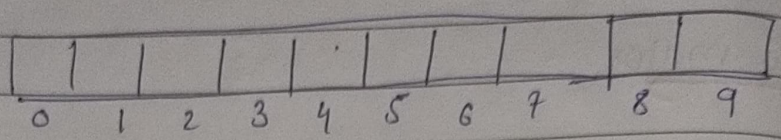
Array ADT

ADT (Abstract Datatype) → Representation of Data
→ operation we can perform on Data

*



int A[10]; or int *A
A = new int [size]

* Operations

1) Display()
2) Add(x) / Append(x)
3) Insert(index, x)
4) Delete(index)
5) Search(x)
6) Get(index)
7) set(index)
8) Max() / Min()
9) Reverse()
10) Shift() / Rotate()

Searching → linear Search
↳ Binary Search.

§ Linear Search.

Succesful Search → Best O(1)
↘ worst O(n)

unsuccessful search → worst O(n)

```
for (i=0; i<length ; i++)
{   if (A[i] = value
        return value i;
}
return -1
```

→ Improving linear search

- Transposition → Reduce the position of the element by 1
- Move to Front/Head → Replace element with index [0]

Binary Search

→ Here array is sorted

→ if value is equal to ~~middle~~ value of array middle position is returned.

→ if middle position value is less than key value then put l = middle + 1

→ else h = m - 1

initial →

l = 0

h = arr. length

m = (l + h)/2

→ Time complexity.

n/2 = n/2 for I = 1

n/2/2 = (n/2)/2 = n/2² for I = 2

n/2/2/2 = ((n/2)/2)/2 = n/2³ for I = 3

n/2^K = for I = K

So Here the loop is ran K times i.e time complexity

n = n/2^K for I = K

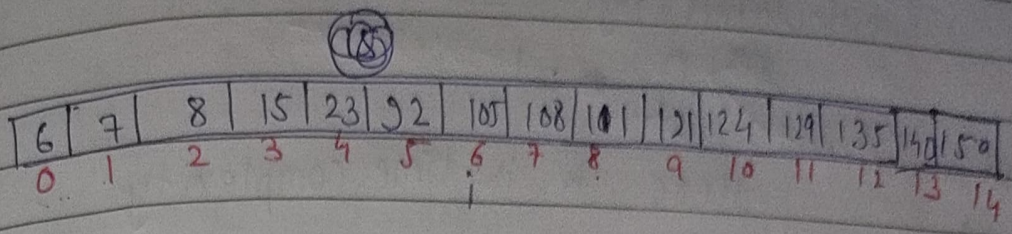
n = 1, taking log on b.s

log(1) = log(n) - K log 2

K ≃ log(n) -- not considering log 2

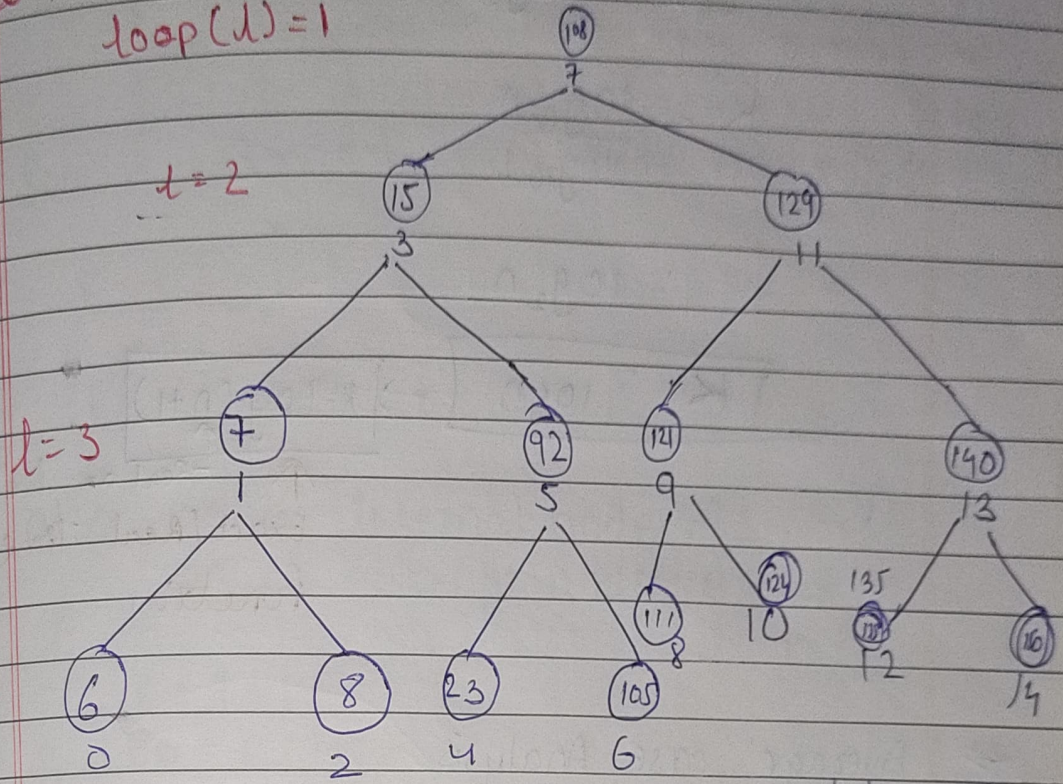
∴ Time

Time complexity :- (worst case → log(n))



→ ∞ middle on first loop

loop(1) = 1



no of element = n

So for t=1 → ~~loop run 1 time~~ n/2

for t=2 → (n/2)/2 = n/2²

for t=3 → n/2³

⋮

for t=k → n/2ᵏ

∴ for k times time complexity is k here

N/2ᵏ

log N − k log 2 = 0 ∴ k = log N

∴ Time complexity is log n

Best case time → 1
worst case time → log(n)
Average case time → log(n)

∴ for n element time complexity is calculated by

$\frac{n}{2^k} = 1$ → 1 element

Actual no of element +1

Here k is time complexity

$\log_{10} n = \log_{10} 2^k$

$k = \frac{\log_{10} n}{\log_{10} 2}$

$= \log_2 n$

$k = \log n$ → $k = \log(n+1)$

Real no of element

Exact/Real Time function

⇒ Average case Analysis from behind example.

$1 + 1 \times 2 + 2 \times 4 + 3 \times 8$

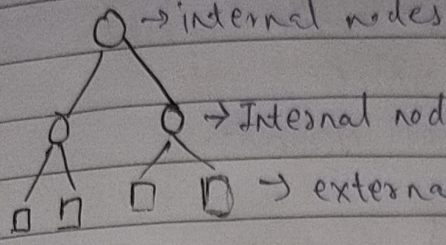
$1 + 2^1 + 2 \times 2^2 + 3 \times 2^3$

$\sum_{i=1}^{\log(n)} \frac{i \times 2^i}{n} = \frac{\log n \times 2^{\log n}}{n}$

$= \frac{\log n \times n}{n}$

$= \log n$

E = sum of path of external nodes.
I = sum of path of internal nodes.
n = number of nodes. i.e height of tree



Sum of path of internal nodes = 1X0 + 2X1 = 2
Sum of path of External node = 4X2 = 8

$E = I + 2n$
$= 2 + 2(3)$
$E = 8 = 8$

also no of internal nodes = i
no of external nodes = e

$e = i + 1$
$E = n \log n$

Average Succesful search $A_s(n) = 1 + \frac{I}{n} = \log n$

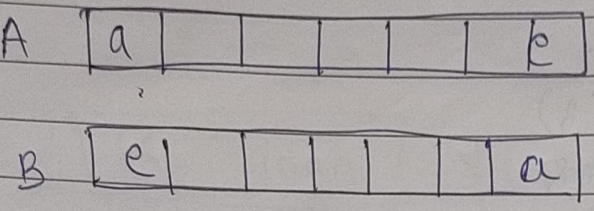
Average unsuccesfull search $A_u(n) = \frac{E}{n+1} = \frac{n \log n}{n+1}$

$= \log n$

* Get() → Reading a value
  Set() → writing a value.
  Max() → Reads maximum value.
  Min() → Reads minimum value.
  Sum() → Sum of all value.
  Average() → Average of all elements.

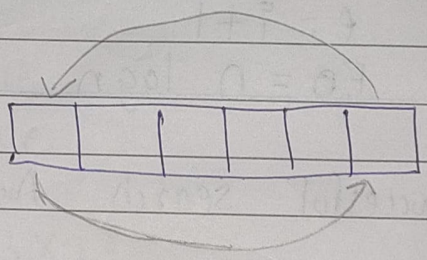
* Reverse and shift of an array.

1st Method
→ Using an auxiliary array B

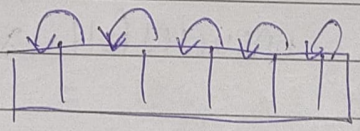


Firstly last element of array A will be copied to B (reversely) i.e as first element then B elements are copied to A

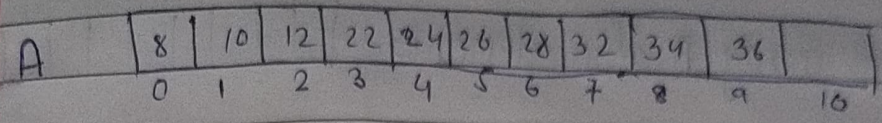
2nd Method
→ Inter changing →



3rd → left shift



* checking if the list is
* Inseating element in a sorted list.



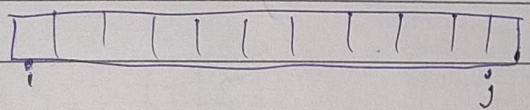
insert 25
∴ let x = 25
i = length of A - 1 i.e arr → length

→
```
while (A[i] > x)
{   A[i+1] = A[i];
    i--;}
A[i+1] = x;
length++;
```

* check if list is sorted.

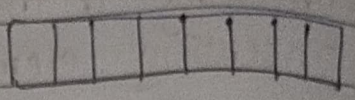
```
if (A[i] > A[i+1])
    return false;
```

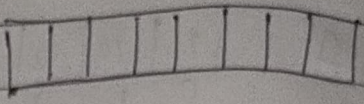
* -ve on left side




```
while (i<j)
{
    while (A[i]<0) {i++}
    while (A[j]>0) {j--}
      if (i<j)
        {swap (A[i], A[j])}
}
```

* Merging:

Merge array A of m elements 

Merge array B of n elements 

Here while

```
i=0;
j=0;
k=0;

while (i<m && j<n)
{
    if( A[i]<B[j]
        c[k++]=A[i++];
    else
        c[k++]=B[j++];
}
for(; i<m; i++)
    c[k++] = A[i++];

for(; j<n; j++)
    c[k++] = B[j++];
```

Date
Page

* Set operations

* Union

for unsorted

A | 3 | 5 | 10 | 4 | 6 | m

B | 12 | 4 | 7 | 2 | 5 | n

C | | | | | | | | | | | |

First copy m elements to C then compare n elements with m elements

$m + n \times m$

$m + m \times n$

$n + m \times n$

$n \approx n^2$

∴ order $\theta(n^2)$

for sorted.

Same as merge

$\theta = (m+n)$

* Intersection.

A | 3 | 5 | 10 | 4 | 6 | m

B | 12 | 4 | 7 | 2 | 5 | n

C | | | | | | | |

∴ Time complexity $O(m \times n)$

Same as merge

$\theta = (m+n)$

* Difference.

Unsorted

A	3	5	10	4	6	m

B	12	4	7	2	5	n

C	8	10	6						
	0	1	2	3	4	5	6	7	8

$\Theta = m \times n$

Sorted

A	3	4	5	6	10	m

B	2	4	5	7	12	n

C	3	6	10		

* find missing element

→ Q.) find missing element in sum of first n natural number.

firstly find sum for i=1; i<n-1; i++)
sum = sum + A[i].

then $S = \frac{n \times n+1}{2}$

Now missing element is S - sum.

→ If array doesn't start from 1

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

diff = 6 - 0

for(i=0; i<6; i++)
  if (A[i] - i != difference)
    print (i + diff is missing)
    break

Q) Find Multiple missing.

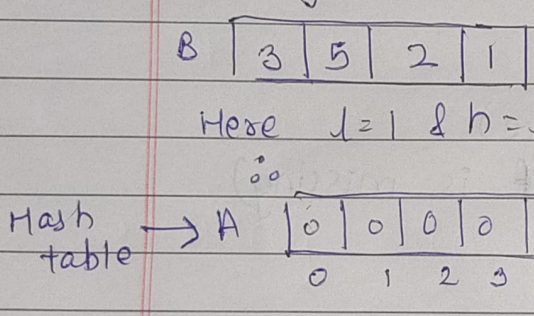
```
diff = A[0] - 0

for (i=0 ; i< A.length ; i++)
    if (A[i] - i != diff)
        { print (i + diff) is missing ;
          diff += 1 ; }
```

Or

```
for (i=0 ; i < A.length ; i++)
    if (A[i] - i != diff)
      { while (diff < A[i] - i)
          { print (i + diff)
            diff ++ ;
          }
      }
```

* Using Hash table i.e Hashing




```
for (i=0 ; i<n ; i++)
    A[B[i]]++ ;
for (i=0 ; i<=h ; i++)
{
    if (H[i]==0)
        printf (i)
}
```

Q) finding Duplicates.

→ for sorted.

| 3 | 2 | 2 | 5 | 5 | 5 |

n = 10

```
last duplicate = 0
for (i=0; i<n-1; i++)   (Time complexity O(n))
{
    if (A[i] == A[i+1] && A[i] != lastDuplicate)
    {
        printf(A[i]);
        lastDublicate = A[i];
    }
}
```

→ Using two variable (Time complexity O(n))

when A[i] == A[i+1]

```
j = i+1
while (A[i] = A[j]) j++;
print A[j] appeared j-i times
i = j-1
```

→ using Hashing.

B

| 0 | 3 | 4 | 5 |

n = 5

A

| 0 | 0 | 0 | 0 | 0 | 0 |

0 1 2 3 4 5

```
for (i=0; i<n; i++)
    A[B[i]]++;

for (i=1; i<m; i++)
    if (A[i] > 1)
        Print i Appears A[i]-1 times
```

* Finding Duplicates for unsorted Array.

```
for (i=0; i<n-1 ; i++) → first loop for i
    for (j=i+1; j<n; j++) → for second loop for j
    {
        if (A[i]==A[j])
            count++;
    }
```

8	7	8	6	2	6	6

Time Complexity $O(n^2)$

→ first place i at index 0
→ then search for 0 index element by matching with each element of array using j
→ if element is same increase count
→ mark duplicate as -1 & increment i value

→ Using Hash table

Time complexity O(n)

---

* Finding a pair of Element with Sum K

→ Using two variable

→ Using Hash table

→ in Sorted array

- Compare begginning array element with array end elements.
- If Sum of element = K then print
- If sum of element < k increment the variable at beg
- Else decrement the variable at end

* Finding Max and Min in single scan.

- Assign first element as Max & min
- Now for every element
  if its less than min set it as min
  else if its more than max set is as max

- Here time complexity is O(n)
- and to be specific in best case its time is n-1
- worst case time is 2(n-1)