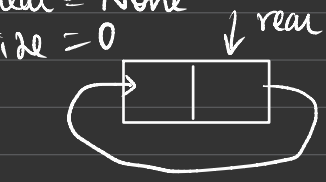


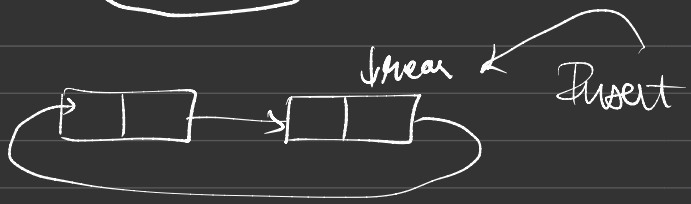


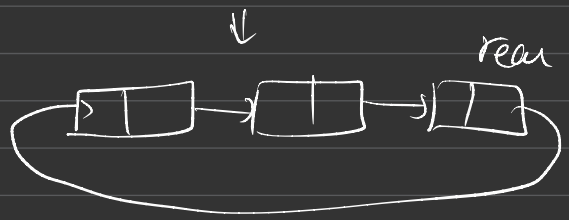
Circular Queue using Singly Linked List:

- No dummy header
- self.rear = None
- self.size = 0

No need self.head.



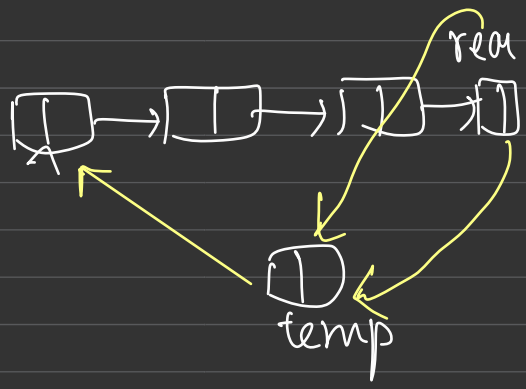




Rotate:

↳ dequeue & immediately enqueue.

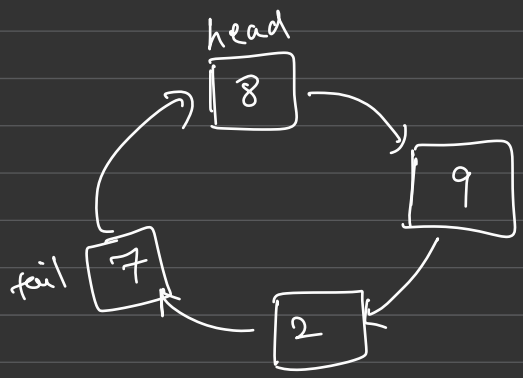
```
class Node:
    __slots__ = ['item', 'next']
    def __init__(self, item = None, next = None):
        self.item = item
        self.next = next

class CircularQueueLL:
    def __init__(self):
        self.rear = None
        self.size = 0
    def enqueue(self, ele):
        temp = Node(ele)
        if self.size == 0:
            temp.next = temp
            self.rear = temp
        else:
            temp.next = self.rear.next
            self.rear.next = temp
            self.rear = temp
        self.size += 1
    def dequeue(self):
        temp = self.rear.next
        self.rear.next = temp.next
    def rotate(self):
        self.rear = self.rear.next
```



(Or)

```
self.rear
= self.rear.next
```

Circular Linked List:



class Node remains same

Insert after tail:

```
temp = Node(ele)
tail.next = temp
temp.next = head
tail = temp
```

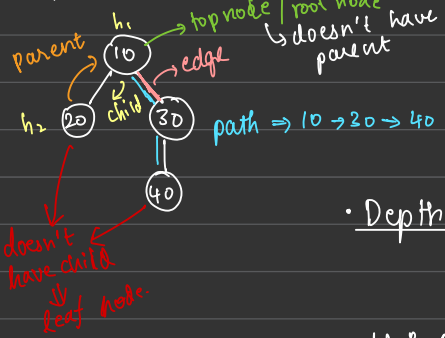
```
if head == Node:
    head = temp
    tail = temp
```

19/6/23 UNIT 4 – Non Linear DS.

Trees: General Trees:

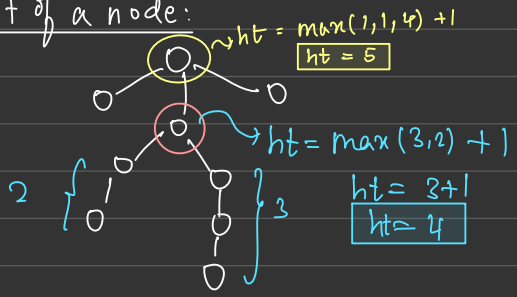
↳ Stores element hierarchially.

↳ Except for top element, each element in a tree has a parent element and zero or more children element.



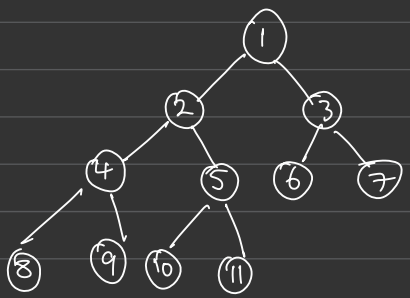
- **Depth of a node:**
  - → For Root node : 0
  - → Depth of 30 : 1
  - → Depth of 40 : 2

- **Height of a node:**



Siblings:
↳ elements having same parent.

Ordered Trees:

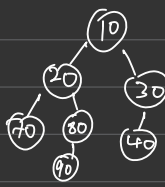


TREE Abstract DATA Type:

class ↖ p.element(): Return the element stored at position p.

Tree class ↖ T.root(): Return the address of root of T.

T.is_root(p): Return True / False after checking if root / not.



T.parent(p): Return parent's address of p

T.num_children(p): Return no. of children of p

T.Children(p): Addresses of all children

T.is_leaf(p): Return True / False

len(T): Return total no of nodes of T

T.is_empty(): Return True / False

T.positions(): Address of all nodes of T

iter(T): Generate an iteration of all elements stored within tree T

Abstract Tree:

@ Abstract Method
- root()
- print(pos)
- num_children(pos)
- children(pos)
- __len__()

concrete method
- isRoot(pos)
- isleaf(pos)
- isEmpty() → no root node.
- depthN(pos)
- heightN(pos)
- height() → ht of root node. (entire tree)
- __iter__() → iterate over all elements.
- positions()
- preorder()
- preordersubtree()

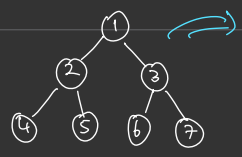
Abstract Binary Tree:

@abstract method
- left(pos)
- right(pos)

concrete method:
- children(pos)
- sibling(pos)

Linked Binary Tree [Using Linked List]
- addRoot(item)
- addLeft(item, pos)
- addRight(item, pos)
- replace(item, pos)

Binary Tree:



0 1 2 3 4 5 6

1	2	3	4	5	6	7

parent index = child's index // 2

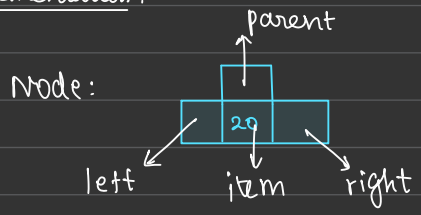
Abstracttree.py

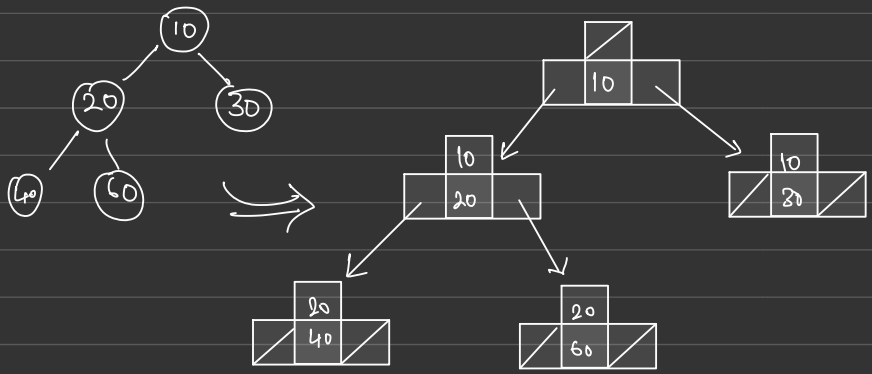
```
from abc import ABC, abstractmethod
class AbstractTree(ABC):
    @abstractmethod
    def root(self):
        pass

    @abstractmethod
    def parent(self, pos):
        pass

    @abstractmethod
    def num_children(self, pos):
        pass

    @abstractmethod
    def children(self, pos):
        pass

    @abstract method
    def __len__(self):
        pass

    def isRoot(self, pos):
        return self.root() == pos

    def isleaf(self, pos):
        return if self.num_children(pos) == 0:

    def isEmpty(self):
        return self.root() == None
                (or)
        return len(self) == 0
```

```
def depthN(self, pos):
    if self.isRoot(pos):
        return 0
    else:
        return depthN(self.parent(pos)) + 1

def heightN(self, pos):
    if self.isLeaf(pos):
        return 0
    else:
        return max(heightN(child) for child in self.children(pos)) + 1

def height(self, pos = None):
    if pos == None:
        if self.isempty() != True:
            return -1
        pos = self.root()
    return self.heightN(pos)
```

```
from Abstracttree.py import AbstractTree
class AbstractBinaryTree(AbstractTree):
    @abstractmethod
    def left(self,pos):
        pass
    @abstractmethod
    def right(self,pos):
        pass
    def children(self,pos):
        if pos is None:
            return None
        if self.left(pos) is not None:
            yield self.left(pos)
        if self.right(pos) is not None:
            yield self.right(pos)
    def sibling(self,pos):
        parent = self.parent(pos)
        if parent is None:
            return None
        if pos == self.left(parent):
            return self.right(parent)
        else:
            return self.left(parent)
```

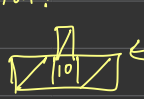
Linked Implementation:



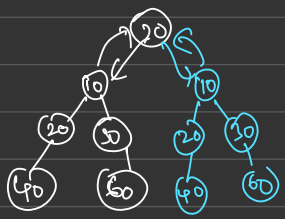


```
class LinkedBinaryTree(AbstractBinaryTree):
    class BTNode:
        __slots__ = ['item', 'left', 'right', 'parent']
        def __init__(self, item, left=None, right=None, parent=None):
            self.item = item
            self.left = left
            self.right = right
            self.parent = parent
        def getitem(self):
            return self.item
        def setitem(self, ele):
            self.item = ele
    __slots__ = ['_root', 'size']
    def __init__(self, item=None, TLeft=None, TRight=None, parent=None):
        self._root = None
        self.size = 0
        if item is not None:
            self._root = self.addRoot(item)
        if TLeft is not None:
            if TLeft.root is not None:
                TLeft._root.parent = self._root
                self._root.left = TLeft._root
                self.size += TLeft.size
                TLeft._root = None
                TLeft.size = 0
        if TRight is not None:
            if TRight._root is not None:
                TRight.parent = self._root
                self._root.right = TRight._root
                self.size += TRight.size
                TRight._root = None
                TRight.size = 0
```

function:




```
def addRoot(self, item)
    if self._root is not None:
        raise ValueError("Root already exist")
    self._root = BTNode(item)
    self.size += 1
    return self._root
```




```
# Driver code:
l = LinkedBinaryTree(10)

# driver code
T1 = LinkedBinaryTree()
T2 = LinkedBinaryTree(10)
T3 = LinkedBinaryTree(20, T2)
```

↳ adding a subtree to tree

```
def __len__(self):
    return self.size

def parent(self, pos):
    return pos.parent

def left(self, pos):
    return pos.left

def right(self, pos):
    return pos.right

def root(self):
    return self._root
```

TRAVERSALS:

1) preorder (fab) ⇒ root → left → right
2) postorder (ab-) ⇒ left → right → root
3) Inorder (a+b) ⇒ left → root → right

```
def preOrder(self, pos):
    res = f"{pos.item}"
    if pos.left is not None:
        res = self.preorder(pos.left)
    if pos.right is not None:
        res = self.preorder(pos.right)
    return res

def __str__(self):
    return self.preorder(self.root)
```

```
def addLeft(self, item, pos):
    if self.left(pos) is not None:
        raise ValueError('  ')
    pos.left = BTNode(item, parent = pos)
    self.size += 1
    return pos.left

def addRight(self, item, pos):
    if self.right(pos) is not None:
        raise ValueError('  ')
    pos.right = BTNode(item, parent = pos)
    self.size += 1
    return pos.right
```

Class Test

1) constructor of Linked Binary Tree
2) Preorder traversal.

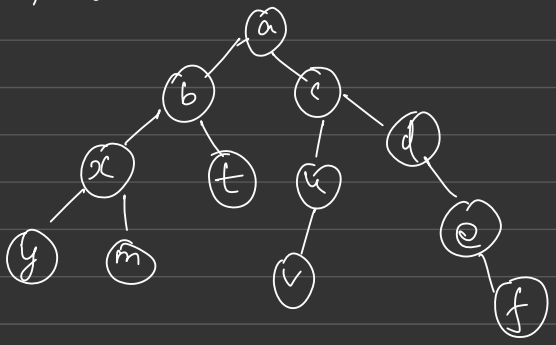
```
class LinkedBinaryTree:
    class BTNode:
        __slots__ = ['item', 'left', 'right', 'parent']
        def __init__(self, item, left=None, right=None, parent=None):
            self.item = item
            self.left = left
            self.right = right
            self.parent = parent

    def getitem(self):
        return self.item

    def setitem(self, ele)
        self.ele = item.

    __slots__ = ['_root', 'size']
    def __init__(self, item=None, Tleft=None, Tright=None, parent=None)
        self._root = None
        self.size = 0
        if item is not None:
            self._root = addRoot(item)

def addRoot(self, item):
    if self._root() is not None:
        raise ValueError("")
    self._root = self.BTNode(item)
    self.size += 1
    return self._root

    if Tleft._root is not None:
        Tleft._root.parent = self._root
        self._root.left = Tleft._root
        self.size += 1
        Tleft._root = None
        Tleft.size = 0

    if Tright._root is not None:
        Tright._root.parent = self._root
        self._root.right = Tright._root
        self.size += 1
        Tright._root = None
        Tright.size = 0
```

2) Preorder

```
def preorder(self, pos):
    res = f"{pos.item}"
    if pos.left is not None:
        res += self.preorder(pos.left)
    if pos.right is not None:
        res += self.preorder(pos.right)
    return res
```

---

3) Given a tree

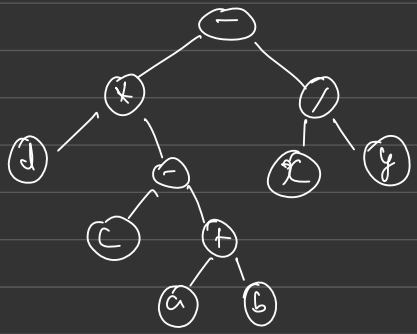


Preorder: a b x y m t c u v d e f

Inorder: y x m b t a v u c f e d

Postorder: y m x t b v u f e d c a

Expression Tree:



Preorder: - * d - c + a b / x y

Inorder: d * c - a + b - x / y

Postorder: d c a b + - * x y / -

Expression tree:

ab+c- ⇒ string

- If operand ⇒ create node ⇒ push it in stack
- operator ⇒ create a node ⇒ pop 2 values from stack
  - 1st pop ⇒ add as right child
  - 2nd pop ⇒ add as left child
  - push into stack
- last value ⇒ pop from stack ⇒ root.

Expression Tree

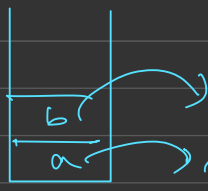
Abstract Tree
↓
Abs Binary Tree
↓
Linked Binary Tree

- Binary Tree
- Internal nodes - operator
- External nodes - operand

Input: ab+c*

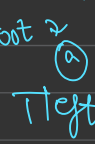
Str ⇒ | a | b | + | c | * |

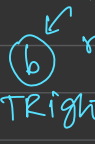
If operator ⇒ pop from stack

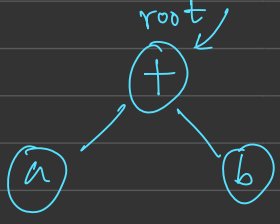


stack

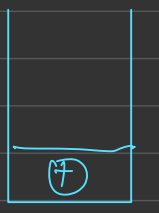
If Operand ⇒ create a node



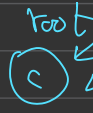




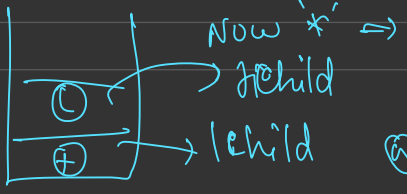
Push address of (+) to stack



Now 'c' ⇒ create a node

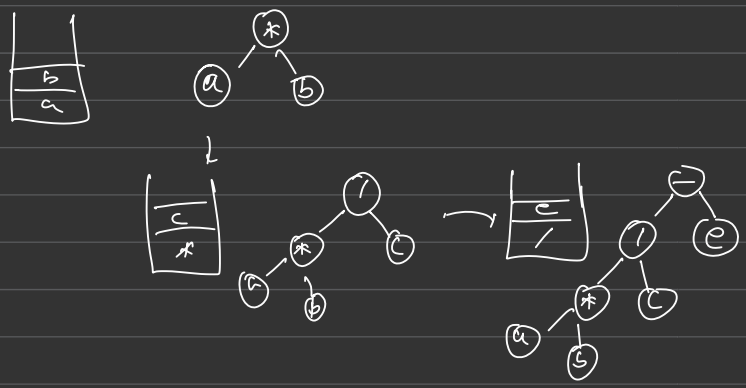


& push into stack



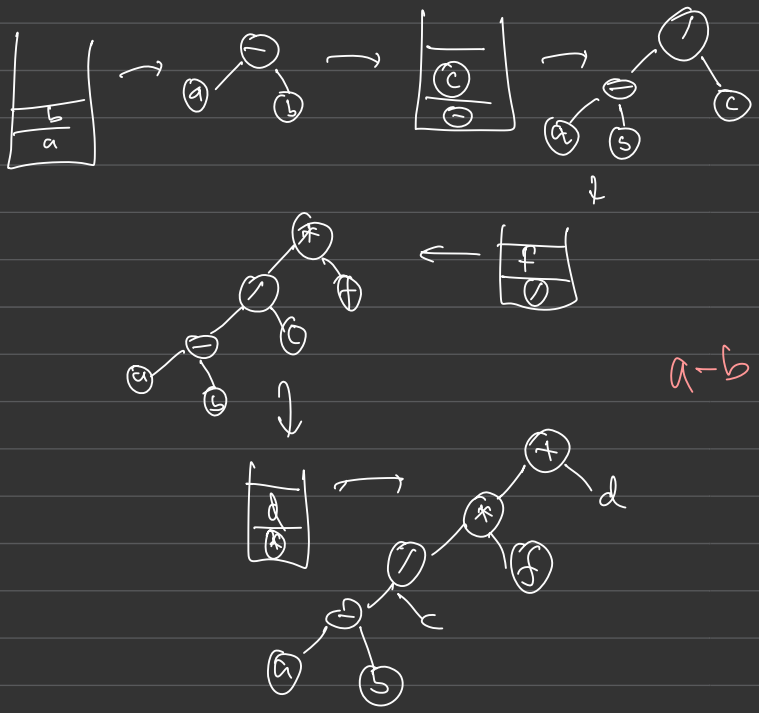
```
Class ExpressionTree(LinkedBinaryTree):
    def __init__(self, item = None, Tleft = None, Tright = None):
        super().__init__(item, Tleft, Tright)

    def construct(self, str):
        S = []
        for ch in str:
            if ch in '+*/-':
                rchild = S.pop()
                lchild = S.pop()
                S.append(ExpressionTree(ch, lchild, rchild))
            else:
                S.append(ExpressionTree(ch))
        return S.pop()
```

Input: ab * c/e -



a*b/c-e

Input : ab - c / f * d +



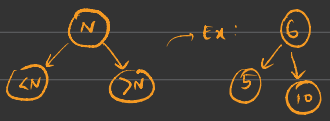
Mirror a tree:

```
def mirror(self, pos):
    if pos.left is not None:
        mirror(pos.left)
    pos.item
    mirror(pos.right)
```

```
def inorder(self, pos):
    if( pos.left is not None ):
        str += inorder(pos.left)
    print(pos.item)
    if (pos.right is not None):
        return inorder ( pos.right)
```

TREES:

Pre Requisites:
- ① Recursion.
- ② OOPS.

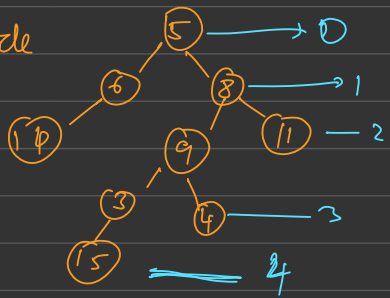
Why:
- Effective time complexity for add / remove
  - → O(log n)
- 

- cost efficient (no restructuring)

Limitation:
- unbalance binary tree ⇒ O(n) for searching
- inefficient for sorted Elements

BT Node: (item, right, left)

Properties:
- size = no of nodes
- child & parent
- sibling
- edge
- height → Max no of edges from the node & leaf nodes.
- leaf nodes.
- level → ht diffn b/w that node & root node
  - root level = 0



1/7/2023

HASHING

↳ technique to make search const time O(1)

likes SNID ← key | value → name/ph no

	key	value
0	10	-
1	20	-
2	25	-
3	32	-
4	63	-

hash(10)

↳ key % 10

↳ 25 % 10 ⇒ 5

↳ store 25 on 5th index instead of 2nd index.

could be anything.

index = key % 10

0	10
1	
2	32
3	63
4	
5	25

hash(key) → returns index value.

Insert (key, value) → Hash(key) → key | value

Find (key)

Purpose: Hash function → map a key to an index.

Hash function → Hash code | compression function

Hash code: non integer key value to int

↓

will take a string & give integer as output (32 bit integer)

compression function: ↓ (32 bit integer)

takes in input from hash code & converts it a legal integer code b/w the index.

hash function: Key % tablesize

Multiply - Add - and - Divide (MAD) function:

↳ hash code i is compressed to an index by

(32 bit integer) $[(ai + b) \bmod p] \bmod \text{Tablesize}$ ⇒ Compression function

'p' → prime no. larger than Tablesize

a, b → random numbers from interval [0, p-1] with a > 0.

How do we generate hash code from arbitrary key?

→ could be a complex expression

↳ pass it into compression func

↳ get the compressed value.

Example: "stop", "tops", "pots", "spot"

tablesize = 10007 , strings are 8 ch long

Worst case: Ascii value of each ch = 127

total ch = 8

⇒ Max^m index = 127 × 8 = 1016

⇒ remaining space is wasted

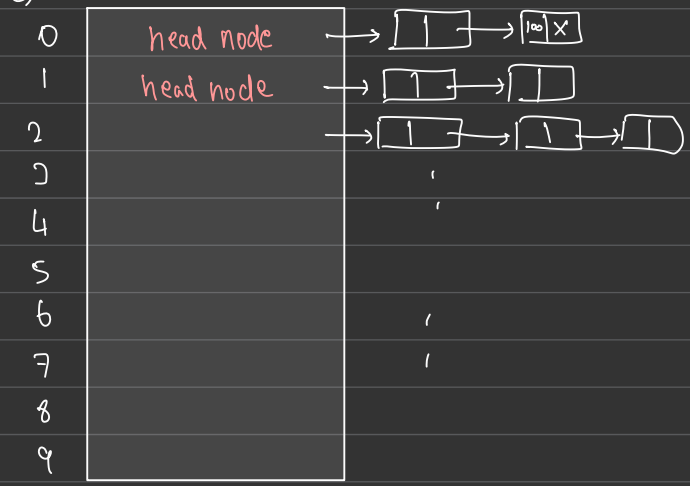
⇒ change the hash code

Changes: key = "stop", key[0] = ascii of s, key[1] = ascii of t

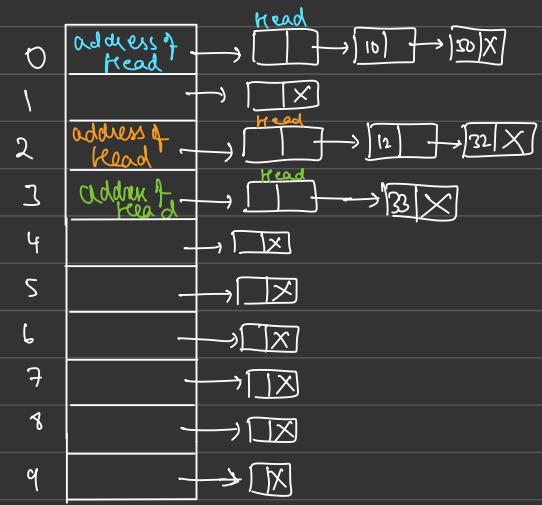
↳ key[0] + 27 * key[1] + 27^2 key[2]

COLLISION HANDLING TECHNIQUE

- Separate Chaining (open hashing)
- Closed hashing.



EX: 33, 10, 12, 50, 32,



Load factor(λ): Ratio of no. of pairs to the Table size
↳ Average length of linked list.

We can also use BST

Search time increases if more collision at same index.

Closed Hashing: using the empty spaces in the array.

↳ $h_i(key) = (Hash(key) + F(i)) \% Tablesize$.
- tells how many index from 'key' should I place the new key.
- Collision resolution probing strategy.

Linear Probing: $F(i) = i$

Insert: 89, 18, 49, 58, 69.

(K - key)

some fn of K
eg: h(K) = 2K+3

$u = h(K) \% Tablesize$

Linear probing:
↳ first free location from $(u+i) \% Tablesize$
where $i = 0$ to $(m-1)$

0	49
1	58
2	69
3	8
4	22
5	75
6	14
7	
8	18
9	89

(using the empty spaces)

storing at closest, empty location

no of key = Table size

- Apply formula
- Check if colliding
- if colliding → linear probing.

Search: 22
↳ Find mod
22 % 10 ⇒ 2 from 2nd index do linear search to find 22.

Analysis of Linear Probing:

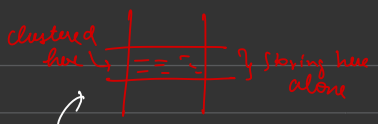
↳ drawback: "primary clustering"

$\lambda = \frac{\text{no. of keys}}{\text{tablesize}}$

expected no. of (probes) = $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$

(probes) = no. of linear search

If $\lambda = 0.5$
⇒ no of probes = 2.5



Quadratic Probing:

↳ to eleminate primary clustering

↳ use $F(i) = i^2$ is a popular choice.

$$h_i(key) = (Hash(key) + F(i))\ \%\ Tablesize.$$

If $F(i) = i^2$

$$h_i(key) = (Hash(key) + i^2)\ \%\ Tablesize$$

Deletion:

0	49
1	58
2	69
3	8
4	32
5	75
6	14
7	
8	~~18~~ X
9	89

using the empty spaces

status: deleted

delete 18.

Now search 58

58 % 10 ⇒ 8 ⇒ do linear search from 8 but 8 has no element

⇒ Linear search not possible.

∴ Perform LAZY DELETION.

now to store 28.

insert K_i at first free location from $(u + i^2)\ \%$ Tablesize where $i = 0$ to $(m-1)$

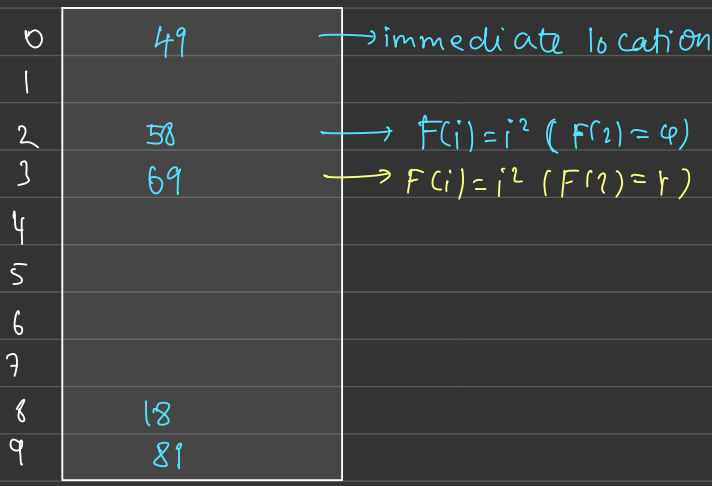
where $u = h(K)\ \%$ Tablesize

where $h(K) = 2K + 3$ (for an example)

↳ can be any fn of K.

Quadratic probing:

Insert: 89, 18, 49, 58, 69.



we are checking if we can store @ i=1, if yes → store

if not → i=2 → if yes → store

if not i = 3 → if yes → store

if not ...

Drawback: not utilising empty spaces

Double Hashing:

↳ $F(i) = i \cdot Hash_2(key)$

index = $((hash(key) + i \times hash_2(key)) \% 10)$

where

$Hash_2(key) = R - (key \% R)$

↳ Prime no within table size

Ex:

Insert 89, 18, 49, 58, 69

R = 7 here

0	69
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

49: i = 0 ⇒ collision, i = 1

(hash(49) + i × 7) % 10

(9 + 7) % 10

6

58: i = 1, i = 0 ⇒ collision

((hash(58) + i × (5)) % 10

(8 + 5) % 10 ⇒ (3)

69: i = 1 for i = 0 ⇒ collision

(9 + 1) % 10 ⇒ 0

Rehashing:

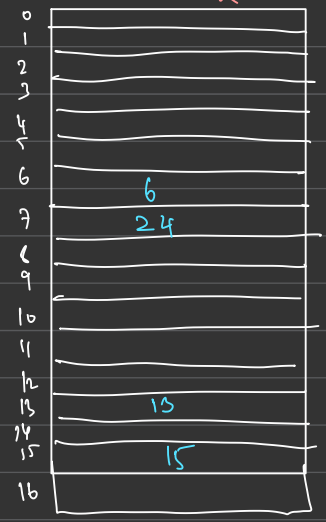
↳ Dynamically varying Tablesize.

6	15		24			13

$\lambda = 0.5$

Insert 23 ⇒ Hash table is >70 % full

⇒ Increase table to next Prime Number.



Rehashing → mod with newsize.

(6 % 17) ⇒ 6

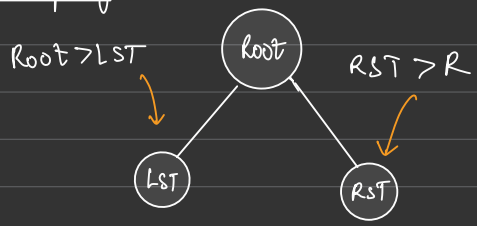
(15 % 17) ⇒ 15

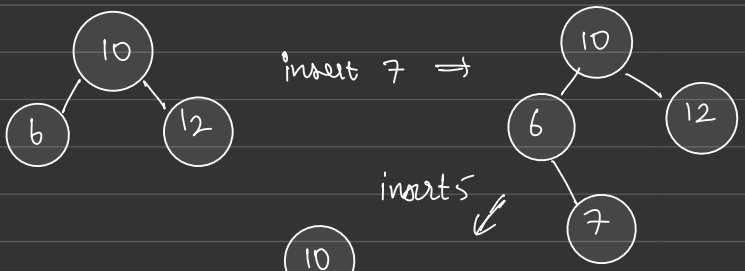
(24 % 17) ⇒ 7

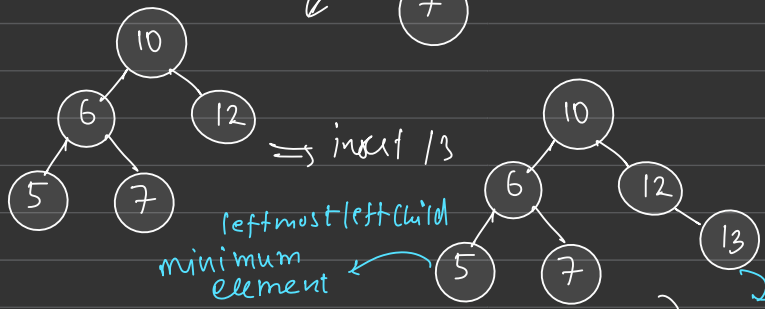
(13 % 17) ⇒ 13

Binary Search Tree:

Ordering Property:







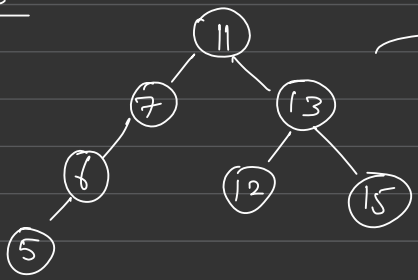
inorder ⇒ sorted numbers ⇒ search time optimised.
(constant time) O(n)

Inorder: 5, 6, 7, 10, 12, 13

```
Class BinarySearchTree(LinkedBinaryTree):
    def __init__(self, item=None, Tleft=None, Tright=None):
        super().__init__(item, Tleft, Tright)

    def insert(self, pos, item):
        if pos is None:
            self.addRoot(item)
        elif (item < pos.item):
            if pos.left is None:
                pos.left = self.addLeft(pos, item)
            else:
                self.insert(pos.left, item)
        elif (item > pos.item):
            if pos.right is None:
                pos.right = self.addRight(pos, item)
            else:
                self.insert(pos.right, item)
        elif item == pos.item:
            return

def search(self, pos, item):
    if item == pos.item:
        return pos
    elif item < pos.item:
        if pos.left is None:
            return None
        else:
            return self.search(pos.left, item)
    elif item > pos.item:
        if pos.right is None:
            return None
        else:
            self.search(pos.right, item)
```

```
def find_min(self, pos):
    if pos.left is None:
        return pos
    else:
        return self.find_min(pos.left)

def find_max(self, pos):
    if pos.right is None:
        return pos
    else:
        return self.find_max(pos.right)
```

Delete:



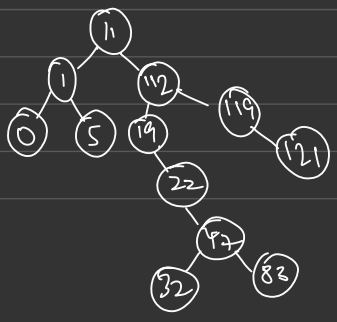
deleting node has:
(i) no child
(ii) one child
(iii) two child

(code in 2 MS)

HW:

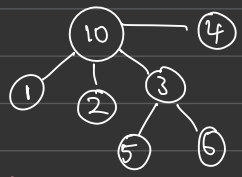
11, 1, 5, 112, 19, 22, 47, 83, 119, 121, 0, 32

Delete: 83, 0, 22



0, 1, 5, 11, 19, 22, 32, 47, 83, 112, 119, 121

29/6/23 - Thursday



Breadth First Search:

↳ Go level by level

1st from root (10) → (1) → (2) → (3) → (4)

then children

⇒ So tho non-linear ⇒ search not optimised just like linear search.

Binary Search Tree:

↳ TO improve the search time complexity (to reduce)

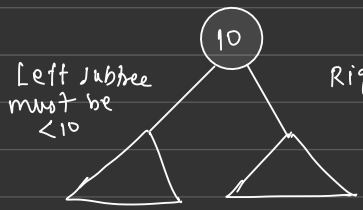
↳ It is not a general tree → also a binary tree.

BST

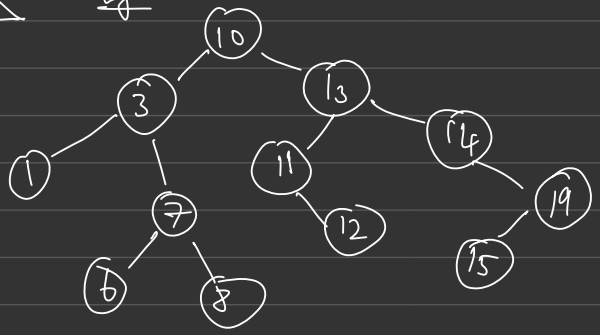
↓

ordering property

(needs to be maintained)



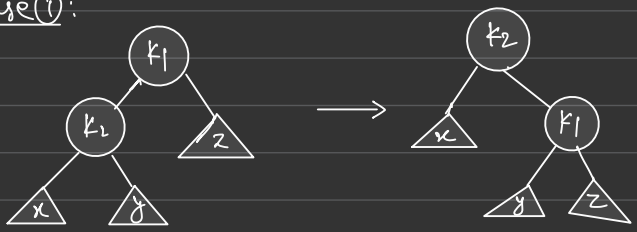
Eg:



- Each subtree also (ap art from rest) must maintain ordering property then only BST.
- This way search is retrived in linear amount of time.
- Challenge

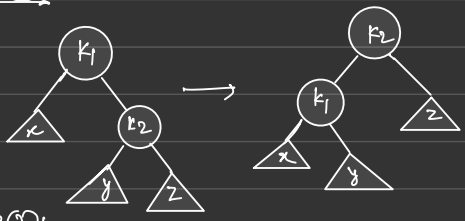
AVL Trees: (Self Balancing tree)

Case (1):



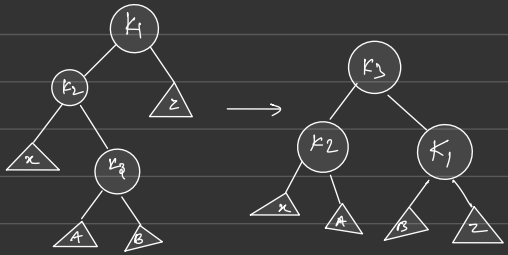
Insertion of LC of LST

Case (2):



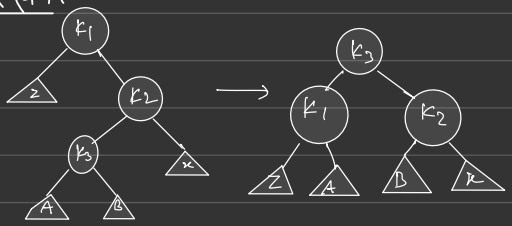
Insertion of RC of RST.

Case (3):



Insertion of RC of LST

Case (4):



Insertion of LC of RST

Balancing Property:

↳ Balance factor = Height LST − Height RST.

(+1, −1, 0)

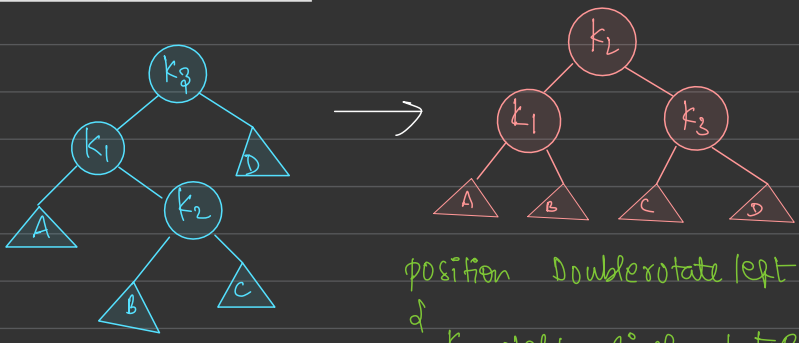
↳ or else not balanced.

Pseudocode

Single rotate left:

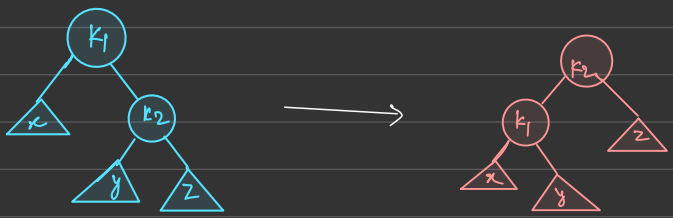

```
position singlerotateleft (Position k2),
{
    Position k1;
    k1 = k2 -> left
    k2 -> left = k1 -> right
    k1 -> right = k2
    k2 -> height = max( height(k2 -> left), height(k2 -> right)) + 1
    k1 -> height = max( height(k1 -> left), height(k1 -> right)) + 1
    return k1
}
```

Double Rotate left:




```
position Doublerotate left
{
    k3 -> left = singlerotateRight ( k3 -> left)
    return singlerotate Left(k3)
}
```

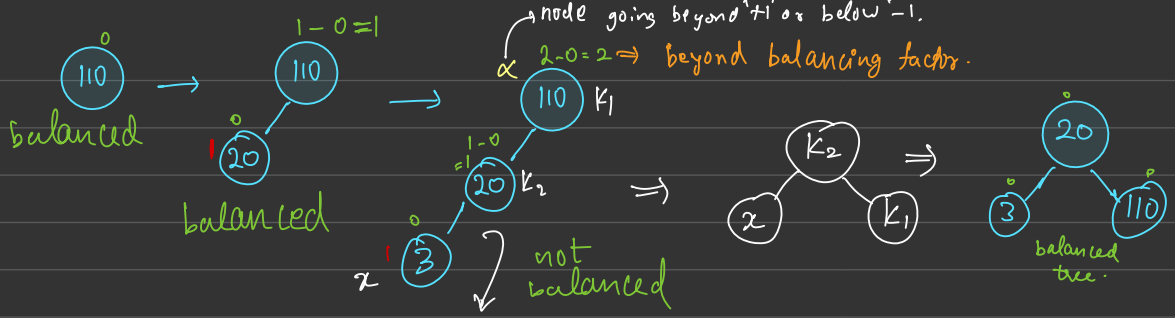
Singlerotate right.



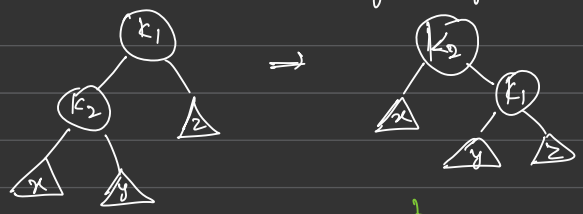

```
position singlerotateright (position k1)
{
    position k2;
    k2 = k1 -> right
    k1 -> right = k2 -> left
    k2 -> left = k1
    k1 -> height = max(height(k1 -> left), height(k1 -> right)) + 1
    k2 -> height = max(height(k2 -> left), height(k2 -> right)) + 1
    return k2
}
```

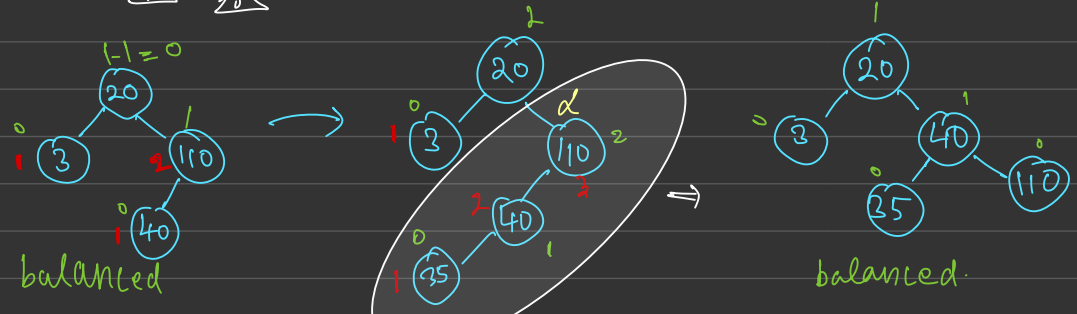
110, 20, 3, 40, 35, 125, 38, 36.

• → balance factor
• → height

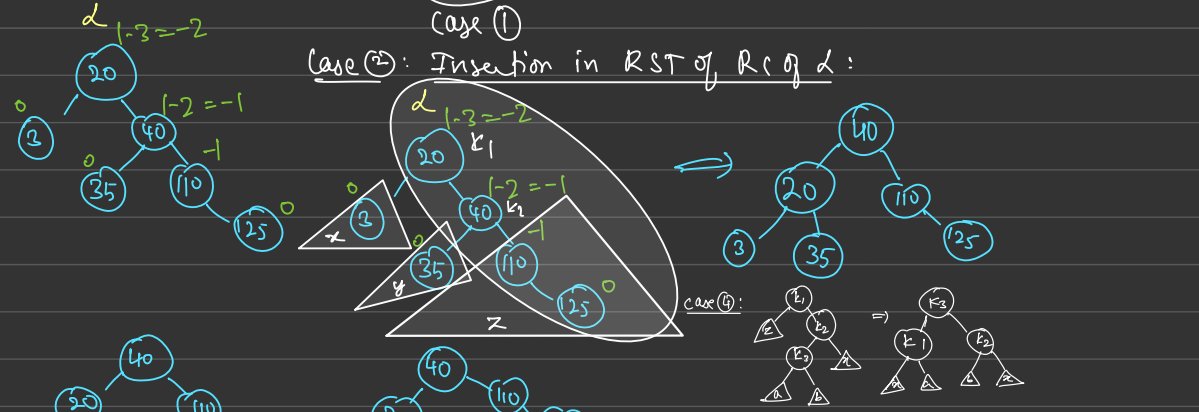


Case ①: Insertion in LST of LC of α (Single rotation)

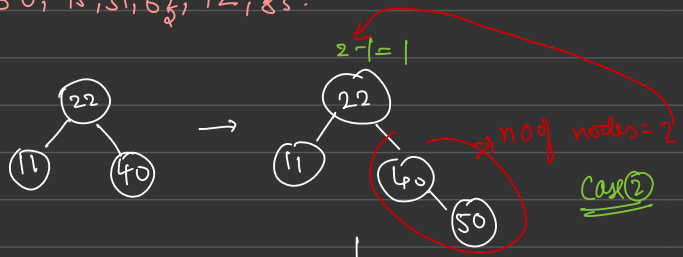


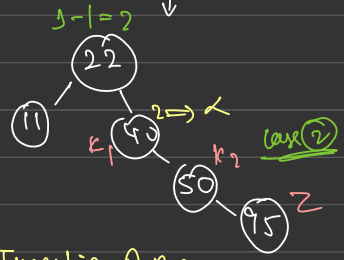


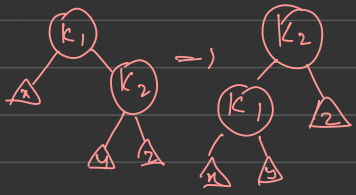
case ①



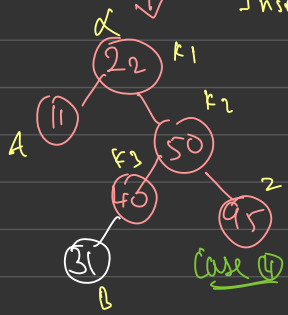
22, 11, 40, 50, 95, 31, 63, 72, 83.

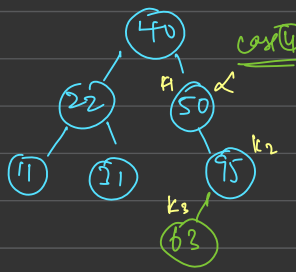


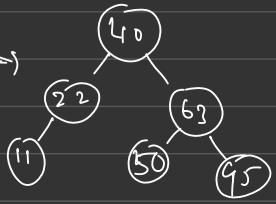


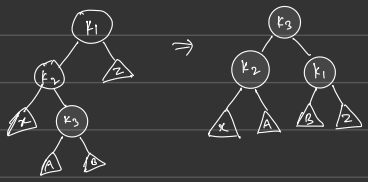


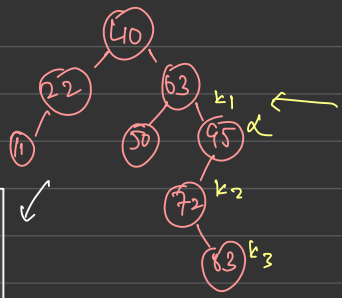
Insertion of R of RST ⇒ case 2

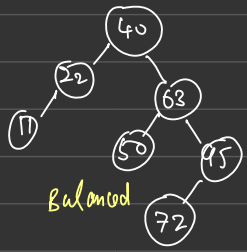


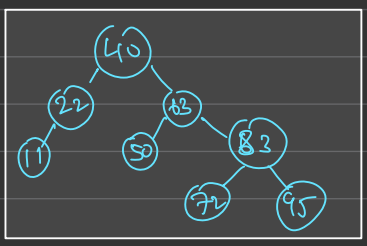






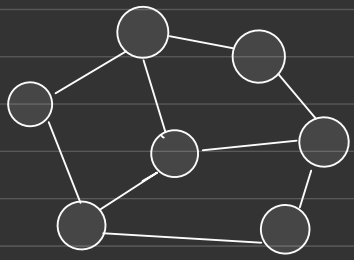




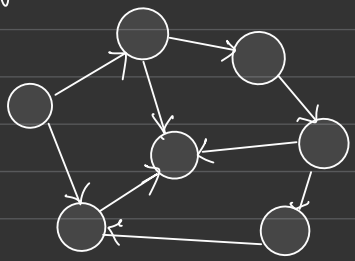


UNIT-5 GRAPHS

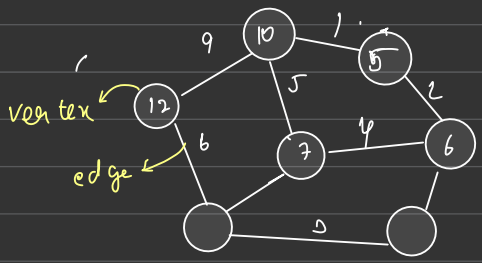
undirected graph:



directed graph:



weighted graph:

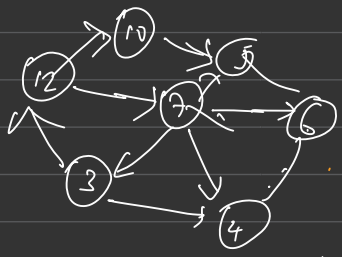


path : 12 to 6

↳ 12 - 10 - 5 - 6

REPRESENTATION OF GRAPHS:

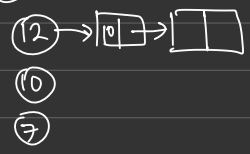
1. Adjacency Matrix
2. Adjacency list.



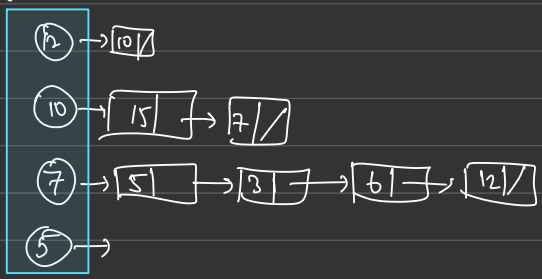
① Adjacency matrix:

	12	10	5	6	3	4
12	X	9	0	0	0	0
10	0	X	3	0	0	0
5			X			
6				X		
3					X	
4						X

⇒ Drawback:
↳ more no. of zeroes.



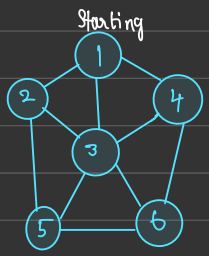
② Adjacency list:



TRAVERSAL OF GRAPHS:

↳ BREADTH First Search (Queue)

↳ Depth First Search. (Stack)

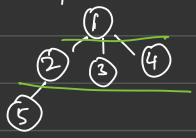


BREADTH FIRST:

↳ First find adj nodes of '1':

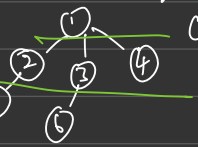
Queue: [2, 3, 4]

↳ In the queue, FIFO. so find adj nodes of '2'



Q: [3, 4, 5]

↳ Next is adj nodes of '3'



Vovoru level oda nodes mudiru apuram remaining

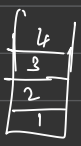
Q: [4, 5, 6]

Q: [5, 6]

Q: [6]

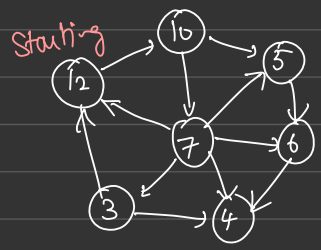
Q: [ ]

DEPTH FIRST: (Stack)

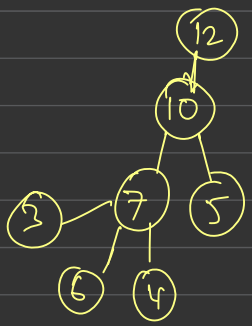


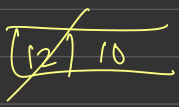
nikkama poite iru. Here there is no adj nodes.

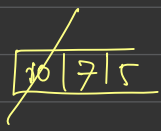
Ex:

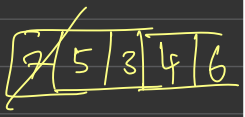

Starting
12
10
5
7
6
3
4

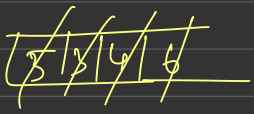
BFS:


12
10
7
5
3
6
4


12
10

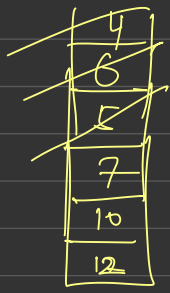

10
7
5

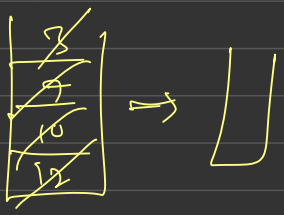

7
5
3
4
6


5
3
4
6

DFS


12
10
7
3
5
6
4

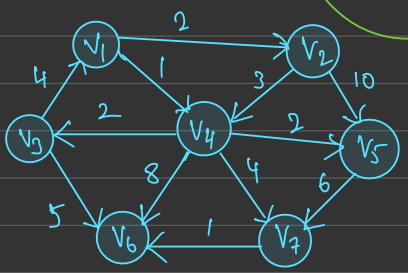

4
6
5
7
10
12


3
5
10
12

DIJKSTRA ALGORITHM

"Shortest Path Algorithm"

→ Single source shortest path Algorithm.



How to Solve:

	Known	distance d_v	Path$_v$
Source → V_1	~~F~~ T	0	—
V_2	~~F~~ T	~~∞~~ 2	V_1
V_3	F T	~~∞~~ 3	V_4
V_4	F T	~~∞~~ 1	V_1
V_5	F T	~~∞~~ 3	V_4
V_6	F T	~~∞~~ ~~9~~ ~~8~~ 6	~~V_4~~ ~~V_3~~ V_7
V_7	F T	~~∞~~ 5	V_4

u — v (adjacent nodes)

V_1 — V_2, V_4

$d_u + C_{uv} < d_v$

- $u = V_1$, $V = V_2$

  $d_{V_1} + C_{V_1V_2} < d_{V_2}$

  $0 + 2 < \infty$

  $2 < \infty$ ⇒ True.

- $u = V_1$, $V = V_4$

  $0 + 1 < \infty$ ⇒ True

u	V
V_4	V_3 V_5 V_6 V_7

- $u = V_4$, $V = V_3$

  $d_{V_4} + C_{V_4V_3} < d_{V_3}$.

  $1 + 2 < \infty$

  $3 < \infty$

- $u = V_4$, $V = V_5$

  $1 + 2 < \infty$

  $3 < \infty$

- $u = V_4$, $V = V_6$

  $1 + 8 < \infty$

- $u = V_4$, $V = V_7$

  $1 + 4 < \infty$

  $5 < \infty$

u	V
V_2	V_4 V_5

- $u = V_2$, $V = V_4$

  $d_{V_2} + C_{V_2V_4} < d_{V_4}$.

  $2 + 3 < 1$

  $5 < 1$ X False

  (no need to update)

- $u = V_2$, $V = V_5$

  $2 + 10 < 3$

  $12 < 3$

  (no change)

u	V
V_3	V_6

$u = V_3$, $V = V_6$

$3 + 5 < 9$

$8 < 9$ ⇒ true

u	V
V_5	V_7

$u = V_5$, $V = V_7$

$3 + 6 < 5$

→ False (no change)

u	V
V_7	V_6

- $5 + 1 < 8$ ⇒ true

Final table

	Known	d_v	$Path_v$
V_1	T	0	—
V_2	T	2	V_1
V_3	T	3	V_4
V_4	T	1	V_1
V_5	T	3	V_4
V_6	T	6	V_7
V_7	T	5	V_4

Path:

↳ $V_3 - V_4 - V_1 \Rightarrow cost = 3$

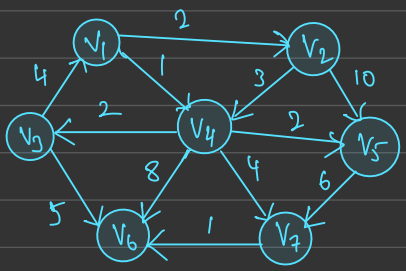
$V_4 - V_1 \Rightarrow cost = 1$

$V_2 - V_1 \Rightarrow cost = 2$

$V_5 - V_4 - V_1 \Rightarrow cost = 3$

$V_6 - V_7 - V_4 - V_1 \Rightarrow cost = 6$

$V_7 - V_4 - V_1 \Rightarrow cost = 5$



Source = V_4.

	Known	d_v	Path$_v$
V_1	~~F~~ T	~~∞~~ 6	V_3
V_2	~~F~~ T	~~∞~~ 8	V_1
V_3	~~F~~ T	~~∞~~ 2	V_4
V_4	~~F~~ T	0	—
V_5	~~F~~ T	~~∞~~ 2	V_4
V_6	~~F~~ T	~~8~~ ~~7~~ 5	~~V_4~~ ~~V_3~~ V_7
V_7	~~F~~ T	~~∞~~ 4	V_4

u	v
V_4	V_6 V_7 V_3 V_5

- $u = V_4, v = V_6$
  $0 + 8 < \infty$
  $8 < \infty$
- $u = V_4, v = V_7$
  $0 + 4 < \infty$
- $u = V_4, v = V_3$
  $0 + 2 < \infty$
- $u = V_4, v = V_5$
  $0 + 2 < \infty$

u	v
V_3	V_1 V_6

- $u = V_3, v = V_1$
  $2 + 4 < \infty$
  $6 < \infty$
- $u = V_3, v = V_6$
  $2 + 5 < 8$

u	v
V_5	V_7

- $u = V_5, v = V_7$
  $2 + 6 < 4$

u	v
V_7	V_6

- $u = V_7, v = V_6$
  $4 + 1 < 7$
  $5 < 7$

u	v
V_1	V_4 V_2

- $u = V_1, v = V_4$
  $6 + 1 < 0$
- $u = V_1, v = V_2$
  $6 + 2 < \infty$

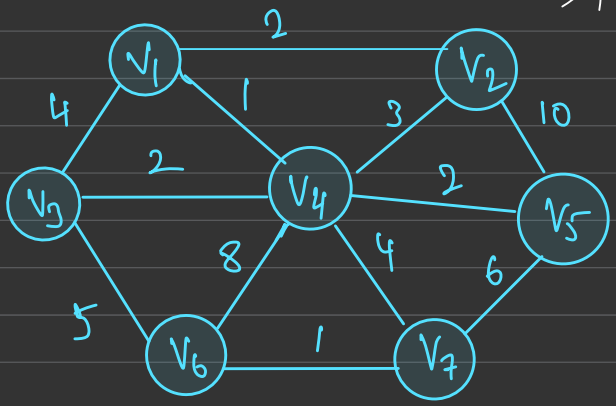
u	v
V_2	V_4, V_5

- $u = V_2, v = V_5$
  $8 + 10 < 2$
  no change.

Path:

↳ $V_1 - V_3 - V_4 \Rightarrow$ cost = 6
↳ $V_2 - V_1 - V_3 - V_4 \Rightarrow$ cost = 8
↳ $V_3 - V_4 \Rightarrow$ cost = 2
↳ $V_5 - V_4 \Rightarrow$ cost = 2
↳ $V_6 - V_7 - V_4 \Rightarrow$ cost = 5
↳ $V_7 - V_4 \Rightarrow$ cost = 4

MINIMUM SPANNING TREE: → Prims
→ Krushkals



same dijkstra table
but use
this equation: $C_{uv} < d_v$

BINARY HEAPS / PRIORITY QUEUE

Heap property.

Operations:
- insert (follow heap property)
- delete min O(1)

Max Heap
parent > child

Min Heap
parent < child

insert → go up
delete → go down

→ can delete only root element
↳ so take the element to delete to the root.

BINARY SEARCH TREE: (Youtube : Brian Faure)

```
Class Node:
    def __init__(self, value = None):
        self.value = value
        self.left_child = None
        self.right_child = None

Class binary_search_tree:
    def __init__(self):
        self.root = None
    def insert(self, value):
        if self.root == None:
            self.root = Node(value)
        else:
            self._insert(value, self.root)
    def _insert(self, value, cur_node):
        if value < cur_node.value:
            if curr_node.left_child == None:
                cur_node.left_child = Node(value)
            else:
                self._insert(value, cur_node.left_child)
        elif value > cur_node.value:
            if cur_node.right_child == None:
                cur_node.right_child = Node(value)
            else:
                self._insert(value, cur.right_child)
        else:
            print("Value already in tree")
```

```
def print_tree(self):
    if self.root != None:
        self._print_tree(self.root)

def _print_tree(self, cur_node):
    if cur_node != None:
        self._print_tree(cur_node.left_child)
        print str(cur_node.value)
        self._print_tree(cur_node.right_child)

def fill_tree(tree, num_elems = 100, max_int = 1000):
    from random import randint
    for _ in range(num_elems):
        cur_elem = randint(0, max_int)
        tree.insert(cur_elem)
    return tree

#Driver Code
tree = binary_search_tree()
tree = fill_tree(tree)
tree.print_tree()
```

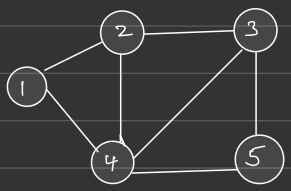
```
def _height(self, cur_node, cur_height):
    if cur_node == None:
        return cur_height
    left_height = self._height(cur_node.left_child, cur_height)
    right_height = self._height(cur_node.right_child, cur_height)
    return max(left_height, right_height)

def height(self):
    if self.root != None:
        return self._height(self.root, 0)
    else:
        return 0
```

```
def search(self,value):
    if self.root != None:
        return self._search(value, self.root)
    else:
        return False

def _search(self, value, cur_node):
    if value == cur_node.value:
        return True
    elif value < cur_node.value and cur_node.left_child != None:
        return self._search(value, cur_node.left_child)
    elif value > cur_node.value and cur_node.right_child != None:
        return self._search(value, cur_node.right_child)
    return False

# driver code

tree = binary_search_tree()

tree.insert(5)
tree.insert(1)
tree.insert(3)
tree.insert(2)
tree.insert(7)
tree.insert(10)
tree.insert(0)
tree.insert(20)
tree.print_tree()
print("tree height", str(tree.height()))
print(tree.search(10))
print(tree.search(20))
```

GRAPHS - 4T

Representation of graph:

1) Adjacency Matrix: (n × n Matrix) ← no. of vertices



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

$\Rightarrow O(n^2)$

a[i][j] = 1, if i & j are adjacent
0, otherwise

2) Adjacency list: (n ← no. of vertices linked lists)

Space complexity: $O(n+2e)$

Minimum Spanning Tree:

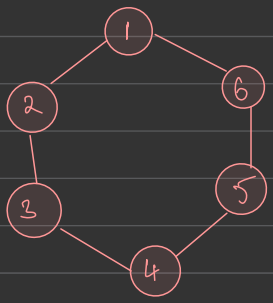
↳ subgraph of undirected graph such that subgraph spans all nodes, connected & acyclic

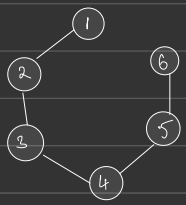
⇒ min total edge weight

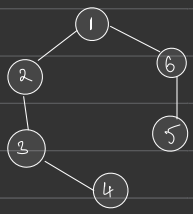
Minimum Cost Spanning Tree:

↳ have all vertices, but edges 1 less than vertices

$G = (V, E)$

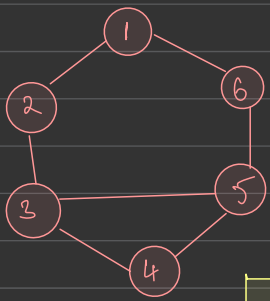






no of edges $|E| = 6$

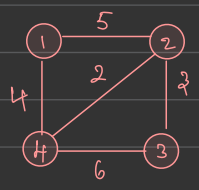
No. of ways to draw spanning tree = ${}^6C_5 = 6$ ways

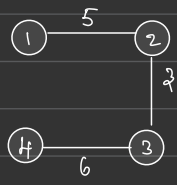


⇒ No of spanning trees = ${}^7C_5 - 2$

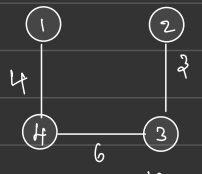
No. of spanning trees = ${}^{|E|}C_{|V|-1}$ − No. of cycles

For weighted graphs:

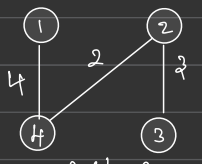




cost = 14

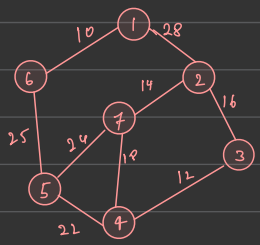


cost = 13

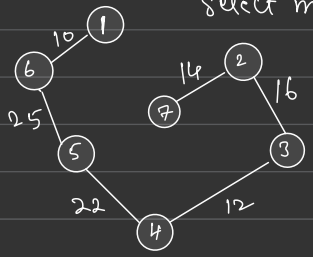


cost = 9

PRIM'S ALGORITHM: can't be applied for not connected graphs

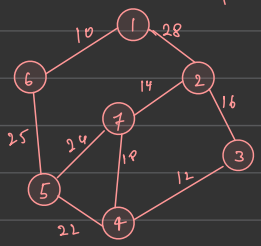


Select the minimum cost edge, and continously select minm weighted adjacent vertices.



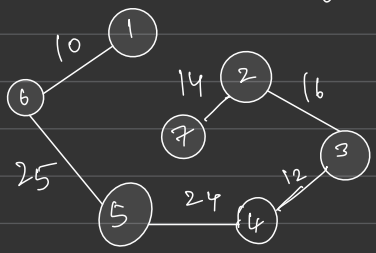
cost = 99 (minm cost)

KRUSKAL'S ALGORITHM:



$O(n^2)$

Select all minm edges ⇒ but don't form cycle



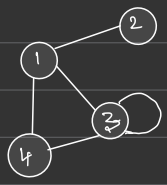
T. SADAKOPA RAMAKRISHNAN
IT-B 3122 22 5002 109

DSA- GRAPH ASSIGNMENT

Q1) $V(G) = \{1,2,3,4\}$

$E(G) = \{(1,2), (1,3), (3,3), (3,4), (4,1)\}$

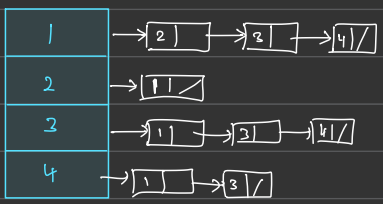
Graph:



Adjacency Matrix:

	1	2	3	4
1	0	1	1	1
2	1	0	0	0
3	1	0	1	1
4	1	0	1	0

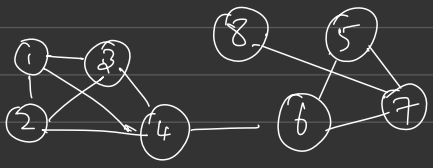
Adjacency list:



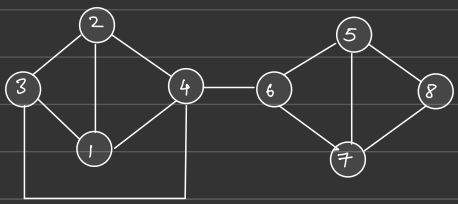
Q2)

Vertex	Adjacent vertices:
1	2, 3, 4
2	1, 3, 4
3	1, 2, 4
4	1, 2, 2, 6
5	6, 7, 8
6	4, 5, 7
7	5, 6, 8
8	5, 7

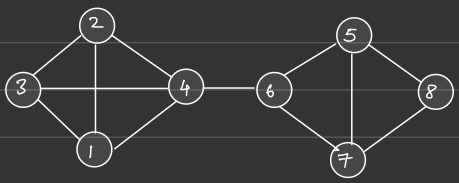
Graph:



proper:

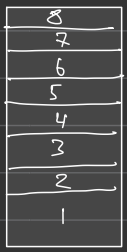


(or)

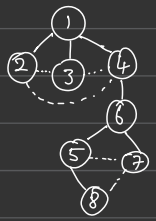


DFS traversal:

Traversal: 1, 2, 3, 4, 6, 5, 7, 8

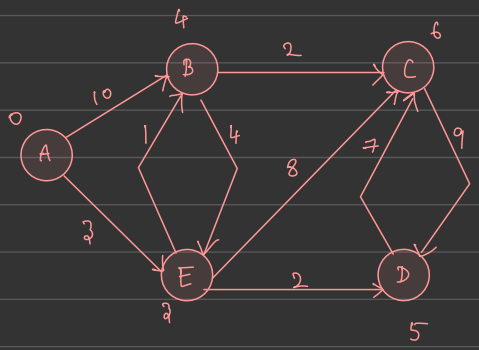


BFS Traversal:



Traversal: 1, 2, 3, 4, 6, 5, 7, 8.

Q3)



$d_u + c_{uv} < d_v$

Solution: Vertex A:

Vertex	known	distance d_v	Path$_v$
A	F	0	—
B	F	∞	
C	F	∞	
D	F	∞	
E	F	∞	

1) $u = A$, $v = B, E$

(i) $u = A, v = B$

$c_{uv} = 10, d_u = 0, d_v = \infty$

$0 + 10 < \infty$

$d_v = 10$

(ii) $u = A, v = E$

$c_{uv} = 3, d_u = 0, d_v = \infty$

$0 + 3 < \infty$

$d_v = 3$

Vertex	known	distance	Path
A	T	0	—
B	F	10	A
E	F	3	A

2) $u = E$, $V = B, D, C$

(i) $u = E$, $v = B$

$c_{uv} = 1$, $d_u = 3$, $d_v = 10$

$c_{uv} + d_u < d_v$

$1 + 3 < 10$

$4 < 10$

$\therefore d_v = 4$

(ii) $u = E$, $V = D$

$c_{uv} = 2$, $d_u = 10$, $d_v = \infty$

$2 + 3 < \infty$

$5 < 8$

$d_v = 5$

(iii) $u = E$, $V = C$

$c_{uv} = 8$, $d_u = 3$, $d_v = \infty$

$11 < \infty$

$d_v = 11$

Vertex	Known	distance	Path
A	T	0	–
B	F	4	E
E	T	3	A
D	F	5	E
C	F	11	E

3) $u = B$, $V = C, E$

(i) $u = B$, $V = C$.

$c_{uv} = 2$, $d_u = 4$, $d_v = 11$

$6 < 11$

$d_v = 6$

(ii) $u = B$, $V = E$

$c_{uv} = 4$, $d_u = 4$, $d_v = 3$

$8 \not< 3$

$d_v = 3$.

Vertex	Known	distance	Path
A	T	0	–
B	T	4	E
E	T	3	A
D	F	5	E
C	F	6	B

4) $u = D$, $v = C$

$c_{uv} = 7$, $d_u = 5$, $d_v = 6$

$12 \not< 6$

$d_v = 6$

Vertex	Known	distance	Path
A	T	0	–
B	T	4	E
E	T	3	A
D	T	5	E
C	F	6	B

5) $u = C$, $v = D$

$c_{uv} = 9$, $d_u = 6$, $d_v = 5$

$11 \not< 5$

$d_v = 5$

Vertex	Known	distance	Path
A	T	0	–
B	T	4	E
E	T	3	A
D	T	5	E
C	T	6	B

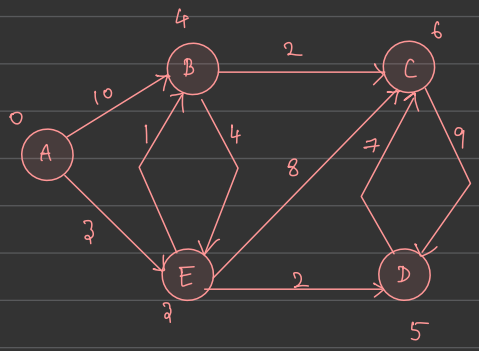
Shortest path to reach vertex A:

(i) $B \rightarrow E \rightarrow A$

(ii) $C \rightarrow B \rightarrow E \rightarrow A$

(iii) $D \rightarrow E \rightarrow A$

(iv) $E \rightarrow A$



Vertex B:

Vertex	known	distance d_v	Path $_v$
A	F	∞	–
B	F	0	
C	F	∞	
D	F	∞	
E	F	∞	

1) $u = B$, $v = E, C$

(i) $u = B$, $v = E$.

$c_{uv} = 4$, $d_u = 0$, $d_v = \infty$

$4 + 0 < \infty$

$d_v = 4$

(ii) $u = B$, $v = C$

$c_{uv} = 2$, $d_u = 0$, $d_v = \infty$

$2 < \infty$

$d_v = 2$.

Vertex	known	distance	Path
B	T	0	–
C	F	2	B
E	F	4	B

2) $u = C,\ v = D$

$c_{uv} = 9,\ d_u = 2,\ d_v = \infty$

$11 < \infty$

$d_v = 11$

Vertex	Known	distance	Path
B	T	0	–
C	T	2	B
E	F	4	B
D	F	11	C

3) $u = E,\ v = D, C$

(i) $u = E,\ v = D$

$c_{uv} = 1,\ d_u = 4,\ d_v = 11$

$5 < 11$

$d_v = 5.$

(ii) $u = E,\ v = C$

$c_{uv} = 8,\ d_u = 4,\ d_v = 2$

$12 \not< 2$

$d_v = 2$

Vertex	Known	distance	Path
B	T	0	–
C	T	2	B
E	T	4	B
D	F	11	C

4) $u = D,\ v = C$

$c_{uv} = 7,\ d_u = 11,\ d_v = 2$

$18 \not< 2$

$d_v = 2$

Vertex	Known	distance	Path
B	T	0	–
C	T	2	B
E	T	4	B
D	T	11	C

5) $u = A$

↳ none

Vertex	Known	distance	Path
B	T	0	–
C	T	2	B
E	T	4	B
D	T	11	C
A	F	∞	–

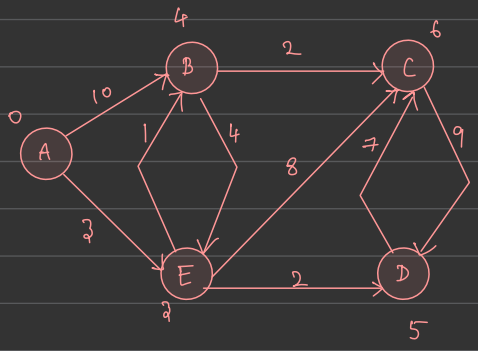
Shortest path to B:

(i) C → B

(ii) E → B

(iii) D → C → B

(iv) A → not possible.



Vertex C:

Vertex	known	distance d_v	Path$_v$
A	F	∞	
B	F	∞	
C	T	0	–
D	F	∞	
E	F	∞	

1) $u = C$, $V = D$

$c_{uv} = 9$, $d_u = 0$, $d_v = \infty$

$9 < \infty$

$d_v = 9$

Vertex	known	distance	Path
C	T	0	–
D	F	11	C

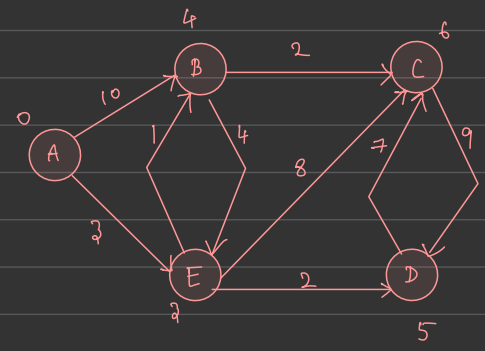
2) $u = D$, $V = C$

$c_{uv} = 7$, $d_u = 11$, $d_v = 0$

$d_v \not< 0$

Shortest path to reach C:

ii) D → C



vertex D:

Vertex	known	distance d_v	Path $_v$
A	F	∞	–
B	F	∞	
C	F	∞	
D	T	0	
E	F	∞	

i) $u = D, V = C$

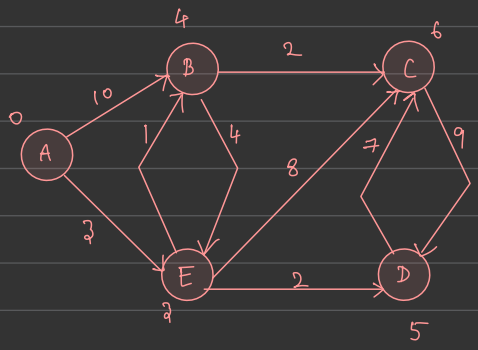
$0 + 7 < \infty$

$d_v = 7$

Vertex	known	distance	Path
D	T	0	–
C	F	7	D

Shortest path to reach D:

ii) C → D



Vertex E :

Vertex	known	distance d_v	Path v
A	F	∞	—
B	F	∞	
C	F	∞	
D	F	∞	
E	T	0	

1) $u = E$, $v = B, C, D$.

(i) $u = E$, $v = B$

$c_{uv} = 1$, $d_u = 0$, $d_v = \infty$

$1 < \infty$

$d_v = 1$

(ii) $u = E$, $v = C$

$c_{uv} = 8$, $d_u = 0$, $d_v < \infty$

$8 < \infty$

$d_v = 8$

(iii) $u = E$, $v = D$

$c_{uv} = 2$, $d_u = 0$, $d_v = \infty$

$2 < \infty$

$d_v = 2$

Vertex	known	distance	Path
E	T	0	—
B	F	1	E
C	F	8	E
D	F	2	E

2) u = B, V = C, E

(i) U = B, V = C

$C_{uv} = 2$, $d_u = 1$, $d_V = 8$

$3 < 8$

$d_V = 3$

(ii) u = B, V = E

$C_{uv} = 4$, $d_u = 1$, $d_V = 0$

$d_V = 0$

Vertex	Known	distance	Path
A	F	∞	
B	T	1	E
C	F	3	B
D	F	2	E
E	T	0	

3) u = C, V = D

$9 + 2 \nless 8$

$d_V = 8$

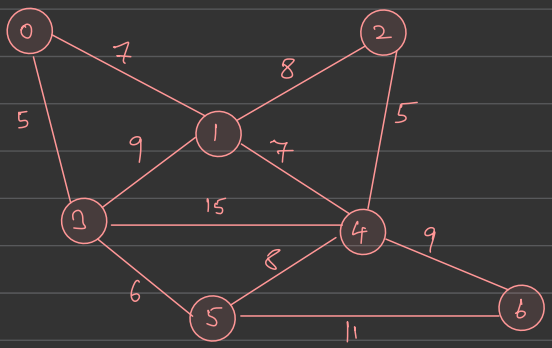
Shortest path to reach E:

(i) E → B

(ii) C → B → E

(iii) D → E

Q4)

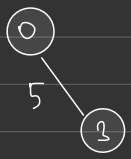


Solution:

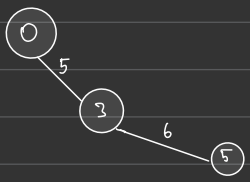
Prim's Algorithm:

Step 1:

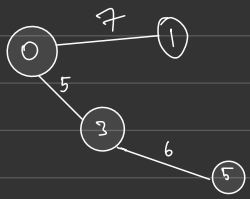
Step 2:



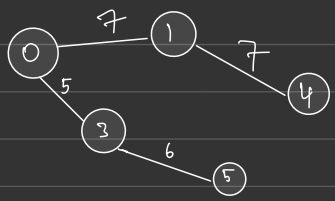
Step 3:



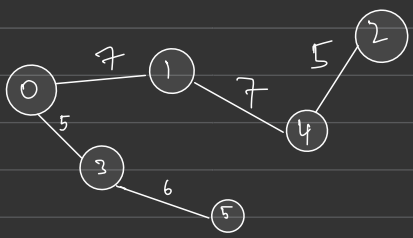
Step 4:



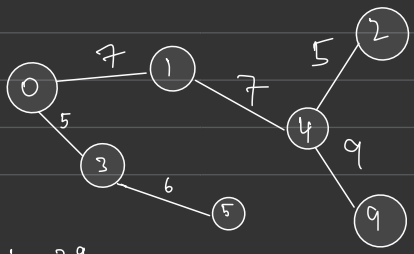
Step 5:



Step 6:



Step 7:

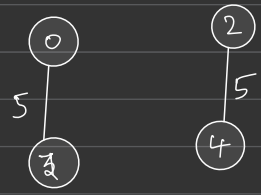


Cost = 39

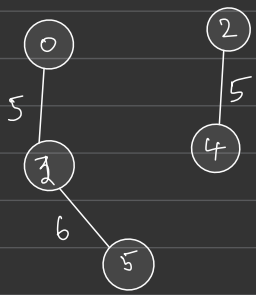
Kruskal's Algorithm:

Step 1:

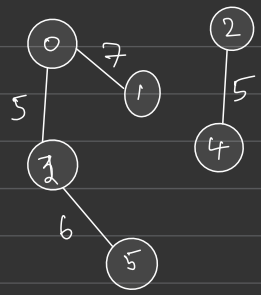
Step 2:



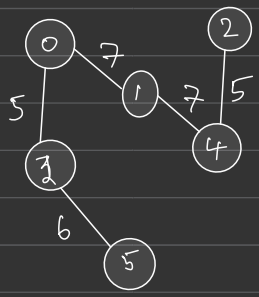
Step 3:



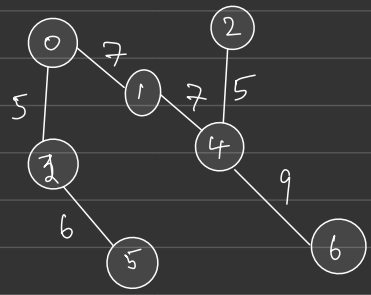
Step 4:



Step 5:

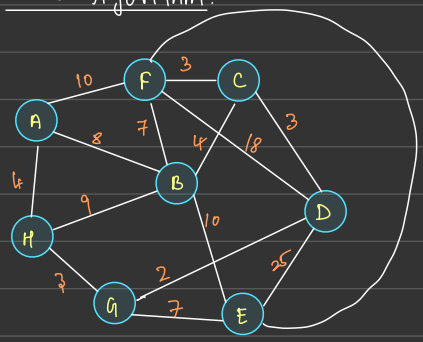


Step 6:



cost = 39

Prim's Algorithm:



	Known	d_v	P_v
A	F	∞	—
B	F	∞	—
C	F	∞	—
D	F	∞	—
E	F	∞	—
F	F	∞	—
G	F	∞	—
H	F	∞	—

① Start with any node, say D and update distances of adjacent, unselected nodes.

	Known	d_v	P_v
A			
B			
C		3	D
D	T	0	—
E		25	D
F		18	D
G		2	D
H			

② Select node with min^m dist, here: DG ⇒ make G to true and update distances of adjacent, unselected nodes of G.

	Known	d_v	P_v
A			
B			
C		3	D
D	T	0	—
E		~~25~~ 7	~~D~~ G
F		18	D
G	T	2	D
H		3	G

③ Select next min^m node, fore c and repeat the process.

	Known	d_v	P_v
A			
B		4	C
C	T	3	D
D	T	0	—
E		~~25~~ 7	~~D~~ G
F		~~18~~ 3	~~D~~ C
G	T	2	D
H		3	G

④ Select F and repeat.

	Known	d_v	P_v
A		10	F
B		4	C
C	T	3	D
D	T	0	—
E		~~25~~ ~~7~~ 2	~~D~~ ~~G~~ F
F	T	~~18~~ 3	~~D~~ C
G	T	2	D
H		3	G

⑤ Select E

	Known	d_v	P_v
A		10	F
B		4	C
C	T	3	D
D	T	0	—
E	T	~~25~~ ~~7~~ 2	~~D~~ ~~G~~ F
F	T	~~18~~ 3	~~D~~ C
G	T	2	D
H		3	G

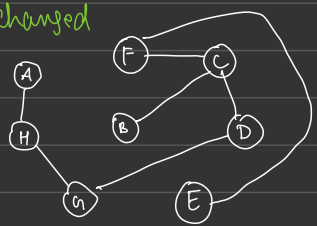
no change

⑥ Select H

	Known	d_v	P_v
A		~~10~~ 4	~~F~~ H
B		4	C
C	T	3	D
D	T	0	—
E	T	~~25~~ ~~7~~ 2	~~D~~ ~~G~~ F
F	T	~~18~~ 3	~~D~~ C
G	T	2	D
H	T	3	G

⑦ Select A & B ⇒ unchanged

	Known	d_v	P_v
A	T	~~10~~ 4	~~F~~ H
B	T	4	C
C	T	3	D
D	T	0	—
E	T	~~25~~ ~~7~~ 2	~~D~~ ~~G~~ F
F	T	~~18~~ 3	~~D~~ C
G	T	2	D
H	T	3	G



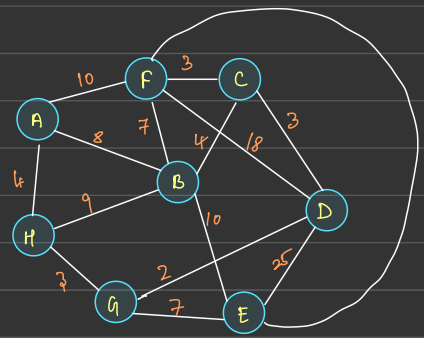
Kruskal's Algorithm:

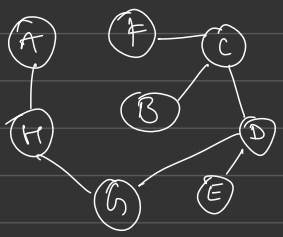
Work with edges, rather than nodes

Two steps:

↳ Sort edges by increasing edge weight

↳ Select the first $|V|-1$ edges that do not generate a cycle.





Edge	dv	
(D, E)	1	✓
(D, G)	2	✓
(E, G)	3	X
(C, D)	3	✓
(G, H)	3	✓
(C, F)	3	✓
(B, C)	4	✓

Edges	dv	
(B, E)	4	X
(B, F)	4	X
(B, H)	4	X
(A, H)	5	✓
(D, F)	6	
(A, B)	8	
(A, F)	10	

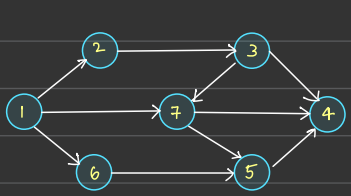
done with all vertices so not req.

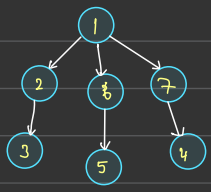
(D, E) → doesn't form a cycle → ✓

(D, G) → ✓

(E, G) → cycle is formed ⇒ X

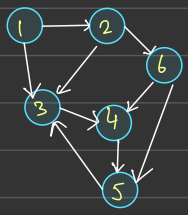
BREADTH FIRST TRAVERSAL:






```
Void BFS(G,S)
{
    //Let Q be Queue
    Q.enqueue(s)
    Mark S visited
    while (Q is not empty)
    {
        V= Q.dequeue()
        for all neighbours
            w of v
        {  if w is not visited
                Q.enqueue(w)
            w mark visited
        }
    }
}
```

DEPTH FIRST SEARCH:




```
void DFS(G,S)
{ Let S be Stack,
    S.push(s)
    Mark S visited
    while (stack not empty)
    {
        V= S.top()
        S.pop()
        for all neighbours w of v
        {
            if w is not visited
            S.push(w)
            mark w as visited
        }
    }
}
```

DIJKSTRA's Algorithm

```
Void Graph:: dijkstra (Vertex s)
{
    for each Vertex v
    {
        V. dist = INFINITY;
        v.known= false;
    }
    S. dist = 0

    for (;;)
    {
        Vertex V= Smallest unknown distance vertex;
        if (v== NOT_A_VERTEX)
            break;
        v.known = true;
        for each Vertex w adjacent to v
            if (!w.known)
                if (v.dist + Cvw < w.dist)
                {
                    // update w
                    decrease(w.dist to v.dist +C vw);
                    w.path = v;
                }
    }
}
```

Recurrance Relation:

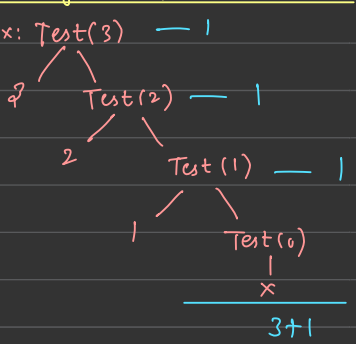
key
[— ⇒ unit of time
— ⇒ unit of time]

1.
```
void Test(int n) — T(n)
{
    if (n>0)
    {
        printf("%d", n); — 1
        Test(n-1); — T(n-1)
    }
}
```

$T(n) = T(n-1) + 1$

Tracing Tree / Recurssive Tree:

Ex: Test(3) — 1



f(n) = (n+1) call O(n)

$$T(n) = \begin{cases} 1 & , n=0 \\ T(n-1)+1 & , n>0 \end{cases}$$

Solving,

$T(n) = T(n-1) + 1$ ∵ $T(n-1) = T(n-2) + 2$

$T(n) = T(n-2) + 2$

$T(n) = T(n-3) + 3$

⋮

$T(n) = T(n-k) + k$

Assume $n-k = 0$

$n = k$

$T(n) = T(n-n) + n$

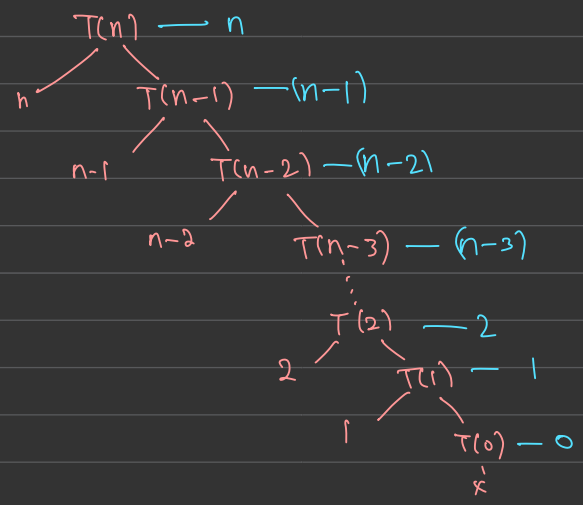
$T(n) = T(0) + n$

$\boxed{T(n) = n+1} \Rightarrow \theta(n)$

2. Decreasing function

```
void Test(int n)          ---- T(n)
{
    if (n>0)              ---- 1
    {
      for (i=0;i<n;i++)   ---- n+1
      {
        printf("%d", n);  ---- n
      }
      Test(n-1);          ---- T(n-1)
    }
}
```

Recursion tree:



$T(n) = T(n-1) + 2n+2$

$2n+2$ is order of n

$T(n) = T(n-1) + n$

$$T(n) = \begin{cases} 1, & n=0 \\ T(n-1)+n, & n>0 \end{cases}$$

$0 + 1 + 2 + \cdots + n-1 + n$

$= \frac{n(n+1)}{2}$.

$T(n) = \frac{n(n+1)}{2}$

$\Theta(n^2)$

$T(n) = T(n-1) + n$ $\quad$ $T(n-1) = T(n-2) + n-1$

$T(n) = T(n-2) + n - 1 + n$ $\quad$ $T(n-2) = T(n-3) + n-2$

$T(n) = T(n-2) + (n-1) + n$

$T(n) = T(n-2) + n-2 + (n-1) + n$

$\vdots$

$T(n) = T(n-k) + (n-k-1) + (n-k-2) + \cdots (n-1) + n$

Assume $n-k = 0$

$n = k$

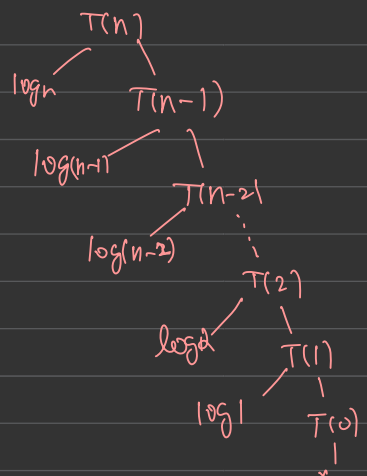
$T(n) = T(0) + (1) + 2 + \cdots (n-1) + n$

$= 1 + \frac{n(n+1)}{2} \Rightarrow \Theta(n^2)$

2.
```
void Test(int n)        ---- T(n)
{
    if (n>0)
    {
        for( i=1; i<n; i=i*2)
        {
            printf("%d", i);    ---- logn
        }
        Test(n-1);      ---- T(n-1)
    }
}
```

$T(n) = T(n-1) + \log n$

$$T(n) = \begin{cases} 1, & n = 0 \\ T(n-1) + \log n, & n > 0 \end{cases}$$



$\log n + \log(n-1) + \cdots + \log 2 + \log 1$

$\log(n \times n-1 \times \cdots \times 2 \times 1)$

$\log(n!)$

$\approx \log(n^n) \Rightarrow n \log n$

$O(n \log n)$

$T(n) = T(n-1) + \log n$

$T(n) = T(n-2) + \log(n-1) + \log n$

$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n$

$\vdots$

$= T(n-k) + \log(n-k-1) + \cdots \log(n-1) + \log n$

Assume $n - k = 0$

$n = k$

$T(n) = T(0) + \log 1 + \log 2 + \cdots \log(n-1) + \log n$

$T(n) > \lceil \log n! \rceil$

$\approx 1 + n \log n$

$O(n \log n)$

Shortcut:

$T(n) = T(n-1) + 1 \longrightarrow O(n)$

$T(n) = T(n-1) + n \longrightarrow O(n^2)$

$T(n) = T(n-1) + \log n \longrightarrow O(n \log n)$

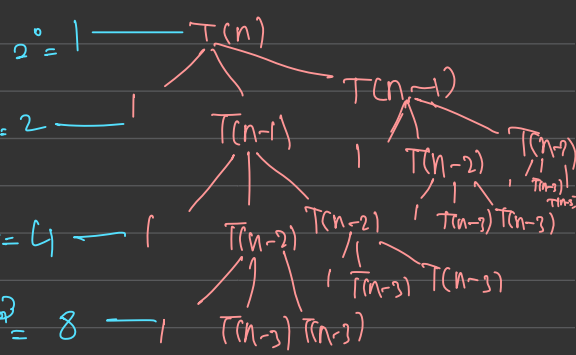
$T(n) = T(n-2) + 1 \longrightarrow O(n)$

$T(n) = T(n-100) + n \longrightarrow O(n^2)$

4. Algorithm Test(int n) — T(n)

```
{
    if (n>0)
    {
        printf("%d", n); — 1
        Test(n-1); — T(n-1)
        Test(n-1); — T(n-1)
    }
}
```

$T(n) = 2T(n-1) + 1$



$1+2+2^2+\cdots 2^k = 2^{k+1} - 1$

$\left(a + ar + ar^2 + \cdots + ar^k = \frac{a(r^{k+1}-1)}{r-1}\right)$

$$T(n) = \begin{cases} 1, & n=0 \\ 2T(n-1)+1, & n>0 \end{cases}$$

$T(n) = 2T(n-1) + 1$

$T(n) = 2[2(T(n-2)+1] + 1$

$= 2^2 T(n-2) + 2 + 1$

$= 2^2[2T(n-3)+1] + 2 + 1$

$= 2^3 T(n-3) + 2^2 + 2 + 1$

$\vdots$

$= 2^k T(n-k) + 2^{k-1} + \cdots 2^2 + 2 + 1$

$T(n) = 2^k T(n-k) + \underbrace{2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 + 1}_{2^k - 1}$

Assume $n - k = 0$

$n = k$

$2^n \times 1 + 2^n - 1$

$2 \times 2^n - 1$

$\Rightarrow \Theta(2^n)$

$\because n - k = 0$

$n = k$

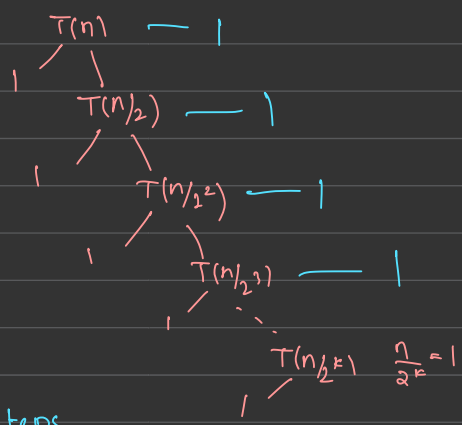
$\Rightarrow 2^{n+1} - 1$

$\Rightarrow \Theta(2^n)$

$T(n) = 2T(n-1) + 1$	$O(2^n)$
$T(n) = 3T(n-1) + 1$	$O(3^n)$
$T(n) = 2T(n-1) + n$	$O(n \cdot 2^n)$

5. **Dividing function:**

```
Algorithm Test(int n) ——— T(n)
{
    if (n>1)
    {
        printf("%d", n); — 1
        Test(n/2); — T(n/2)
    }
}
```



$T(n) = T(n/2) + 1$

$\frac{n}{2^k} = 1$

$k = \log_2 n$

K steps → $\theta(\log n)$

$$T(n) = \begin{cases} 1 & , n = 1 \\ T(n/2) + 1 & , n > 1 \end{cases}$$

$T(n) = T(n/2) + 1$

$= T(n/2^2) + 1 + 1$

$= T(n/2^3) + 2$

$\vdots$

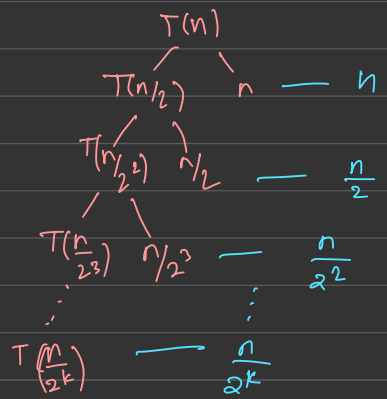
$= T(n/2^k) + k$

$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

$\Rightarrow T(n) = T(1) + \log_2 n$

$\boxed{\theta(\log n)}$

6) $T(n) = \begin{cases} 1 & , n=1 \\ T(n/2)+n & , n>1 \end{cases}$



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

$= T(n/2^2) + n/2 + n$

$= T\left(n/2^3\right) + \frac{n}{2^2} + \frac{n}{2} + n$

$= T\left(\frac{n}{2^k}\right) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \cdots \frac{n}{2} + n$

$T(n) = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^k}$

$\frac{n}{2^k} = 1$

$k = \log_2 n$

$T(n) = n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^k}\right]$

$= n \sum_{i=0}^{k} \frac{1}{2^i} = n \times 1 \quad \Theta(n)$

$= T(1) + n\left(\frac{1}{2^{k-1}} + \frac{1}{2^{k-2}} + \cdots + \frac{1}{2} + 1\right)$

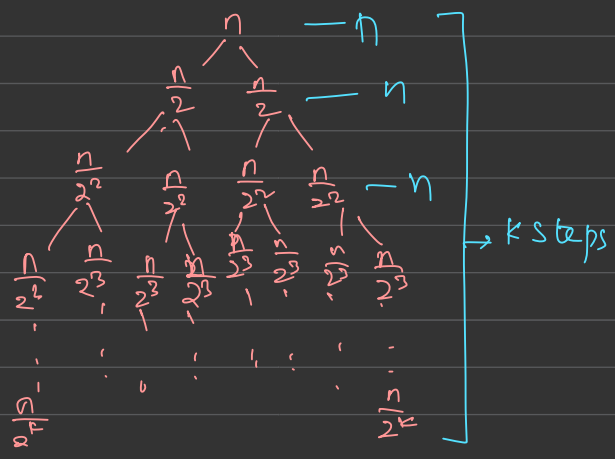
$= T(1) + n[1+1]$

$= 1 + 2n$

$\Theta(n)$

7)
```
void Test (int n)        — T(n)
{
    if (n>1)             — 1
    {
        for (i=0; i<n; i++)
        {
            stmt;
        }                → n
        Test(n/2);       — T(n/2)
        Test(n/2);       — T(n/2)
    }
}
```

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



Assume $\frac{n}{2^k} = 1$

$T(n) = nk$

$= n \log n$

$\Theta(n \log n)$

$$T(n) = \begin{cases} 1 & , n=1 \\ 2T(n/2)+n & , n>1 \end{cases}$$

$T(n) = 2T(n/2) + n$ $\quad T(n/2) = 2T(n/2^2) + n/2$

$T(n) = 2\left[2T(n/2^2) + \frac{n}{2}\right] + n$ $\quad T(n/2^2) = 2T(n/2^3) + n/2^2$

$T(n) = 2^2 T(n/2^2) + n + n$

$T(n) = 2^2\left[2T(n/2^3) + \frac{n}{2^2}\right] + 2n$

$T(n) = 2^3 T(n/2^3) + 3n$

$\vdots$

$T(n) = 2^k T(n/2^k) + kn$

$\frac{n}{2^k} = 1 \Rightarrow k = \log n$

$T(n) = 2 \times 1 + n \log n \Rightarrow \boxed{O(n \log n)}$

HEAPS

↳ A certain kind of complete binary tree.

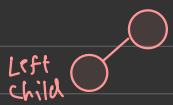
↳ All levels are completely filled except the last level that may or may not be completely filled

↳ Elements are filled from L to R

How Insertion works?

Step 1:

Step 2:



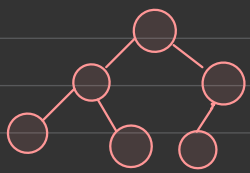
Step 3:

Step 4:

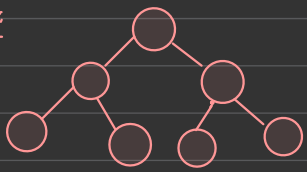
The next nodes always fill the next level from left to right

Step 5:

Step 6:

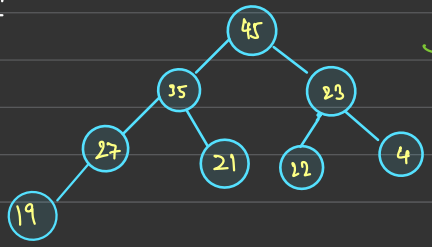


Step 7:



- Each node in a heap contains a key that can be compared to other node's keys.

Ex:

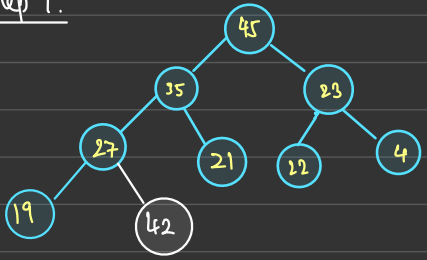


⇒ Each nodes' key ≥ keys of its children

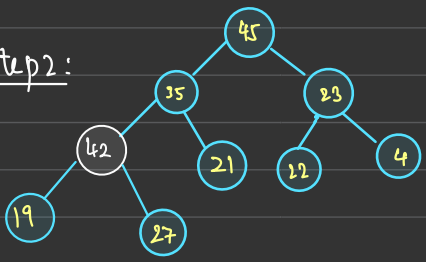
Adding new node to Heap: Add (42)

↳ O(logn)

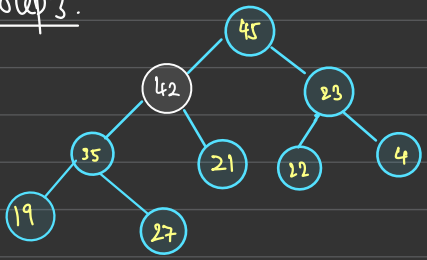
Step 1:



Step 2:



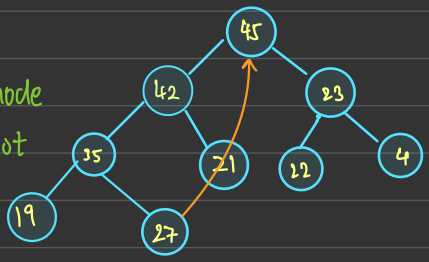
Step 3:

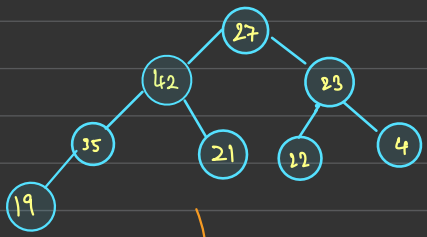


⇒ "Reheapification upward".

Removing the top of a Heap: $\Theta(\log n)$

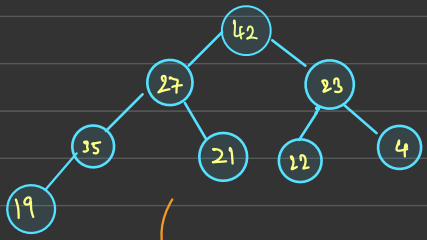
Step 1:
Move last node onto the root





Step 2:

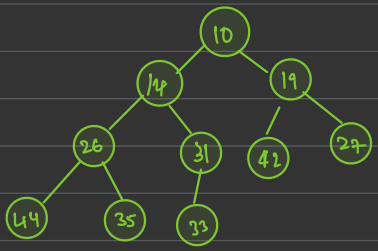
Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location



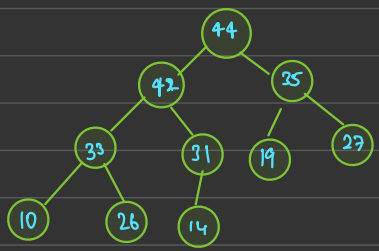
"Reheapification Downward".

Input: 35 33 42 10 14 19 27 44 26 31

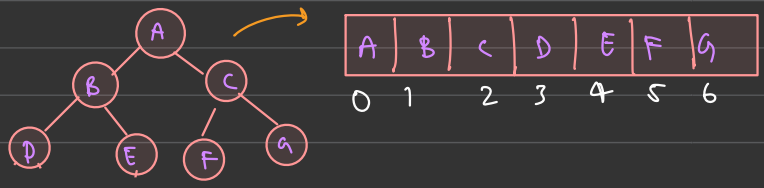
Min Heap: Where the value of the root node is less than or equal to either of its children.



Max Heap: Where the value of the root node is greater than or equal to their children.



Representation of Binary Tree:



Up heap $O(n \log n)$

Given: | 1 | 3 | 2 | 5 | 4 |

Step 1: | 1 | | 3 | 2 | 5 | 4 | Add 1st element

Step 2: | 1 | 3 | | 2 | 5 | 4 | Adding next element

| 3 | 1 | | 2 | 5 | 4 | Swapping with parent as 3 > 1

sorted

Step 3: | 3 | 1 | 2 | | 5 | 4 | Next element is added

Already sorted

Step 4: | 3 | 1 | 2 | 5 | | 4 | Next element is added

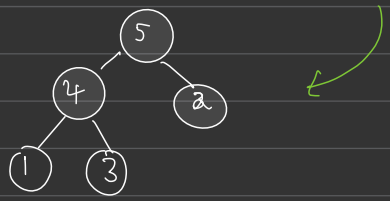
| 3 | 5 | 2 | 1 | | 4 | 5 is swapped with parent 1

| 5 | 3 | 2 | 1 | | 4 | 5 is again swapped with parent 3

sorted

Step 5: | 5 | 3 | 2 | 1 | 4 | 4 is added

| 5 | 4 | 2 | 1 | 3 | sorted | 4 is swapped with parent 3

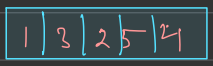


Down heap : O(nlogn)

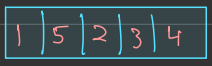
Given: | 1 | 3 | 2 | 5 | 4 |

Start from i = len(arr)//2 - 1

Step1:

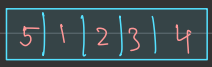


Compare i=1 with its children 5 & 4



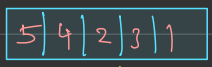
swap 5 & 3.

Step2:



Move to i=0,
swap 5 & 1

Step3:



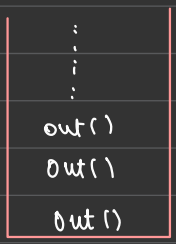
sorted.

Now compare 1 with 3 & 4
Swap 4 & 1.

Recursion

When a function calls itself until a specified condition is met

```
def Out():
    print(1)
    Out()

def main():
    Out()
```

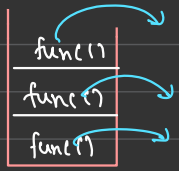


stack overflow

Infinite recursion

Base Condition:

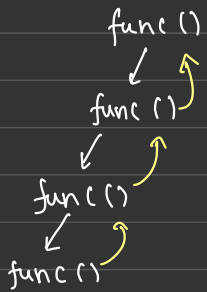
```
cnt = 0
def func():
    if cnt == 3:
        return
    print(cnt)
    cnt += 1
    func()
```



output

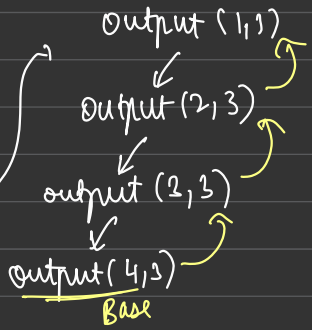
0
1
2

Recursion Tree:



Q) Print name `N` times

```
def output(i,n):
    if i>n:
        return
    print("sada")
    output(i+1,n)
output(1,3)
```



TC: O(n)

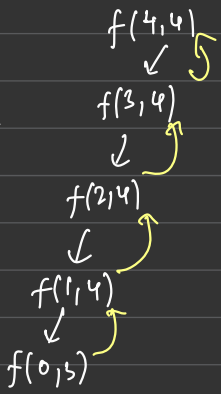
SC: O(n)

Print linearly from 1 to N.

```
def f(i,n):
    if i>n:
        return
    print(i)
    f(i+1,n)
```

Print N to 1.

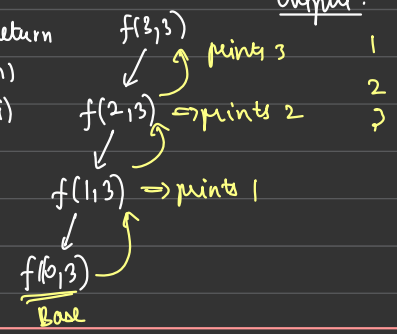
```
def f(i,n):
    if i<1:
        return
    print(i)
    f(i-1,n)
n=4
f(n,n)
```



Print linearly from 1 to N (By Backtracking)

```
def f(i,n):
    if i< 1:
        return
    f(i-1,n)
    print(i)
```

n=3
f(n,n)

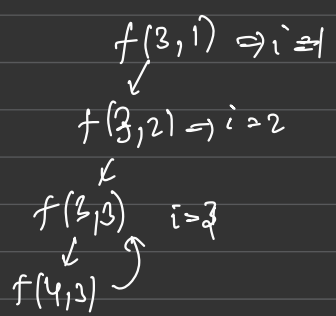


Output:
1
2
3

Print from N to 1 (Backtracking)

```
def f(i,n):
    if i <1:
        return
    f(n,i+1)
    print(i)
```

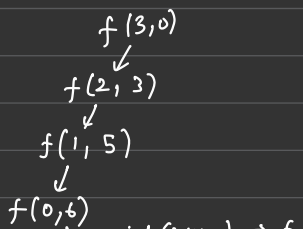
n=3
f(n,1)



Sum of first N numbers:

Parameterised way.

```
def f(i, sum):
    if i < 1:
        print(sum)
        return
    f(i-1, sum+i)
n = 3
f(n, 0)
```



functional:

```
def f(n):
    if n == 0:
        return 0
    else:
        return n + f(n-1)
```

Heapq Module

→ Provides implementation of min heap.
→ Easy implementation of priority queues

```
import heapq
```

Functions:

① **heappush(heap_name, item):**

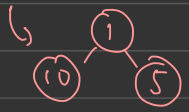
↳ push item onto heap, maintaining the heap property.

Ex: `heap = []`

`heapq.heappush(heap, 10)` → output : [10]

`heapq.heappush(heap, 1)` → output : [1, 10] → min heap property

`heapq.heappush(heap, 5)` → output : [1, 10, 5]



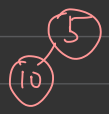
② **heappop(heapname):**

↳ returns smallest value & deletes it from heap

Ex: `heapq.heappush(heap, 5)` → output : [1, 10, 5]

Basically root node is deleted.

`heapq.heappop(heap)` → output : 1



③ heapify(heap_name): → Convert a list to heap

Ex: list1 = [1, 3, 5, 2, 4, 6]

heapq.heapify(list1) ⇒ [1, 2, 5, 3, 4, 6]

④ heappushpop(heap_name, item):

↳ perform heappush & heappop at the same time

Ex: list1 = [1, 3, 5, 2, 4, 6]

heapq.heappushpop(list1, 89) ⇒ output: 1

↳ list1: [2, 3, 5, 89, 4, 6]

⑤ heapreplace(heap_name, item):

↳ perform pop & then push

Ex: list1 = [2, 3, 5, 89, 4, 6]

heapq.heapreplace(list1, 100) ⇒ output: 2

↳ list1 = [3, 4, 5, 89, 100, 6]

⑥ nsmallest(n, iterable, key = None):

↳ return 'n' smallest numbers

if n = 2 ⇒ 2 smallest numbers

Ex: heap = [1, 20, 5, 4, 3, 6, 2]

heapq.nsmallest(2, heap) ⇒ output: [1, 2]

⑦ nlargest(n, iterable, key = None)

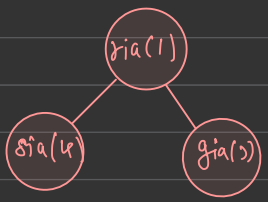
Priority Queue using heapq:

Priority ↓

list1 = [(1, "ria"), (4, "sia"), (3, "gia")]

element ↑ ⇒ smallest priority, popped out first

heapq.heapify(list1) ⇒ list1 = [(1, 'ria'), (4, 'sia'), (3, 'gia')]




```
for i in range(len(list1)):
    print(heapq.heappop(list1))
```

⇒ output: (1, 'ria')
(3, 'gia')
(4, 'sia')