Data Structure

GATEWAY CLASSES

Unit-III

Searching

& SORTING

ONE SHOT

KAPIL SIR

DATA STRUCTURE
AKTU
UNIT-3 Syllabus

Searching: Concept of Searching, Sequential search, Index Sequential Search, Binary Search. Concept of Hashing & Collision resolution Techniques used in Hashing. **Sorting:** Insertion Sort, Selection, Bubble Sort, Quick Sort, Merge Sort, Heap Sort and Radix Sort.

1. Searching — Linear | Binary | Indexsequential search
2. Sorting
3. Hashing

What is Searching?

- The process of finding the location of a specific data item or record with a given key value or finding the locations of all records, which satisfy one or more conditions in a list, is called "Searching". If the item exists in the given list then search is said to be successful otherwise if the element if not found in the given list then search is said to be unsuccessful.
- In other words, searching is the process to find the location of an item from the entire collection of items.

Types of Searching

1. External searching
2. Internal searching

- *External searching* means searching the records using keys where there are many records, which reside in files stored on disks.
- *Internal searching* means searching the records using keys where there are less number of records residing entirely within the computer's main memory.

- On the basis of process to find the location of an item from entire collection of items, following are the two important searching techniques:

1. Linear or Sequential Searching
2. Binary Searching

(3) Index sequential search

- In data structures when we use the word searching, we actually refer to internal searching.

LINEAR OR SEQUENTIAL SEARCHING-

- Suppose DATA is a linear array with n elements. Given no other information about DATA The most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one-by-one.
- That is, first we test whether DATA [1] is equal to ITEM, and then we test whether DATA [2] == ITEM, and so on. This method, which traverses DATA sequentially to locate ITEM, is called linear search or sequential search.
- In linear search, each element of an array is read one-by-one sequentially and it is compared with the desired element. A search will be unsuccessful if all the elements are read and the desired element is not found.

Algorithm: Linear Search(A,N,LOC,ITEM)

This algorithm is used to find the location LOC of an ITEM from array A which is of N elements.

Step 1:- Start

Step 2:- For i=0 to N

Step 3:- Repeat steps 4 and 5 if ITEM is equal to A[i]

Step 4:- Set LOC := i.

Step 5:- Exit from step 3 loop.

[End of step 2 loop]

Step 6:- if i is equal to N

Step 7:- Write or Print "ITEM is not found in the list" or "unsuccessful search".

Step 8:- otherwise

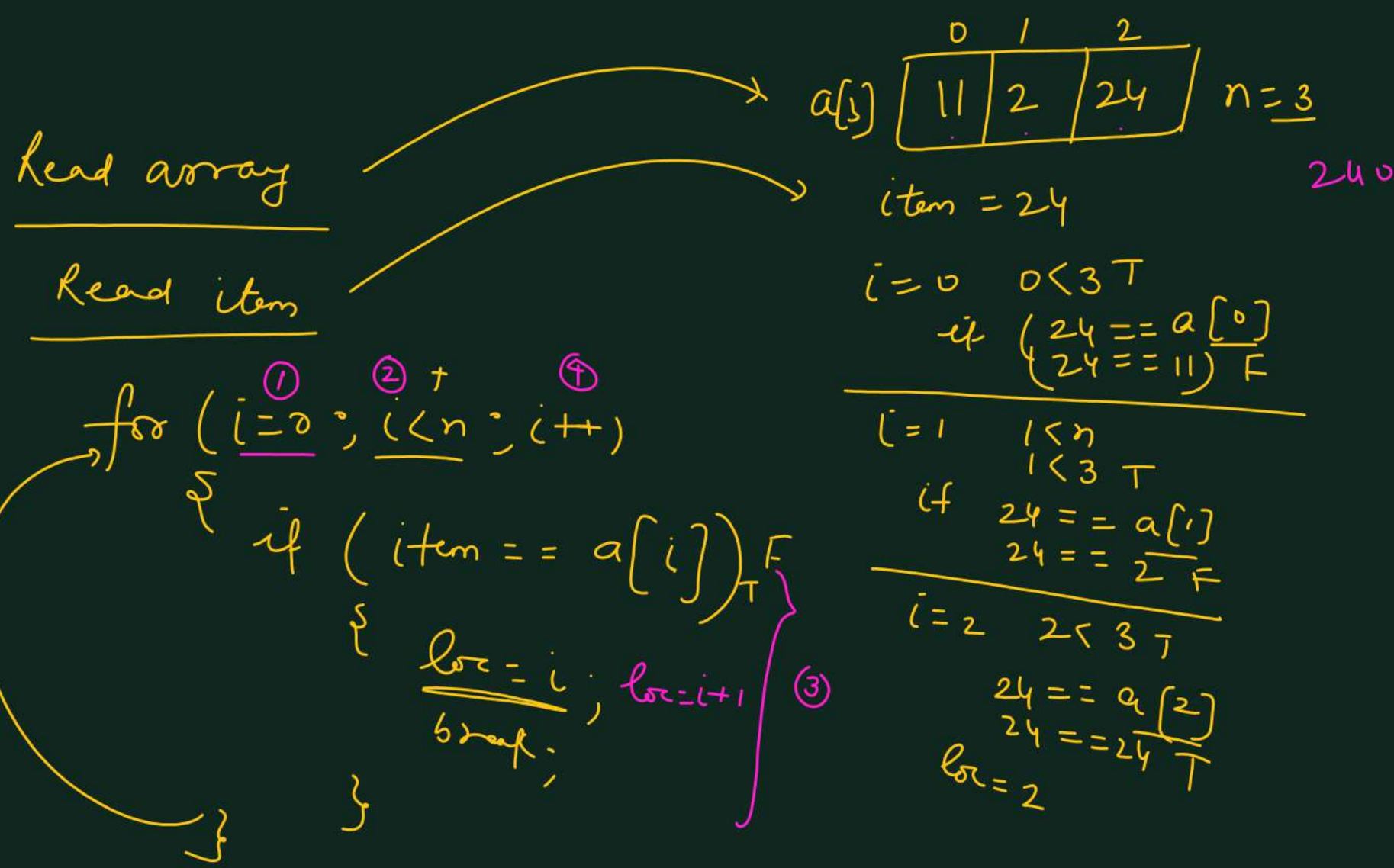
Step 9: -Write or print "ITEM is at LOC location"

[End of step 6 loop or End of if condition at step 6]

Step 10:- Stop

Read array

Read item



C Implementation of Linear Search:-

```
#include<stdio.h>
void main()
{
int a[20];
int n,i,item,loc;
clrscr();
printf("\n How many numbers you want to input for the array:");
scanf("%d",&n);
if(n≥20)
{
printf("\n Invalid input, it should be less then 20");
getch();
exit();
}
for(i=0;i<n;i++)
{
printf("\n enter the number %d:",i+1);
scanf("%d",&a[i]);
}
printf("\n enter the item for which you want to search the location:");
scanf("%d",&item);
for(i=0;i<n;i++)
{
if(a[i]==item)
{
loc=i+1;
break;
}
}
if(i==n)
printf("\n item %d is not in the list",item);
else
printf("\n Item %d is %d element",item,loc);
getch();
}
```

Example

```
for (i=0; i<n; i++)
{
    if (a[i] == item)
    {
        loc = i;
        break;
    }
}
```

	0	1	2
a[3]	11	2	9

n = 3

item = 99

i = 0 | 0 < 3 T | a[0] == 99 | 11 == 99 F

i = 1 | 1 < 3 T | a[1] == 99 | 2 == 99 F

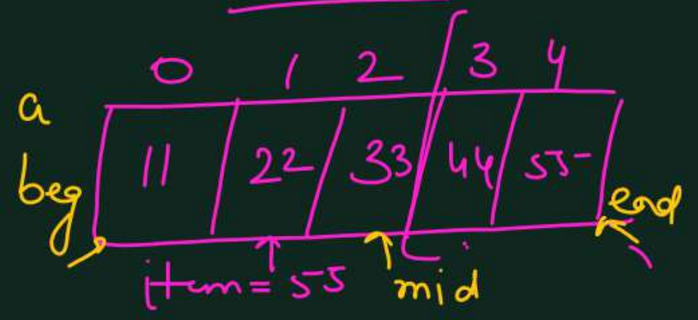
i = 2 | (2 < 3) T | if (a[2] == 99) | 9 == 99 F

i = 3 | (3 < 3) F

BINARY SEARCH

- If we place our items in an array and sort them in either ascending or descending order on the key first, then we can obtain much better performance with an algorithm called binary search.
- Binary search is an extremely efficient algorithm when it is compared to linear search.
- In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater then the item sought must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

Requirement / limitation } List / Array — Sorted



beg = 0, end = n-1

$$mid = \frac{beg + end}{2} = \frac{0+4}{2} = 2$$

```
if ( item == a[mid] );
    loc = mid;
    if ( item < a[mid] )
        end = mid-1;
```

```
if ( item > a[mid] )
    beg = mid+1
```

Concept of Binary Search Algorithm:-

1. Find the middle element of the array *i.e.*, $\frac{n}{2}$ is the middle element where n is number of elements.

2. Compare the middle element with the item to be searched, then there are following three cases:-

(a) If it is a desired element, then search is successful.

(b) If it is less than desired data, then search only the first half of the array, *i.e.*, the elements which come to the left side of the middle element.

(c) If it is greater than the desired data, then search only the second half of the array, *i.e.*, the elements which come to the right side of the middle element. Repeat the same step until an element is found or exhaust the search area.

Algorithm : Binary Search (A, n, beg, end)

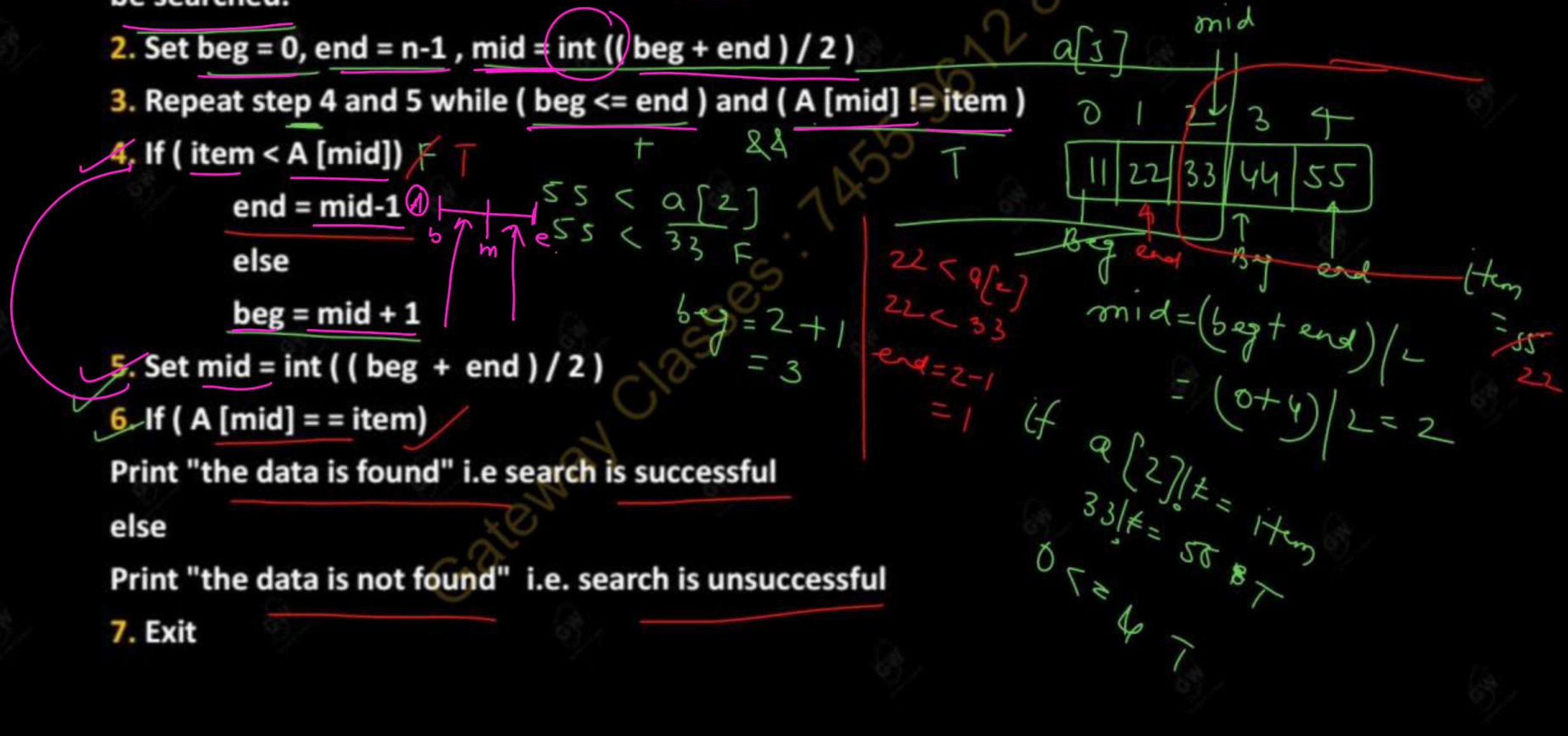
1. Read an array A of n elements in sorted order and read an item for which the location is to be searched.
2. Set beg = 0, end = n-1 , mid = int ((beg + end) / 2)
3. Repeat step 4 and 5 while (beg <= end) and (A [mid] != item)
4. If (item < A [mid])
   end = mid-1
   else
   beg = mid + 1
5. Set mid = int ((beg + end) / 2)
6. If (A [mid] = = item)

Print "the data is found" i.e search is successful

else

Print "the data is not found" i.e. search is unsuccessful

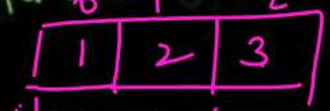
7. Exit



C implementation of Binary Search (non recursive / iterative implem)

```
void main()
{
int a[20],n,i,item,loc,beg,end,mid;
clrscr();
printf("\n How many numbers you want to input:");
scanf("%d",&n);
if(n>20)
{
printf("\n Invalid input, it should be less then 20");
getch();
exit();
}
for(i=0;i<n;i++)
{
printf("\n enter numbers in sorted order, enter the
number %d:",i+1);
scanf("%d",&a[i]);
}
printf("\n enter the item for which you want to search the
location:");
scanf("%d",&item);
beg=0;
mid=(beg+end)/2;
while(beg<=end)
{
if(a[mid]==item)
{
loc=mid;
printf("\n item is on %d location ",loc);
getch();
exit();
}
if(item<a[mid])
end=mid-1;
if(item>a[mid])
beg=mid+1;
mid=(beg+end)/2;
}
if(item>=a[n-1])
printf("\n item is not in the list");
getch();
}
```

cnt = n-1



item=3, beg=0, end=2

0<=2 T mid=0+2/2=1

if(a[1]==item)

2==3 F

3<a[1]

3<2 F

if 3>a[1]

3>2 T

beg=1+1=2

11 22 33 44 46

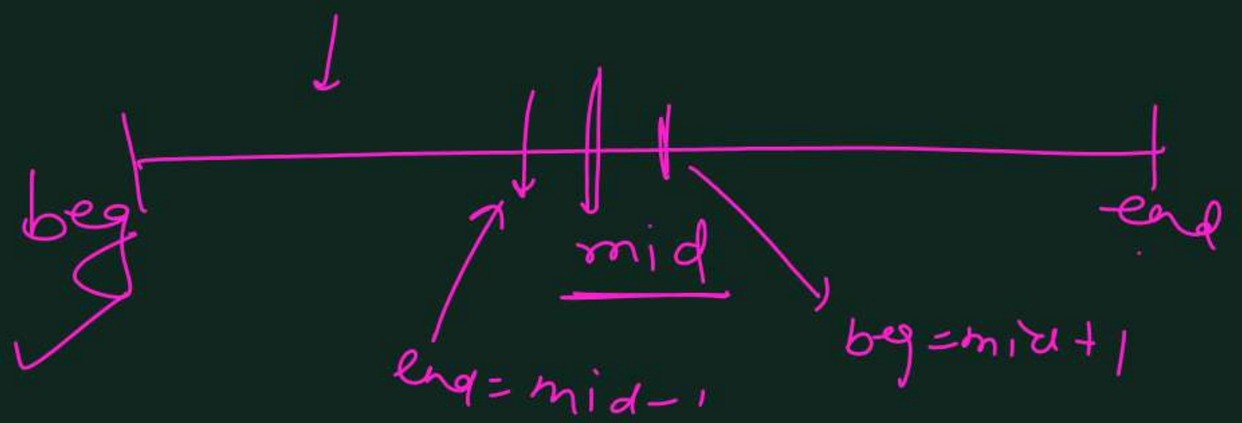
mid=2+2/2=4/2=2

item=SI

while(2<=2) T

a[2]==3

3==3 T


beg
mid
end
end = mid - 1
beg = mid + 1

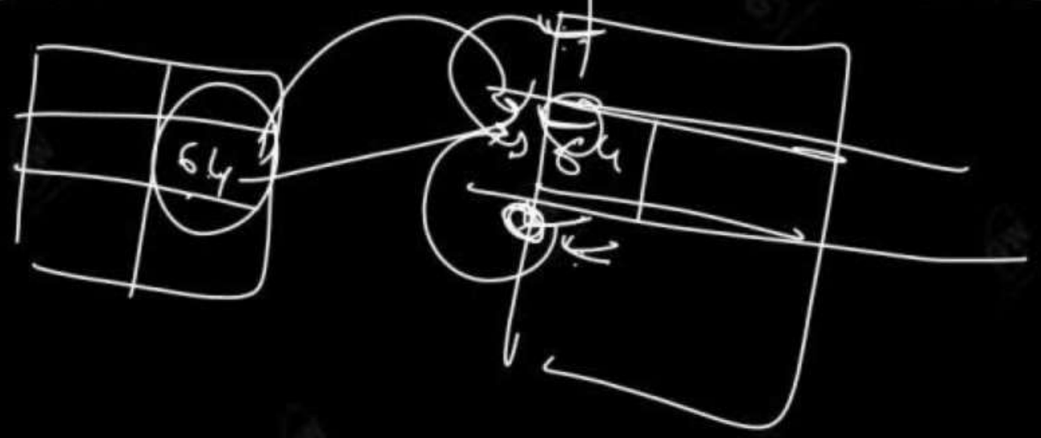
Recursive implementation of Binary Search

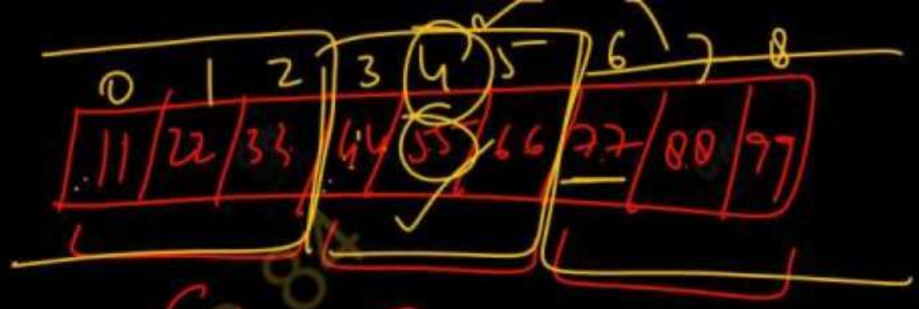
```
int binarysearch ( int a[], int beg, int end, int
item)
{
int mid;
If ( end > = 1 )
{
mid = ( beg + end ) / 2;
If ( a[mid] == item )
return mid;
If ( item < mid )
return binarysearch (a, beg, mid-1, item);
else
return binarysearch (a, mid+1, end, item);
}
return -1;
}
```

```
/* binary search recursive*/
void main()
{
int a[20],n,i,item,loc;
printf("\n how many numbers you want to
input:");
scanf("%d",&n);
if(n>20)
{
printf("\n Invalid input, it should be less than
20");
getch();
exit();
}
printf("\n enter the numbers in ascending
order");
for(i=0;i<n;i++)
{
printf("\n enter number %d:",i+1);
scanf("%d",&a[i]);
}
printf("\n enter item:");
scanf("%d",&item);
loc=binarysearch(a,0,n-1,item);
if(loc==-1)
printf("\n Item not found");
else
printf("\n Item is on %d index",loc);
getch();
}
```

INDEXED SEQUENTIAL SEARCH TECHNIQUE

- This is a combination of two searching methods.
- This searching technique uses sequential and random access searching method.
- It search according to groups.
- This searching technique is useful in searching direct access secondary storage devices.
- The general strategy of an indexed search is that the key is used to search the index, find the relative record position of the associated data and from there only one access is made to find the data.



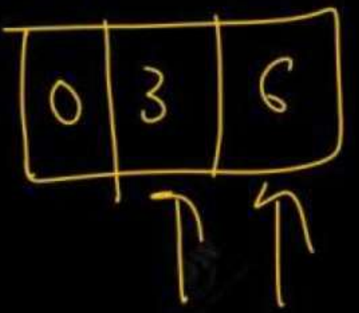


Algorithm: Index Sequential Search

1. Read the element by the user to be search.
2. Divide the array into groups according to the group size.
3. Create index array that contains starting index of the groups.
4. If group is present and first element of that group is less than or equal to key element go to next group
5. Otherwise apply linear search in previous group.
6. Repeat step 4 for all groups.

GS = 3

INDEX:



For example consider the following array-

Index =

Key = 28

GS = 3

Key means the item to search say i.e. 28,

Now choose group size say 3 i.e. first group is from 12 to 17 that is from index 0 to 2, place its starting index in "index", the second group is from 21 to 36 that is from index 3 to 5, place its starting index to "index" , third and last group is from 57 to 99 that is from index 6 to 8, place its starting address to "index".

Now execute the algorithm step by step i.e.-

- Compare the key 28 with the first element of first group, first element of first group is less than key 28, move to second group
- Compare the key 28 with first element of second group, first element of second group is less than key 28, move to the 3rd group
- Compare the first element of third group with key 28 but first element is not less than key 28 then come out of step 4 of algorithm and read the else part.
- The else part says apply linear search in previous group.

SORTING TECHNIQUES

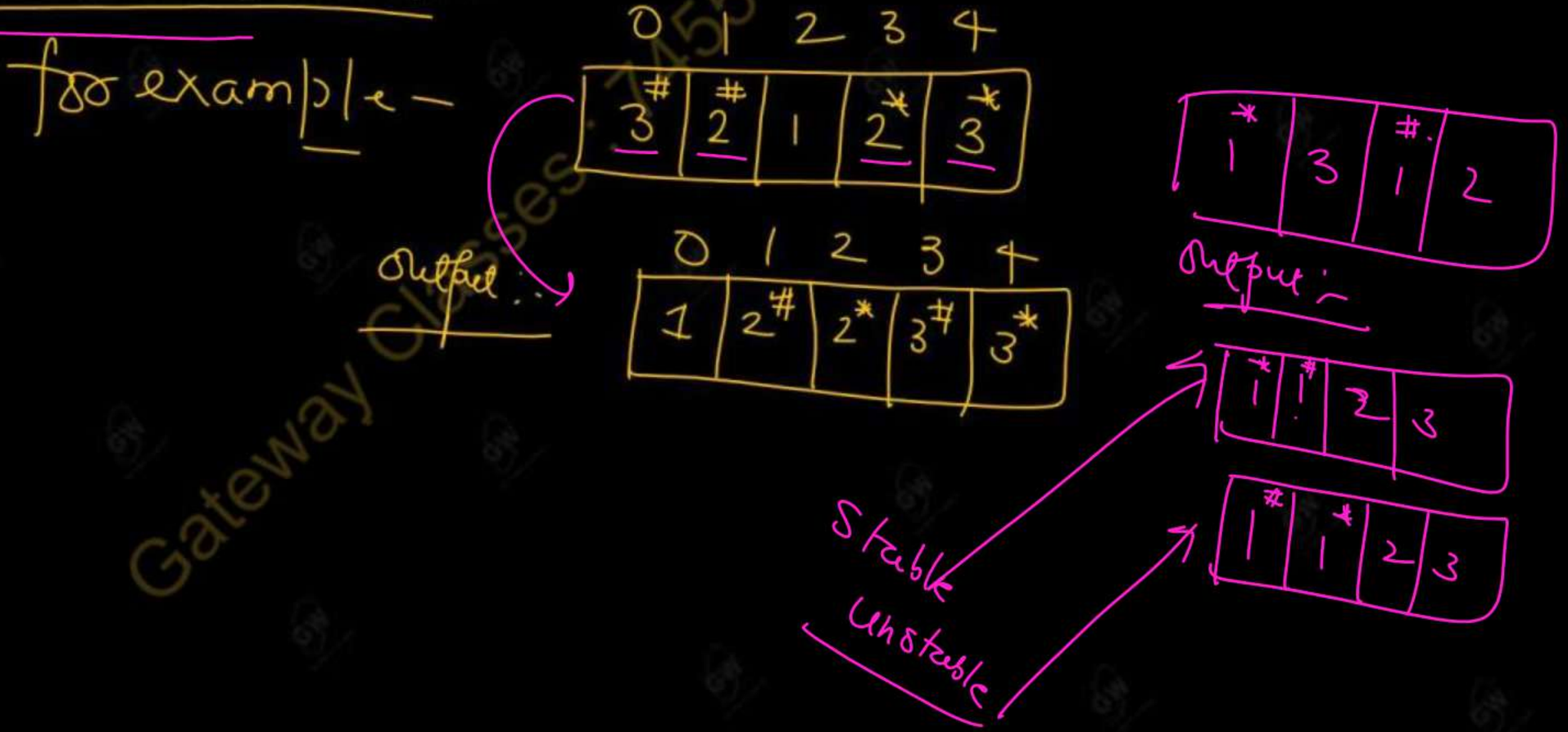
- The efficiency of data handling can often *be* substantially increased if the data are sorted according to some criteria of order.
- For example, *it* would be practically impossible to find a name in *the* telephone directory *if* the name were not alphabetically ordered. The same can be said about dictionaries, book indexes, payrolls, bank accounts, student lists, and other alphabetically organized materials.
- The convenience of using sorted data is unquestionable and must be addressed in computer science as well.
- So, retrieval of information is made easier when it is stored in some predefined order. Sorting is, therefore, a very important computer application activity. Many sorting algorithms are available. Different environments require different sorting methods.

1. Bubble sort
2. Insertion
3. selection
4. Quick
5. heap
6. merge
7. Radix

STABLE AND IN - PLACE SORTING TECHNIQUES

1. **Stable and Unstable Sorting Techniques:** If the sorting algorithm preserves the relative order of any two equal elements, then the sorting algorithm is stable otherwise it is unstable or not stable sorting technique.

for example –



2. In Place and Not in - Place Sorting Techniques :

- Sorting algorithms may require some extra memory for comparison and temporary storage for few data elements.
- If an algorithm does not require an extra memory space, then the algorithm is said to be In-place sorting technique. For example bubble sort, insertion sort, selection sort, heap sort etc.
- On the other hand, If an algorithm requires an extra memory space, then the algorithm is said to be Not In-place sorting technique. For example consider the merge sort technique in which we require additional space or additional temporary array to sort left sub array and right sub array according to the concept of the merge sort algorithm. Another example of not in place sorting algorithm is quick sort algorithm.

INTERNAL AND EXTERNAL SORTING

There are two basic categories of sorting methods:

1. Internal Sorting: (sorting of data items in the main memory)
2. External Sorting: (when the data to be sorted is so large that some of the data is present in the memory and some kept in *auxiliary* memory).

- *Internal* sorting is applied when the entire collection of data to be sorted is small enough so that *the* entire sorting can take place within the main memory.
- The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods.
- *External sorting* methods are applied to larger collection of data which reside on secondary devices.
- Read *and* write access times are a major concern in determining sorting performance of such methods.

VARIOUS INTERNAL SORTING

By internal sorting means we are arranging the numbers within the array only which is in computer primary memory.

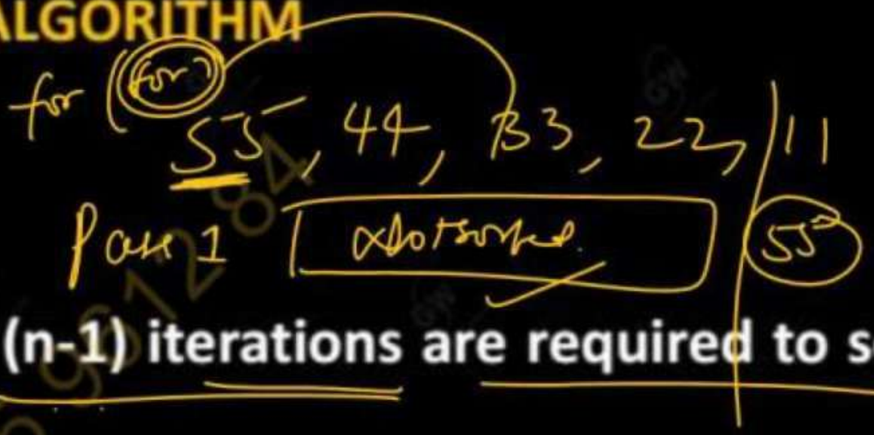
The various internal sorting methods are:

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Quick sort
5. Merge sort
6. Heap sort
7. Radix sort

BUBBLE SORT

- Bubble sort, also known as the Exchange Sort, is a simple sorting algorithm.
- It works by repeatedly stepping through the list to be sorted.
- Comparing two items at a time and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which means the list is sorted.
- The algorithm gets its name from the way smaller elements "bubble to the top (i.e., the beginning) of the list via the swaps.
- In this method, we assume that the lower value items from the set of items are light and bubble up to the top. Because it only uses comparison to operate on elements, it is a *comparison sort*.
- This is the easiest comparison sort to implement.

BASIC IDEA OF BUBBLE SORT ALGORITHM



The idea applied for the bubble sort is as follows:

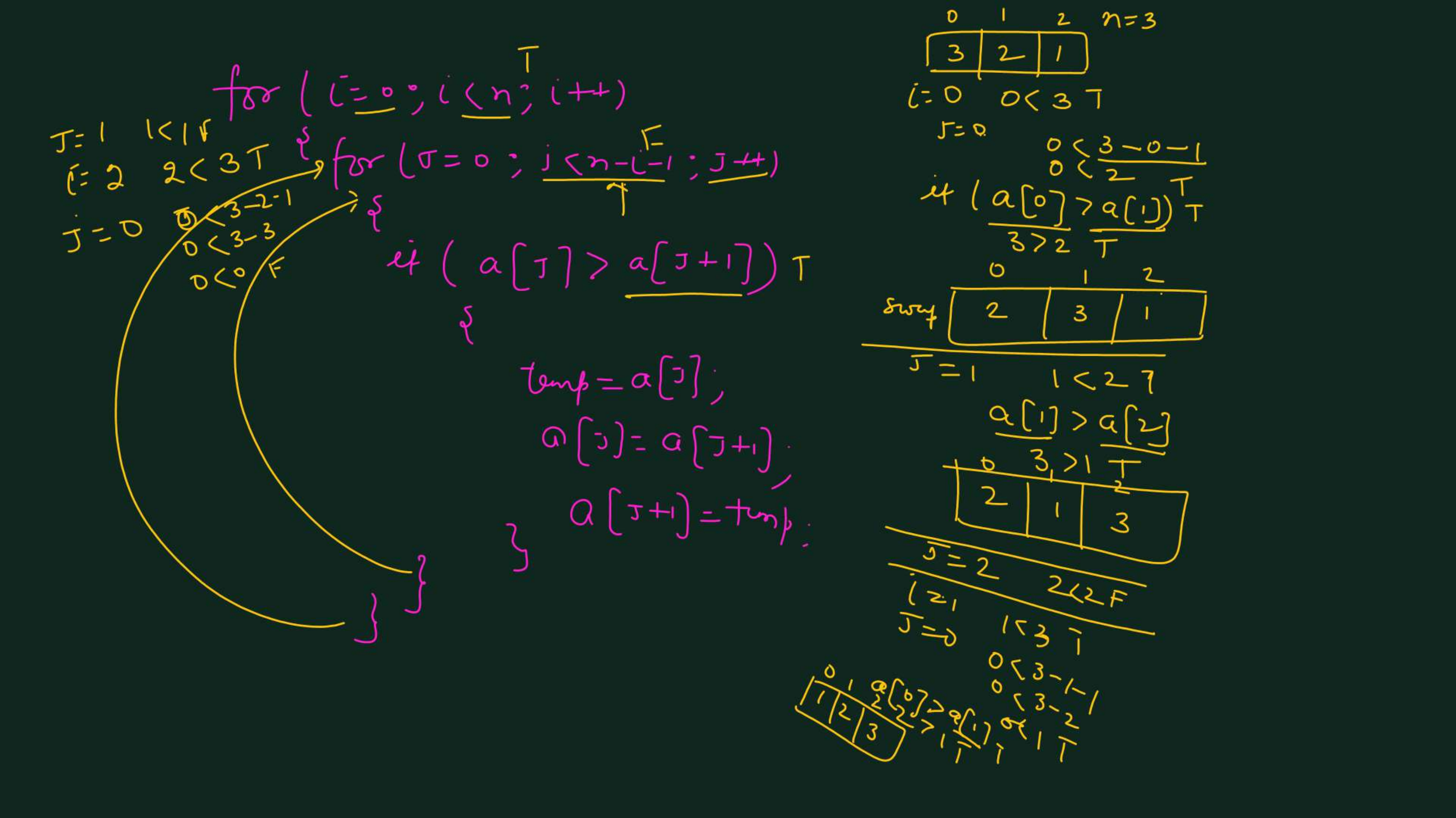
(a) Suppose if the array contains n elements, then (n-1) iterations are required to sort this array.

(b) The set of items in the array are scanned again and again and if any two adjacent items are found to be out of order, they are reversed.

(c) At the end of the first iteration, the lowest value is placed in the first position.

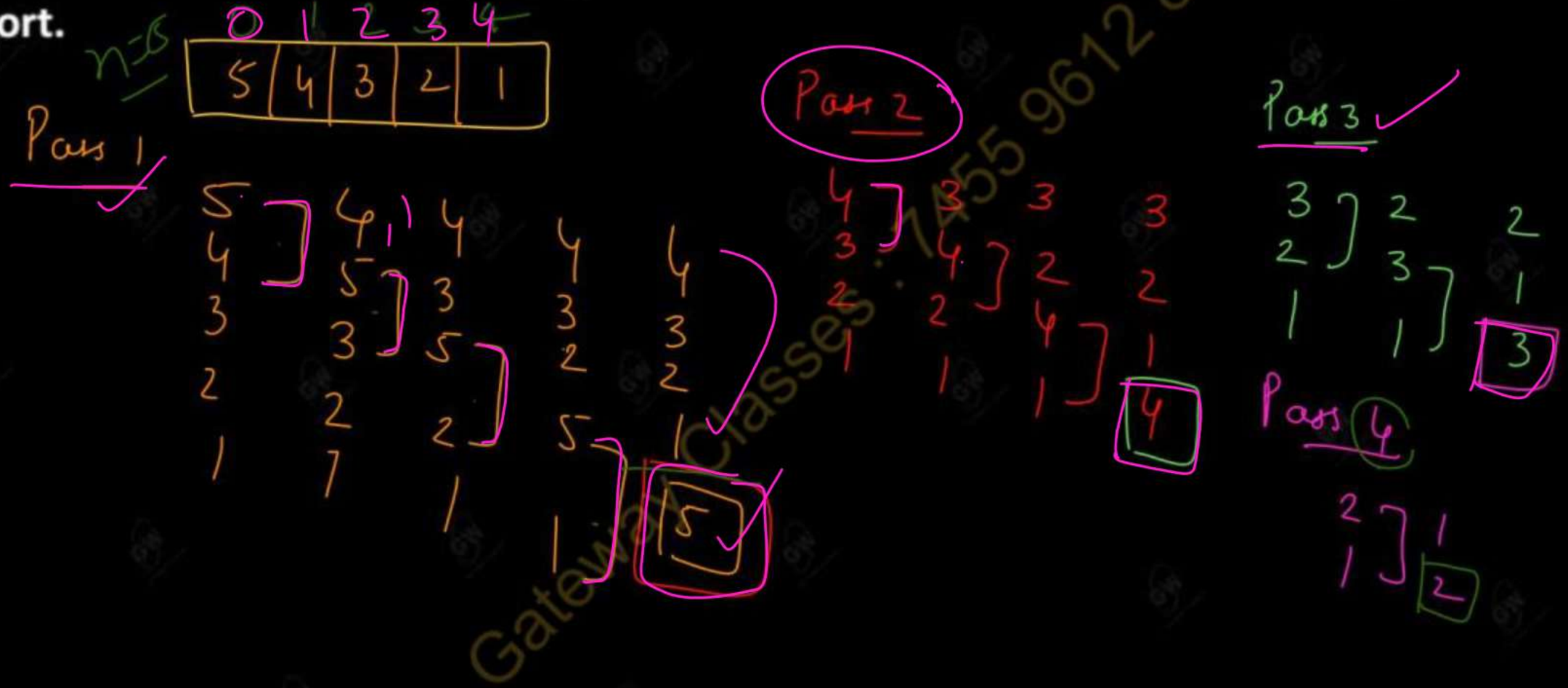
(d) At the end of the second iteration, the next lowest value is placed in the second position and so on.

(e) It is very efficient in large sorting jobs. For n data items, this method requires n(n-1)/2 comparisons.


for (i=0 ; i<n ; i++)
{
for (j=0 ; j<n-i-1 ; j++)
{
if (a[j] > a[j+1])
{
temp = a[j];
a[j] = a[j+1];
a[j+1] = temp;
}
}
}
T
F
J=1 1<1 F
i=2 2<3 T
J=0 0<3-2-1
0<3-3
0<0 F
0 1 2 n=3
3 2 1
i=0 0<3 T
J=0
0<3-0-1
0<2 T
if (a[0] > a[1]) T
3>2 T
0 1 2
swap 2 3 1
J=1 1<2 T
a[1] > a[2]
3>1 T
0 1 2
2 1 3
J=2 2<2 F
i=1 1<3 T
J=0 0<3-1-1
0<3-2
0<1 T
0 1 2
1 2 3
a[0] > a[1]
1 > 1
1 1

Example of Bubble Sort

Let us take the following array of 5 elements to understand the concept of bubble sort.



Algorithm: Bubble Sort (A , N)

This algorithm is used to sort an integer array A which is of N elements in ascending order.

Step 1:- Start.

Step 2:- Repeat steps 3 to 7 For i=0 to N.

Step 3:- Repeat step 4 to 7 for j=0 to N-i-1

Step 4:- if A[j] is greater than A[j+1] then

Step 5:- Set Temp := A[j].

Step 6:- Set A[j] := A[j+1]

Step 7:- Set A[j+1] := Temp

swap

[End of If structure mentioned in step 4]

[End of Step 3 loop]

[End of Step 2 loop]

Step 8: -Stop

C IMPLEMENTATION OF BUBBLE SORT

```
{
① read array for n elements
for (i=0; i<n; i++)
② {
  for (j=0; j<n-i-1; j++)
  {
    if (a[j] > a[j+1])
    {
      temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
    }
  }
}
```

③ write array a[i]

a[5]

0	1	2	3	4
5	4	3	2	1

n = 5

i=0 0<5 T {0, 1, 2, 3,}

4

J=0 j<n-i-1

0<5-0-1

0<4 T

if (a[0] > a[1])

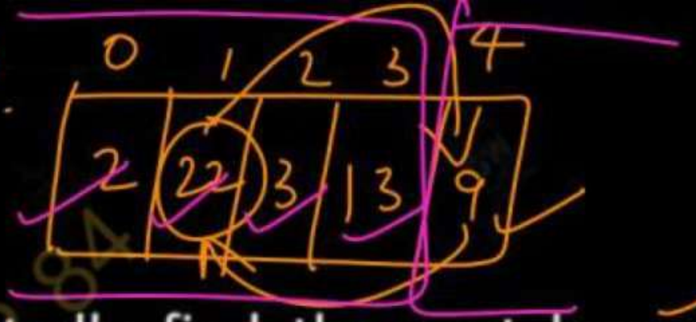
5 > 4 T

temp = a[0] = 5

a[0] = a[1] = 4

a[1] = 5

0	1	2	3	4
4	5	3	2	1



SELECTION SORT

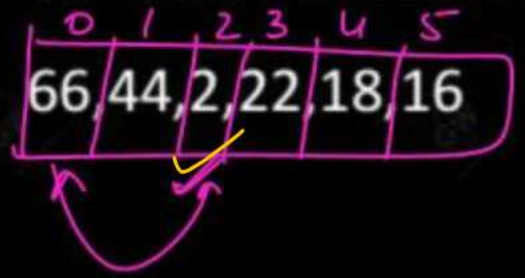
- The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array.
- Assume that we wish to sort the array in increasing order, i.e., the smallest element at the beginning of the array and the largest. at the end of the array.
- We begin by selecting the largest element and moving it to the highest index position.
- We can do this by swapping the element at the highest index and the largest element.
- We then reduce the effective size of the array by one element and repeat the process on the smaller sub- array.

SELECTION SORT

- The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).
- Thus, the selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled.
- In this method, if n elements have to be sorted then (n-1) iterations are required.
- The selection sort has a complexity of $O(n^2)$.

Working of Selection Sort Algorithm

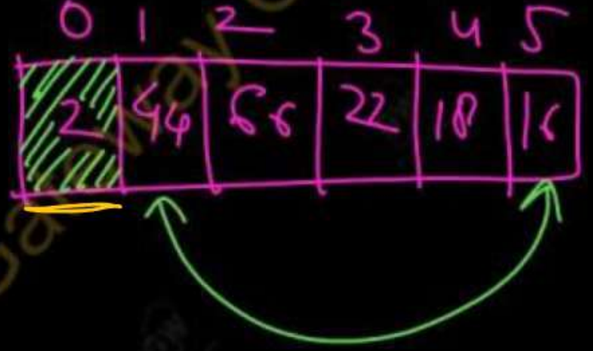
Consider the following 6 elements in the memory –



min = 2, loc = 2

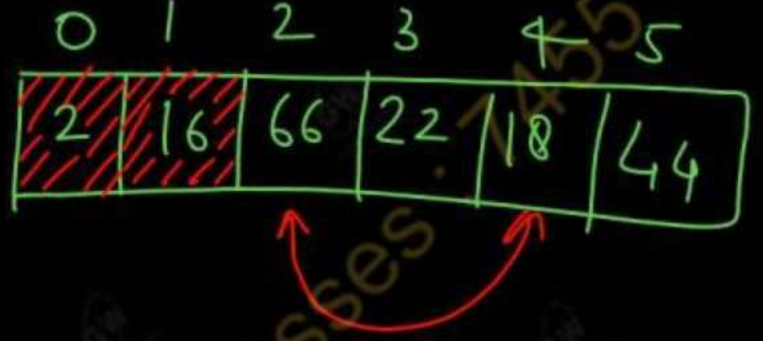
Now apply the selection sort algorithm on these data items to sort them in ascending order.

Step 1: Find the location of minimum element that is 2 at index 2, interchange a[0] and a[2]. After exchange the updated array is as follows:-



Now a[0] is sorted.

Step 2: Now find the location of next minimum element (except the element of subscript a[0]) that element is 16 at index 5, interchange a[1] and a[5]. After exchange the updated array is as follows:-



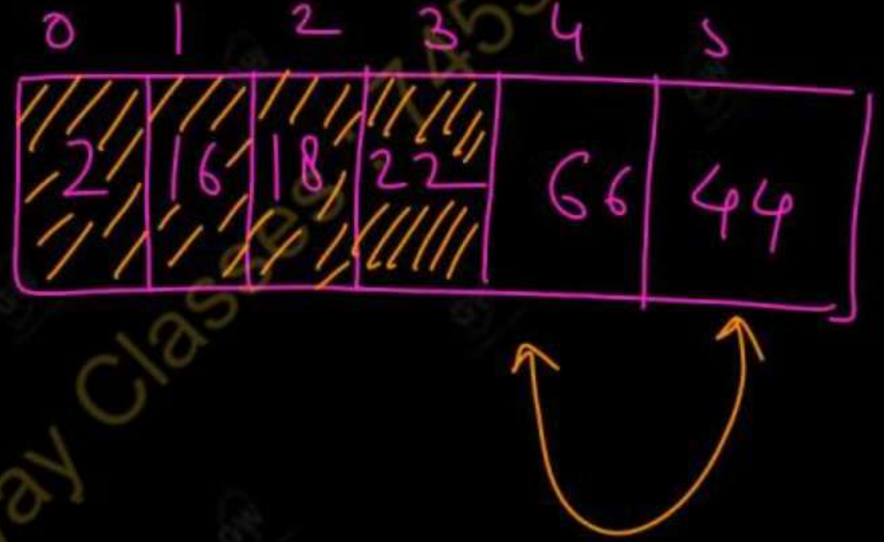
Now a [0] , a [1] is sorted.

Step 3: Now find the location of next minimum element (except elements at subscript a[0] and a[1]) that element is 18 at index 4, now interchange a[2] and a[4]. Updated array is as follows:-

Now a[0] , a[1] , a[2] is sorted.

Step 4: Now find the location of next minimum element (except a [0] , a [1] and a [2]) that is 22 on index 3, now no need to interchange because 22 is on its right position.

The array is as follows:-



In this way a [0] , a [1] , a [2] , a [3] is sorted.

Step 5: Now find the location of next minimum element (except a [0] , a [1] , a [2] and a [3]) that is 44 on index 5, now interchange a [4] and a [5]. The updated array is as follows:-

Now the complete array is sorted.

ALGORITHM SELECTION SORT (A , n)

Step 1: Find the minimum element from the entire array & interchange that minimum element location with a [0]. Now a [0] is sorted.

Step 2: Find the second minimum element of the array except the element at index 0 & interchange with a [1]. Now a [0], a [1] is sorted.

Step 3: Find the third minimum element of the array except the elements at index 1 and 1 and interchange with a [2]. Now a [0], a [1] and a [2] is sorted.

Step n-1: Find the smallest element from the rest of the elements and interchange with a [n – 2]. Now a [0] , a [1] , a [n – 2] , a [n – 1] is sorted.

C IMPLEMENTATION OF SELECTION SORT

```
void main()
{
int a[20],n, i, min, loc, temp, j;
clrscr();
printf("\n How many numbers you want to input for the
array:");
scanf("%d",&n);
if(n>20)
{
printf("\n Invalid input, it should be less then 20");
getch();
exit();
}
for(i=0;i<n;i++)
{
printf("\n Enter the number %d:",i+1);
scanf("%d",&a[i]);
}
for(i=0;i<n;i++)
{
min=a[i];
location=i;
for(j=i+1;j<n;j++)
{
if(a[j]<min)
{
min=a[j];
loc=j;
}
if(loc!=i)
{
temp=a[i];
a[i]=a[loc];
a[loc]=temp;
}
}
printf("\n the sorted array is follows");
for(i=0;i<n;i++)
printf("\n %d",a[i]);
getch();
}
```

Read

Reading of an array

Pseudocode ✓

ALGORITHM: SELECTION SORT (A, N, MIN ...)

This

Step 1 :— Read an array A of N element.

Step 2 :— Repeat steps 3 to 12 for i = 0 to N−1.

Step 3 :- Set min := A[i]

Step 5 :- Set loc := i

Step 6 :- Repeat steps 7 to 12 for J = i+1 to N

Step 7 :- If [A[j] < min] then

Step 8 :- Set min = a[j] & loc = j.

[END OF STEP 7 Condition]

Step 9 :- if loc != i then.

Step 10 :- set temp := a[i].

Step 11 :- set a[i] := a[loc].

Step 12 :- set a[loc] := temp.

[End of step 9 condition].

[End of step 6 loop].

[End of step 2 loop].

Step 13 :- Write the resultant / sorted array.

Step 14 :- Stop.

```
for (i=0 ; i<n-1 ; i++)
{   min = a[i];
    loc = i;
    for ( j = i+1 ; j<n ; j++)
    {  if ( a[j] < min)
       {
           min = a[j];
           loc = j;
       }
    if ( loc != i);
    { temp = a[i];
      a[i] = a[loc];
      a[loc] = temp;
    } }
}
```

	0	1	2
a	3	2	1

n=3

i=0 0<3-1 0<2 True

min = a[0] = 3, loc = 0

j = 1 1<3 T

if a[1] < min

a[1] < 3

2 < 3 True

min = 2 loc = 1

if (1 != 0) T

{ temp = a[0] = 3

a[0] = a[1] = 2

a[1] = 3

}

	0	1	2
	2	3	1

J = 2 2<3 T

if a[2] < 2

1 < 2 True

min = a[2] = 1

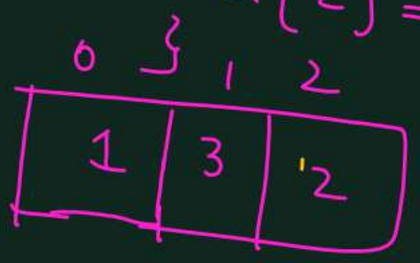
loc = 2

if (2 != 0) True

{ temp = a[0] = 2

a[0] = a[2] = 1

a[2] = 2 }



J = 3 3<3 F

---

i = 1 1<2 T

min = a[1] ⇒ 3, loc = 1

J = 2 2<3 T

if a[2] < 3

2<3 T

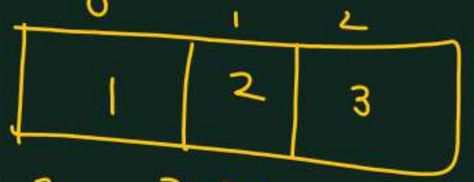
min = a[2] ⇒ 2 loc = 2

if (2 != 1) T

{ temp = a[1] = 3

a[1] = a[2] ⇒ 2

a[2] = 3



J = 3 3<8 F

i = 2

2 < 8-1

2<2 F

```
for (i=0; i<n; i++)
{
    min = a[i];
    loc = i;
    for (j=i+1; j<n; j++)
    {
        if (a[j] < min)
        {
            min = a[j];
            loc = j;
        }
    }
    if (loc != i)
    {
        temp = a[i];
        a[i] = a[loc];
        a[loc] = temp;
    }
}
```

	0	1	2
n=3	3	2	1

i=0 0<3 T

min = a[0] = 3

loc = 0

j = 1 1<3 T

if a[1] < 3

2 < 3 T

min = a[1] = 2

loc = 1

if 1 != 0 T

temp = a[0] = 3

a[0] = 2

a[1] = 3

	0	1	2
	2	3	1

INSERTION SORT

- **Insertion sort** is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
- The insertion sort works just like its name suggest it inserts each item its proper place in the initial list.

Consider the following procedure-

In Insertion sort we assume that first element of the array is already sorted and then read the second element of the array. Now there are two possible position of second element.

- Either it is on the right position or
- It can come before the first element.

After arrangement of first and second element, now scan second element, then scan the third element of the array. There are three possible positions of this third element.

- It is on the right position.
- It can come between first and second element of the array or
- It can come before the first element of the array.

In this manner the complete array will be sorted.

ALGORITHM : INSERTION SORT (A, n)

Suppose an array A of n elements A [0] , A [1] , A [2] ,……, A [n – 1] is in memory.

Scan the list of items from left to right as per the following–

Step 1: A [0] is sorted itself.

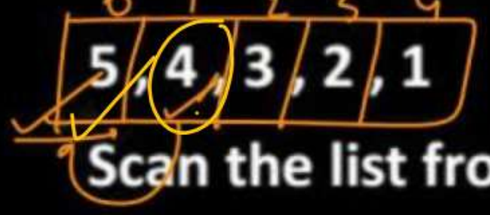
Step 2: A [1] is compared to A [0] and can be inserted before A [1], now A [0], A [1] is sorted.

Step 3: A [2] is compared with A [0] and A[1] and can be inserted before A [0], between A [0] and A [1] and after A [1]. Now A [0], A [1] and A [2] is sorted.

Step n: A [n – 1] is compared with all elements of previous sorted array. This item can be inserted to its proper position. Now A [0] , A [1], A [2] ,……, A [n – 1] is sorted.

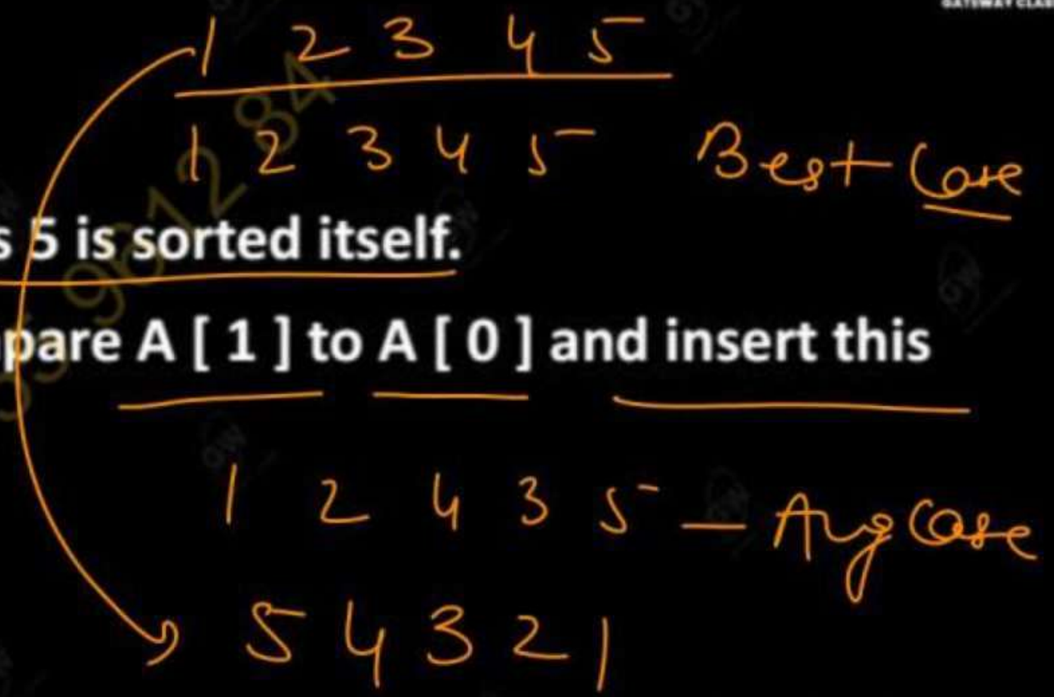
EXAMPLE : INSERTION SORT

Consider the following array A of 5 elements.



Scan the list from left to right, assume A [0] that is 5 is sorted itself.

Now read the second element A [1] that is 4, Compare A [1] to A [0] and insert this element to its proper place as-



Now read A [2] i.e. 3 and compare with previously sorted array i.e. A [0] and A [1] and place this element A [2] to its proper place as

After applying the same process on rest of the elements the sorted array is as follows:-

1 , 2 , 3 , 4 , 5

C IMPLEMENTATION OF INSERTION SORT

```
#include<stdio.h>
void main()
{
int a[20];
int n,i;
clrscr();
printf("\n How many numbers you want to input
for the array:");
scanf("%d",&n);
if(n>20)
{
printf("\n Invalid input, it should be less then
20");
getch();
exit();
}
for(i=0;i<n;i++)
{
printf("\n Enter the number %d:",i+1);
scanf("%d",&a[i]);
}
for(i=0;i<n;i++)
{
temp=a[i];
j=i-1;
while((temp<a[j]) && (j>=0))
{
a[j+1]=a[j];
j--;
}
a[j+1]=temp;
}
printf("\n the sorted array is follows");
for(i=0;i<n;i++)
printf("\n %d",a[i]);
getch();
}
```

```
for (i=0; i<n; i++)
{
    temp = a[i];
    j = i++; i-1;
    while ( temp < a[i] ) && (j >= 0))
    {
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = temp;
}
```

a =

0	1	2
3	2	1

n=3

i=0 0<3 T

temp = a[0] = 3

j = 0-1 = -1

a[0] = 3

0	1	2
3	2	1

i=1 1<3 T

temp = a[1] = 2

j = 1-1 = 0

while (2 < a[0] && j >= 0

2 < 3 T

0 >= 0 T

a[j+1] = a[j]

a[1] = a[0] ⇒ 3

j = -1

a[0] = 2

0	1	2
2	3	1

---

i = 2 2 < 3 T

temp = a[2] = 1

j = 2 - 1 = 1

1 < a[1]

1 < 3 T && 1 >= 0 T

a[2] = a[1] = 3

j = 0

1 < a[0]

1 < 2 T && 0 >= 0 T

{

a[1] = a[0] = 2

j = -1

a[0] = 1

0	1	2
1	2	3

```
for(i=0; i<n; i++)
{
    temp = a[i];
    j = i-1;
    while (temp < a[j] && j>=0)
    {
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = temp;
}
```

T

F

n=3

	0	1	2
a[3]	3	2	1

i=0 0<3 T

temp = a[0] = 3

j = 0-1 = -1

while (temp < a[-1])

X

a[-1+1] = temp

a[0] = 3

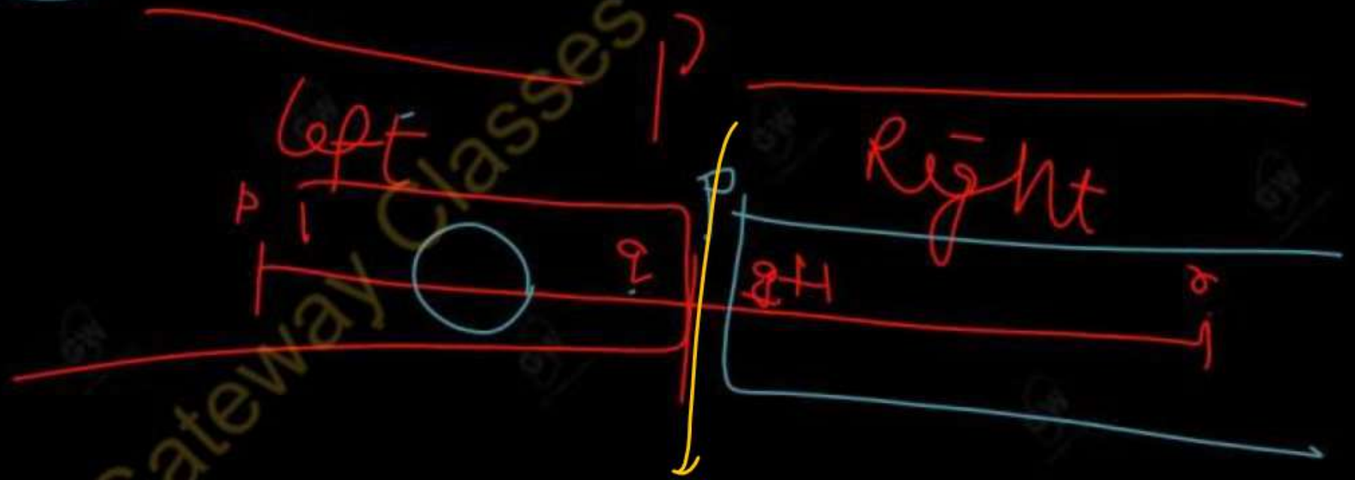
	0	1	2
	3	2	1

QUICK SORT

- C.A.R. Hoare implements quick sort by *divide and conquer method.*
- *Divide and conquer* means divide the problem into two small problems and then those two small problems into two small ones and so on.
- In Quick sort, we divide the original list into two sub-lists. We choose the item from list called key or *pivot* from which all the left side of elements are smaller and all the right side of elements are greater than that element.
- Generally, quicksort is not a stable sorting algorithm. Stable version of quick sort is very expensive. Stable sorting preserves the relative ordering of duplicate elements.
- The name "Quick-sort" stems from the fact that it can sort a list of data elements substantially faster (twice or three times faster) than any other sorting method. Quicksort is one of the most efficient sorting algorithms. It works by breaking an array (partition) into smaller ones and swapping (exchanging) the smaller ones, depending on a comparison with the 'pivot' element picked.

QUICK SORT

- So we can create two lists, one list on the left side of pivot element and the *second* list is on light side of the pivot.
- Thus quick sort works by partitioning a given array A[p…r] into two non-empty sub arrays A[p….q] and a[q+1….r] such that every key in A[p..q] is less than or equal to every key in A[q+1…r]. Then the two sub-arrays are sorted by recursive calls to Quick sort.



Quick sort : An Example

- Quicksort is an algorithm of the divide-and-conquer type i.e. the problem of sorting a set is reduced to the problem of sorting two smaller sets.
- See this reduction step in the following example-
- Suppose A is the following list of 12 numbers:

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

44 , 33 , 11 , 55 , 77 , 90 , 40 , 60 , 99 , 22 , 88 , 66

P

Right to left

- The reduction step of quicksort algorithm finds the final position of one of the numbers; in this example, we use the first number, 44. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list-

22 , 33 , 11 , 55 , 77 , 90 , 40 , 60 , 99 , 44 , 88 , 66

left to right

- (Observe that the numbers 88 and 66 to the right of 44 are each greater than 44.) Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each element with 44 and stopping at the first element greater than 44. The number is 55. Interchange 44 and 55 to obtain the list-

  22 , 33 , 11 , 44 , 77 , 90 , 40 , 60 , 99 , 55 , 88 , 66

  ← Right to left

- (Observe that the numbers 22, 33 and 11 to the left of 44 are each less than 44.) Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. The number if 40. Interchange 44 and 40 to obtain the list-

  22 , 33 , 11 , 40 , 77 , 90 , 44 , 60 , 99 , 55 , 88 , 66

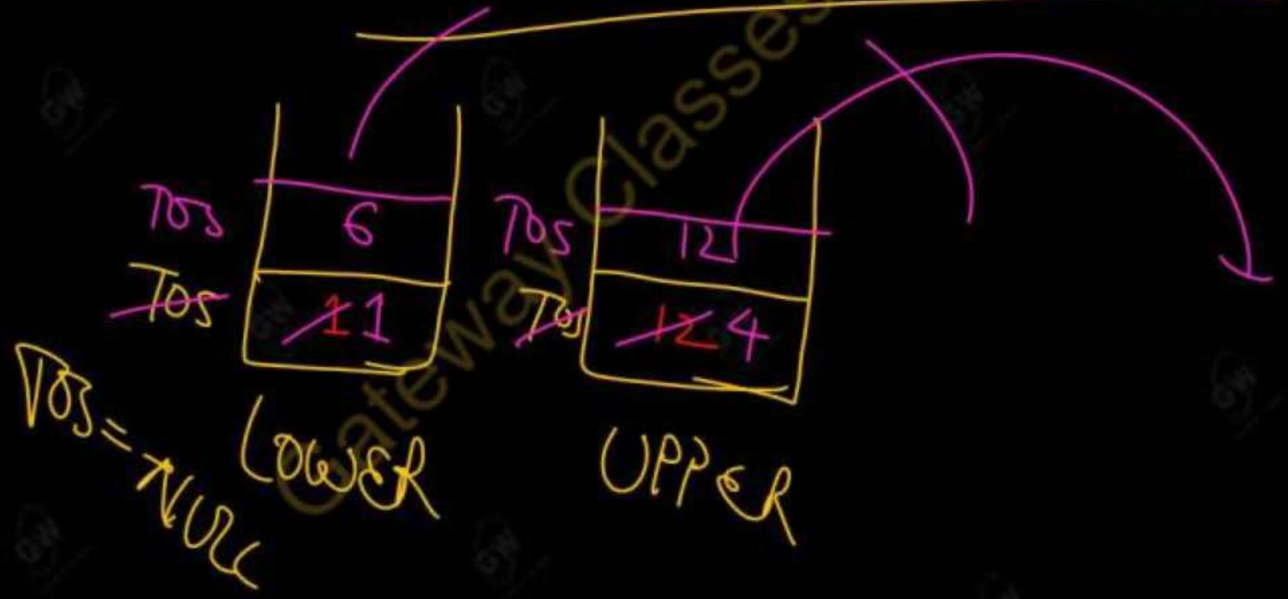
  left to right

- (Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

22 , 33 , 11 , 40 , 44 , 90 , 77 , 60 , 99 , 55 , 88 , 66

- (Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such a number before meeting 44. This means all numbers have been scanned and compared with 44, and all numbers greater than 44 now from the sub-list of numbers from the right of 44, as follows:

Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to task of sorting each of the above sub-lists.



- The above reduction step is repeated with each sub-list containing 2 or more elements.
- Since we can process only one sub-list as time, we must be able to keep track of some sub-lists for future processing.
- This is accomplished by using 2 stacks, called LOWER and UPPER, to temporarily hold such sub-lists. That is, the address of the first and last elements of each sub-list, called its boundary values, are pushed onto the stacks LOWER and UPPER, respectively; and the reduction step is applied to a sub-list only after its boundary values are removed from the stacks.

- The following example illustrates the way the LOWER and UPPER are used.
- Consider the above list A with n = 12 elements.
- The algorithm begins by pushing the boundary values 1 and 12 of A onto the stacks to yield -

  LOWER = 1 UPPER = 12
- In order to apply the reduction step, the algorithm first removes the top values 1 and 12 from the stacks, leaving - LOWER = (empty) UPPER = (empty)
- And then applies the reduction step to the corresponding list A [1] , A [2] , ……., A [12]. The reduction step, as executed above, finally places the first element , 44 , in A [5]. Accordingly , the algorithm pushes the boundary values 1 and 4 of the first sub-list and the boundary values 6 and 12 of the second sub-list on to the stacks to yield -

LOWER : 1 , 6 UPPER : 4 , 12

- In order to apply the reduction step again, the algorithm removes the top values, 6 and 12 from the stacks. Leaving -

LOWER : 1 UPPER : 4

- and then applies the reduction step to the corresponding sub-list A [6], a [7] ,……., a [12].
- The reduction step changes this list as per the following.

A[6],	A[6],	A[8],	A[9],	A[10],	A[11],	A[12]
90,	77,	60,	99,	55,	88,	66
66,	77,	60,	99,	55,	88,	90
66,	77,	60,	90,	55,	88,	99
66,	77,	60,	88,	55,	90,	99

Observe that the second sub-list has only one element. Accordingly, the algorithm pushes only the boundary values 6 and 10 of the first sub-list onto the stack to yield -

LOWER: 1 , 6 UPPER: 4 , 10

And so on, The algorithm ends when the stacks do not contain any sub-list to be processed by the reduction step.

Algorithm QUICK SORT

The quick sort algorithm is divided into two parts. The first part gives a procedure, called “QUICK”, which executes the reduction step of the algorithm and the second part uses “QUICKSORT” to sort the entire list.

QUICK (A , N , BEG , END , LOC)

Here A is an array with N elements.

Parameters BEG and END contain the boundary values of the sub-list of A to which this procedure applies.

LOC keeps track of the position of the first element A [BEG] of the sub-list during the procedure.

The local variable LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

1. [Initialize] Set LEFT : = BEG , RIGHT : = END and LOC : = BEG.
2. [Scan the list from right to left]

    (a) Repeat while A [LOC] < = A [RIGHT], and LOC != RIGHT:

        RIGHT = RIGHT – 1.

    [End of Loop]

(b) If LOC = RIGHT , then : return.

(c) If A [LOC] > A [RIGHT], then:

(i) [interchange A [LOC] and A [RIGHT]]

TEMP : = A [LOC] ,

A [LOC] : = A [RIGHT],

A [RIGHT] : = TEMP.

swap

(ii) Set LOC : = RIGHT.

(iii) Go to Step 3.

[End of if structure.]

3. [Scan the list from left to right]

(a) Repeat while A [LEFT] <= A [LOC] and LEFT != LOC:

LEFT = LEFT + 1.

[End of loop.]

(b) If LOC = LEFT, then : Return.

(c) If A [LEFT] > A [LOC], then

(i) [interchange A [LEFT] and A [LOC]]

TEMP : = A [LOC],

A [LOC] : = A [LEFT],

A [LEFT] := TEMP.

(ii) Set LOC := LEFT

(iii) Go to Step 2.

[End of If structure]

Algorithm Quick_Sort :

This algorithm sorts an array A of N elements.

1. [Initialize] TOP : = NULL.
2. [Push boundary values of A onto stacks when A has 2 or more elements]

   If N > 1, then: TOP = TOP + 1, LOWER [1] := 1 , UPPER [1] := N.

3. Repeat Steps 4 to 7 while TOP ! = NULL.
4 [pop sub-list from stacks]

   Set BEG : = LOWER [TOP] , END : = UPPER [TOP],

   TOP : = TOP – 1.

5. Call QUICK (A , N, BEG, END, LOC).

6. [Push left sub-list onto stacks when it has 2 or more elements.]

if BEG < LOC – 1 , then:

TOP : = TOP + 1, LOWER [TOP] : = BEG,

UPPER [TOP] = LOC – 1.

[End of if structure.]

7. [Push right sub-tree onto stacks when it has 2 or more elements.]

if LOC + 1 < END, then :

TOP : = TOP + 1, LOWER [TOP] : = LOC + 1,

UPPER [TOP] := END.

[End of if structure.]

[End of step 3 loop.]

8. Exit

❑ Suppose A is the following list of 12 numbers:

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

44 , 33 , 11 , 55 , 77 , 90 , 40 , 60 , 99 , 22 , 88 , 66

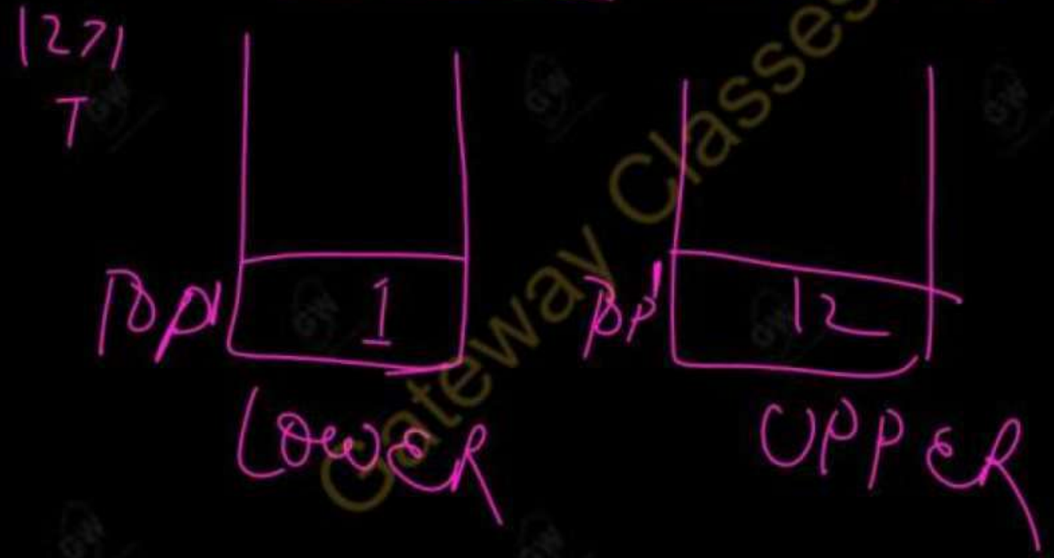
Execute the algorithm on A –

1. [Initialize] TOP : = NULL.

Set TOP = NULL

2. [Push boundary values of A onto stacks when A has 2 or more elements]

If N > 1, then: TOP = TOP + 1, LOWER [1] := 1 , UPPER [1] := N.



- Suppose A is the following list of 12 numbers:

  44 , 33 , 11 , 55 , 77 , 90 , 40 , 60 , 99 , 22 , 88 , 66

  10
  N

Execute the algorithm on A –

Step - 3. Repeat Steps 4 to 7 while TOP ! = NULL.

T

1 TOP ! = NULL T

Step – 4. [pop sub-list from stacks]

Set BEG : = LOWER [TOP] , END : = UPPER [TOP],

TOP : = TOP – 1.

BEG = 1 END = 12

TOP = 0

5. Call QUICK (A , N, BEG, END, LOC).

QUICK (A , N , BEG , END , LOC)

Here A is an array with N elements.

Parameters BEG and END contain the boundary values of the sub-list of A to which this procedure applies.

LOC keeps track of the position of the first element A [BEG] of the sub-list during the procedure.

The local variable LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

1. [Initialize] Set LEFT : = BEG , RIGHT : = END and LOC : = BEG.

left = 1 Right = 12 Loc = 1

2. [Scan the list from right to left]

(a) Repeat while A [LOC] < = A [RIGHT], and LOC != RIGHT:

RIGHT = RIGHT – 1.

A[1] <= A[12) && Loc != Right

44 <= 66 T

1 != 12 T

Right = 11

[End of Loop]

5. Call QUICK (A , N, BEG, END, LOC).

6. [Push left sub-list onto stacks when it has 2 or more elements.]

    if BEG < LOC – 1 , then:

    TOP : = TOP + 1, LOWER [TOP] : = BEG,

    UPPER [TOP] = LOC – 1.

    [End of if structure.]

7. [Push right sub-tree onto stacks when it has 2 or more elements.]

    if LOC + 1 < END, then :

    TOP : = TOP + 1, LOWER [TOP] : = LOC + 1,

    UPPER [TOP] := END.

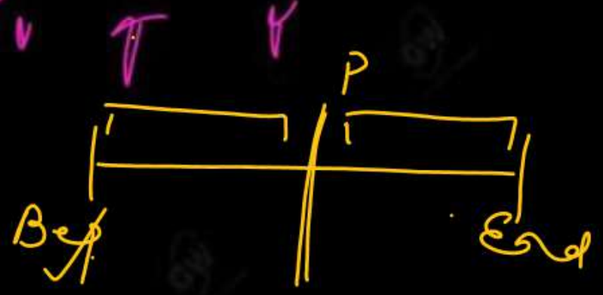
    [End of if structure.]

[End of step 3 loop.]

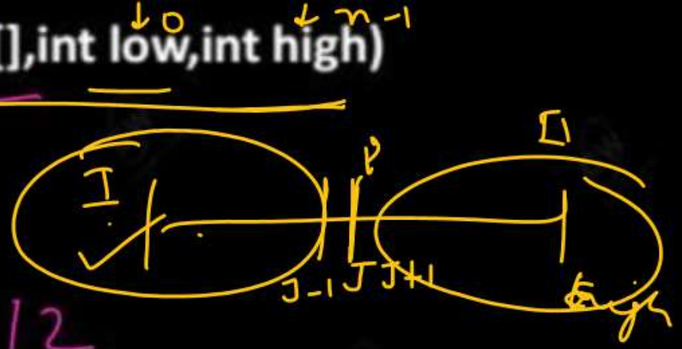
8. Exit

RECURSIVE IMPLEMENTATION OF QUICK SORT

```
#include<stdio.h>
void quicksort(int a[],int low, int high);
int partition(int a[],int low,int high);
int a[20],n;
void main()
{
int i,j;
clrscr();
printf("\n How many numbers you want to input forprintf("\n the sorted array is as follows:");
the array:");
scanf("%d",&n);
if(n>20)
{
printf("\n Invalid input, it should be less then 20");
getch();
exit();
}
for(i=0;i<n;i++)
{
printf("\n Enter the number %d:",i+1);
scanf("%d",&a[i]);
}
printf("\n the original array is as follows:");
for(i=0;i<n;i++)
printf("\n%d",a[i]);
quicksort(a,0,n-1);
for(i=0;i<n;i++)
printf("\n%d",a[i]);
getch();
}
```



```
void quicksort(int a[],int low,int high)
{
int j;
if(low<high)
{
j=partition(a,low,high);
quicksort(a,low,j-1);
quicksort(a,j+1,high);
}
}
int partition (int a[],int low,int high)
{
int i,j,temp,key;
key=a[low];
i=low+1;
j=high;
while(1)
{
while(i<high && key>a[i])
i++;
while(key<a[j])
j--;
if(i<j)
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
else
{
temp=a[low];
a[low]=a[j];
a[j]=temp;
}
return j;
}
}
```



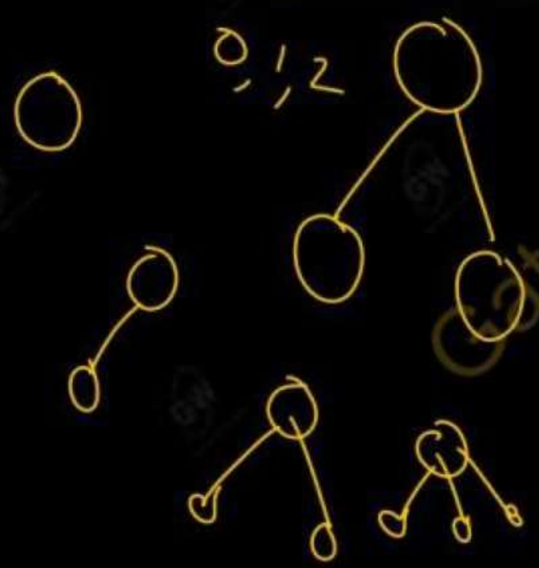
0 12

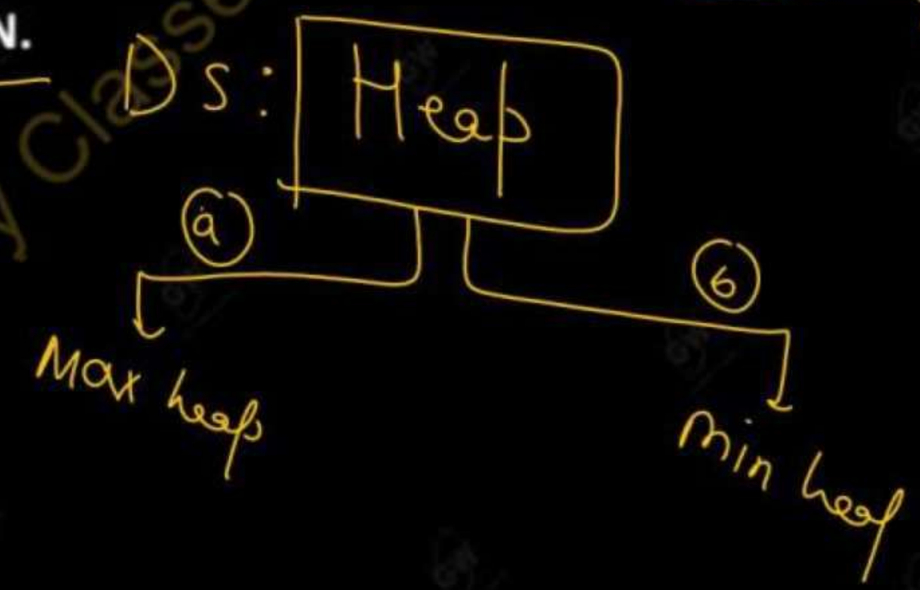
a, 0, 4

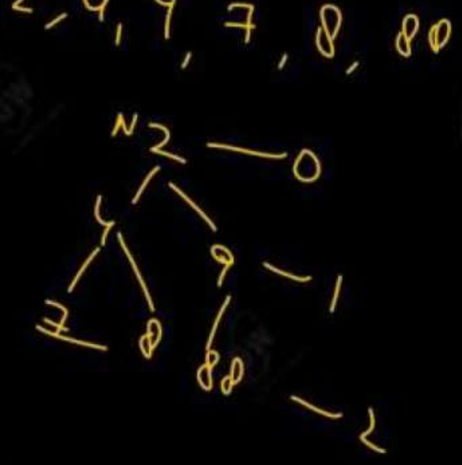
a, 5, 12

HEAP SORT

- Heap sort *was* invented by John Williams.
- Before discussing heap sort, we will begin by defining a new structure, the heap.
- We can define minimum heap and maximum heap also min heap and max heap respectively.
- Suppose H is a complete binary tree with n elements. Then H is called a heap, or a max-heap, if each node N of H has the following property: The value at N is greater than or equal to the value at each of the children of N. Accordingly, the value at N is greater than or equal to the value at any of the descendants of N.





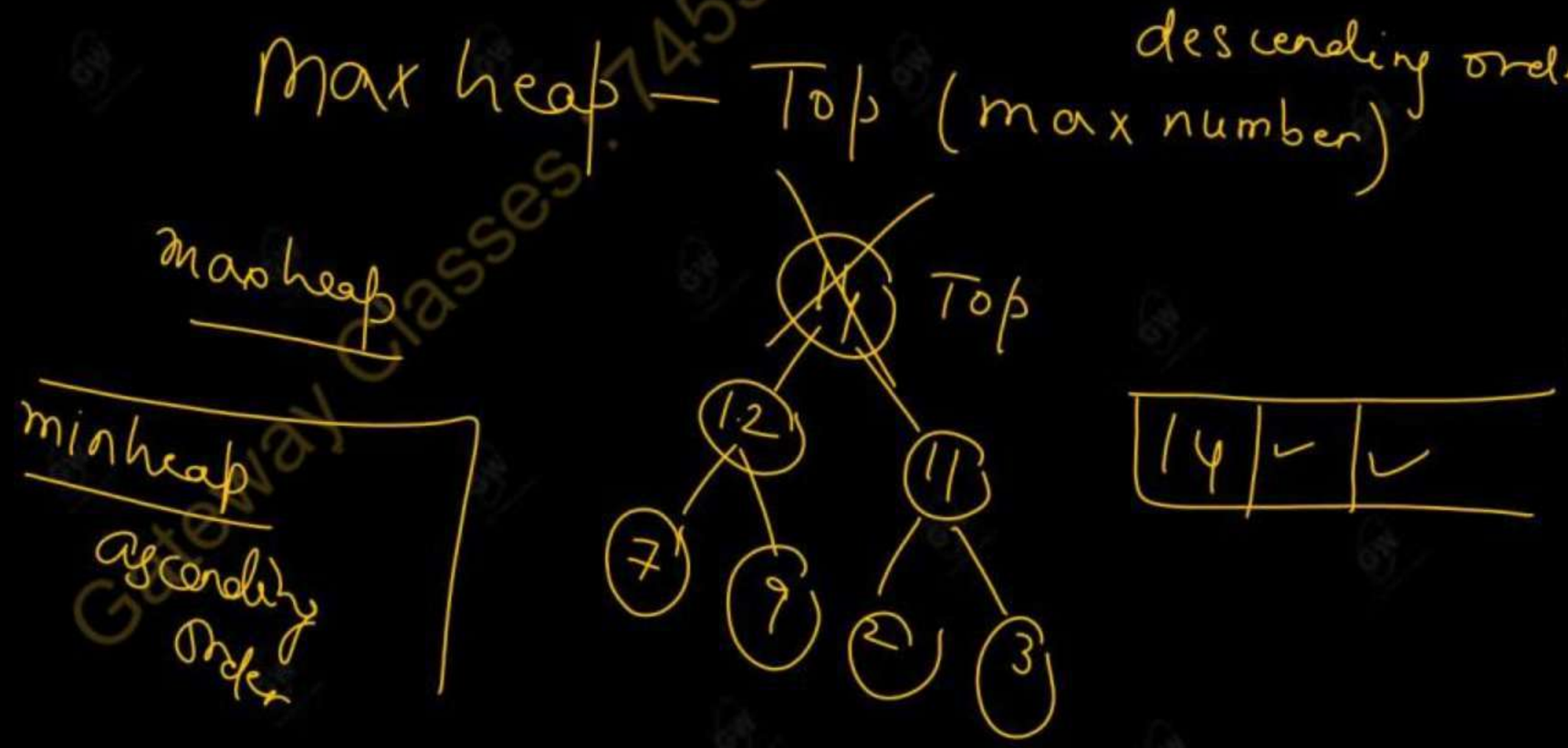


Heap → maxheap ✓ Root — highest

→ minheap ✓ Root → lowest

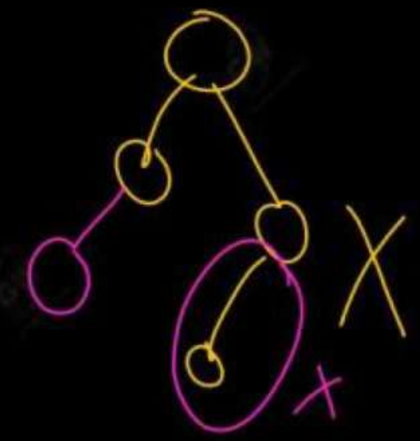
HEAP SORT

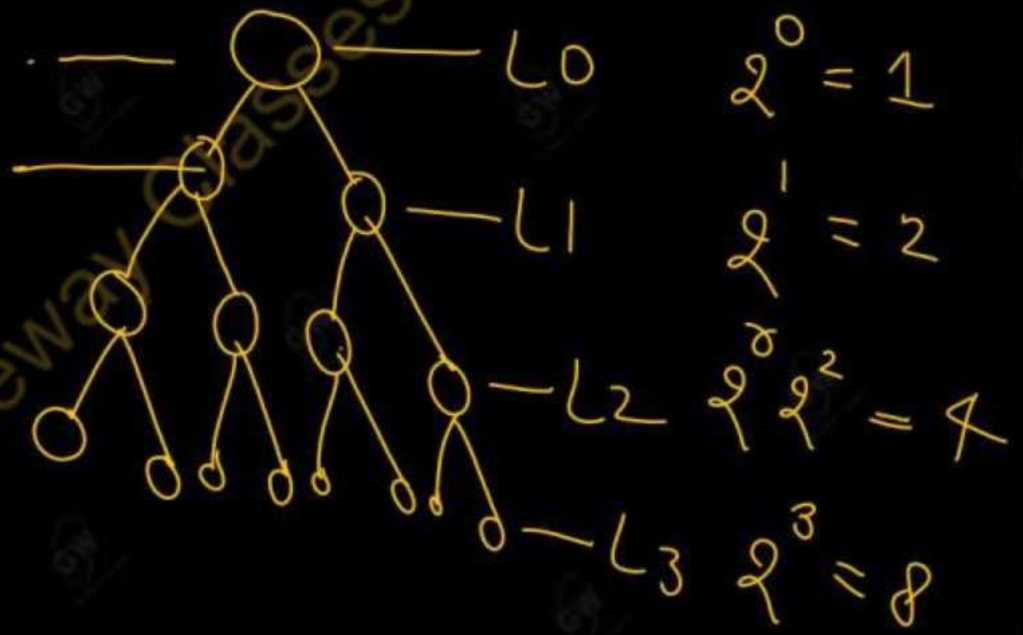
- A min-heap is defined as: the value at N is less than or equal to the value at any of the children of N.
- The definition of a max heap implies that one of the largest elements is at the root of the heap. If the elements are district, then the root contains the largest element.

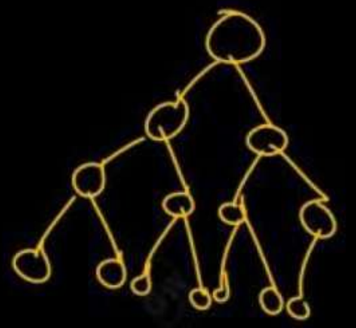


Complete Binary Tree

- Consider a binary tree T.
- Each node of T can have at most 2 children.
- Level r of T can have at most 2^r nodes. where r = level
- The tree is said to be complete if all its levels, except possibly the last level, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

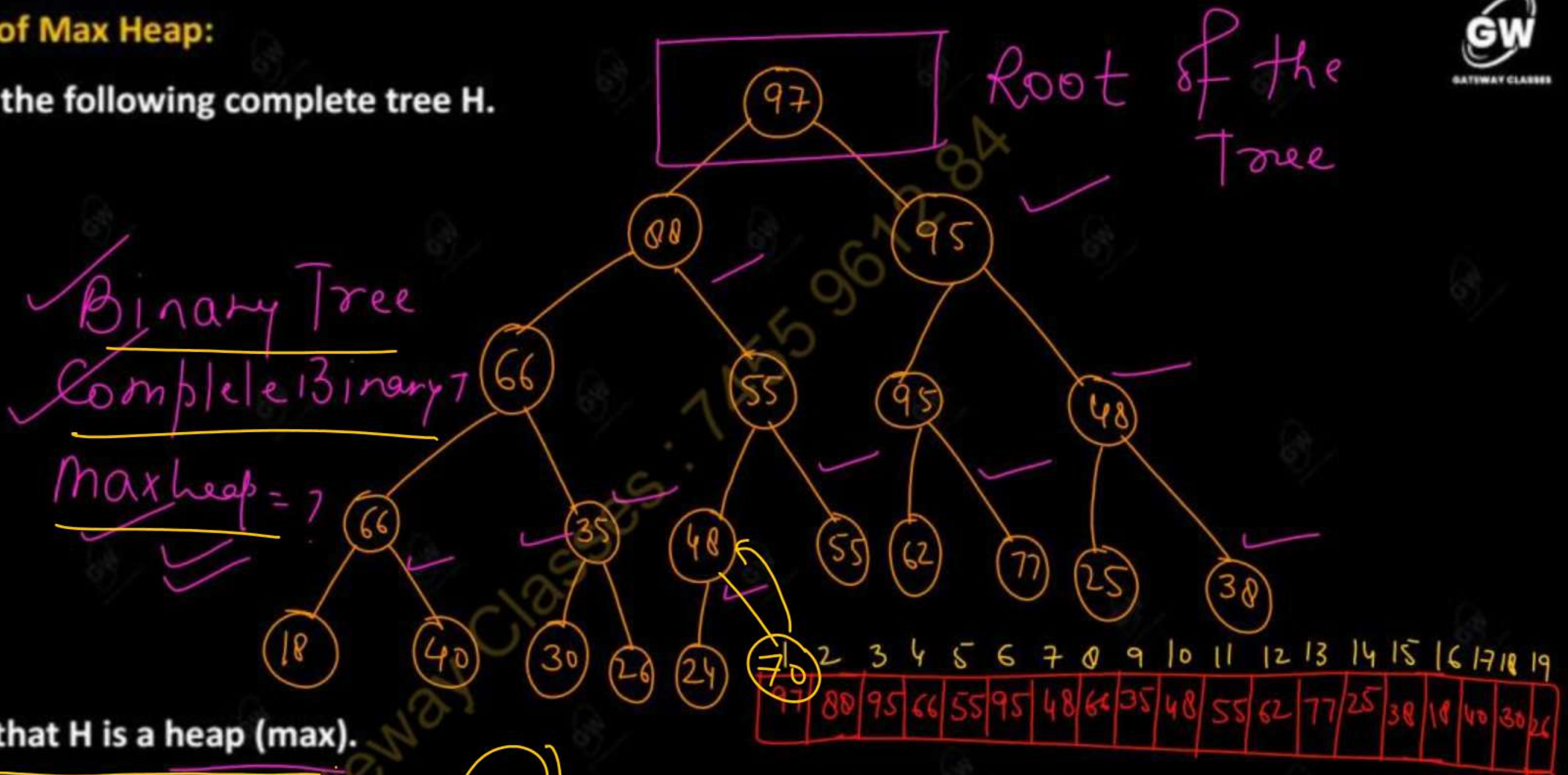






Example of Max Heap:

Consider the following complete tree H.



Observe that H is a heap (max).

This means, that the largest element in H appears at the top of the heap, that is , at the root of the tree.

Inserting into a Heap:-

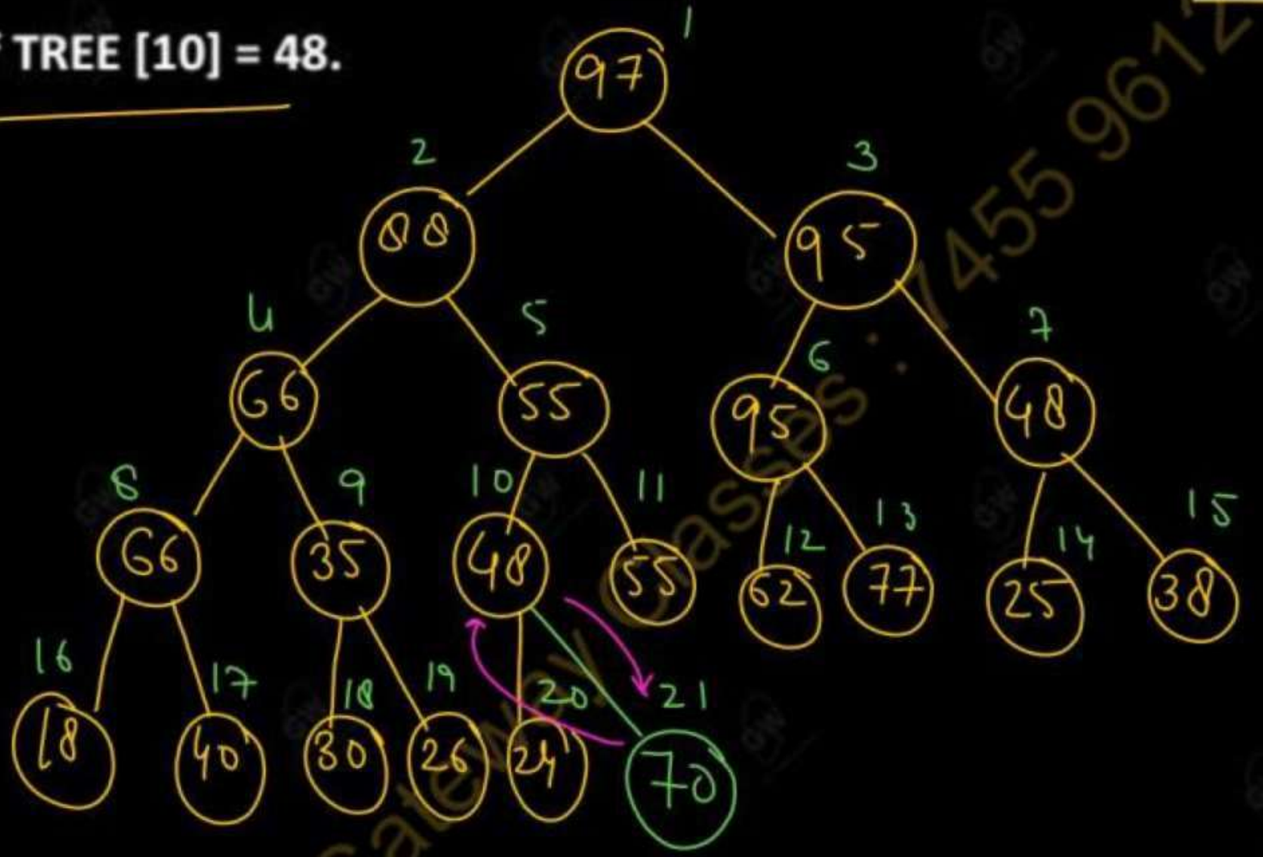
Suppose H is a heap with N elements, and suppose an ITEM of information is given. We insert ITEM onto the heap H as follows:

1. First adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.
2. Then let ITEM rise to its "appropriate place" in H so that H is finally a heap.

Example – Insertion of an ITEM into a heap (max heap)

Consider the heap in the previous tree. Suppose we want to add ITEM = 70 to the heap.

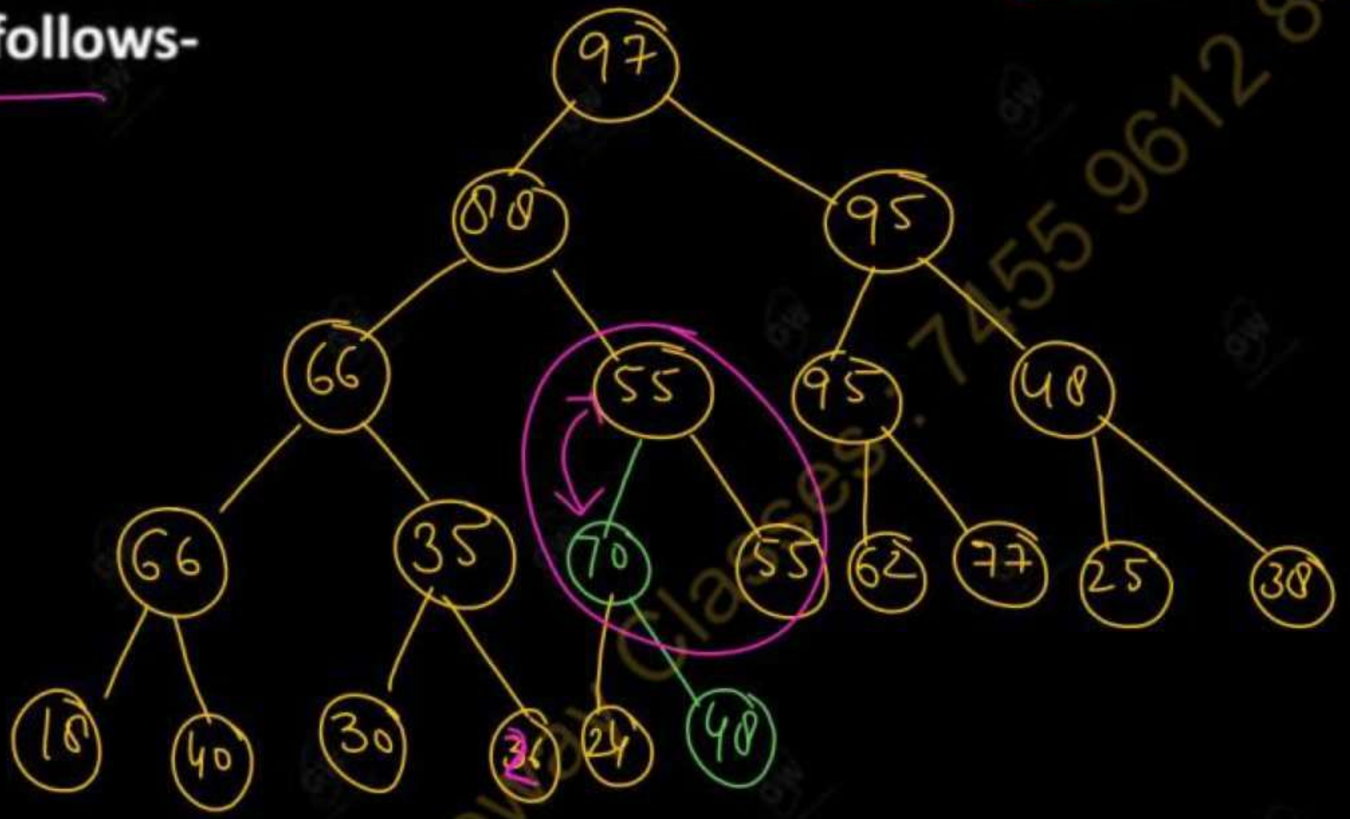
First we adjoin 70 as the next element in the complete tree; that is TREE [21] = 70. Then 70 is the right child of TREE [10] = 48.



This is not Maxheap. (Descending)

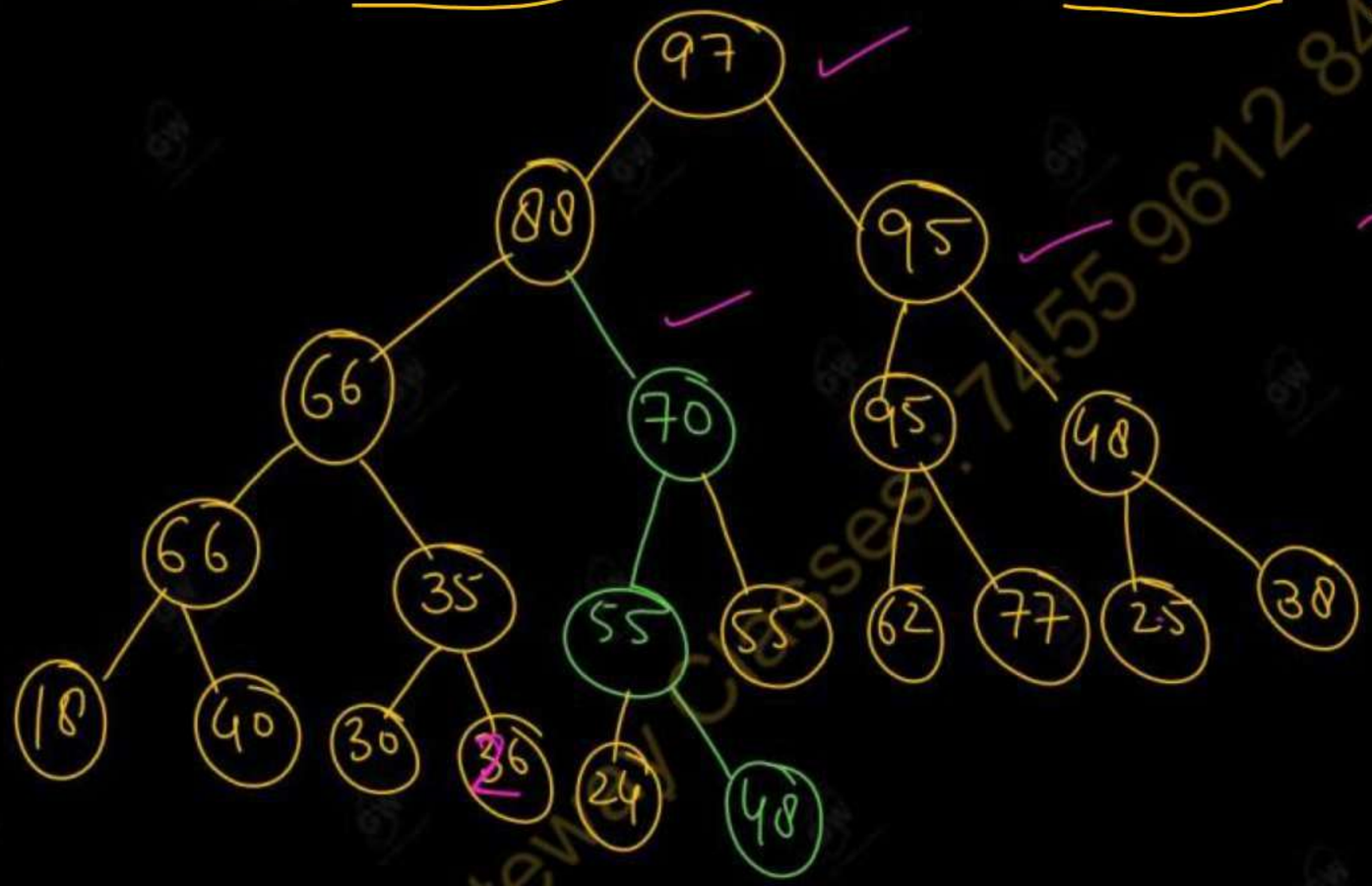
We now find the appropriate place of 70 in the heap as follows:

(a) Compare 70 with its parent, 48. Since 70 is greater than 48, interchange 70 and 48 as follows-



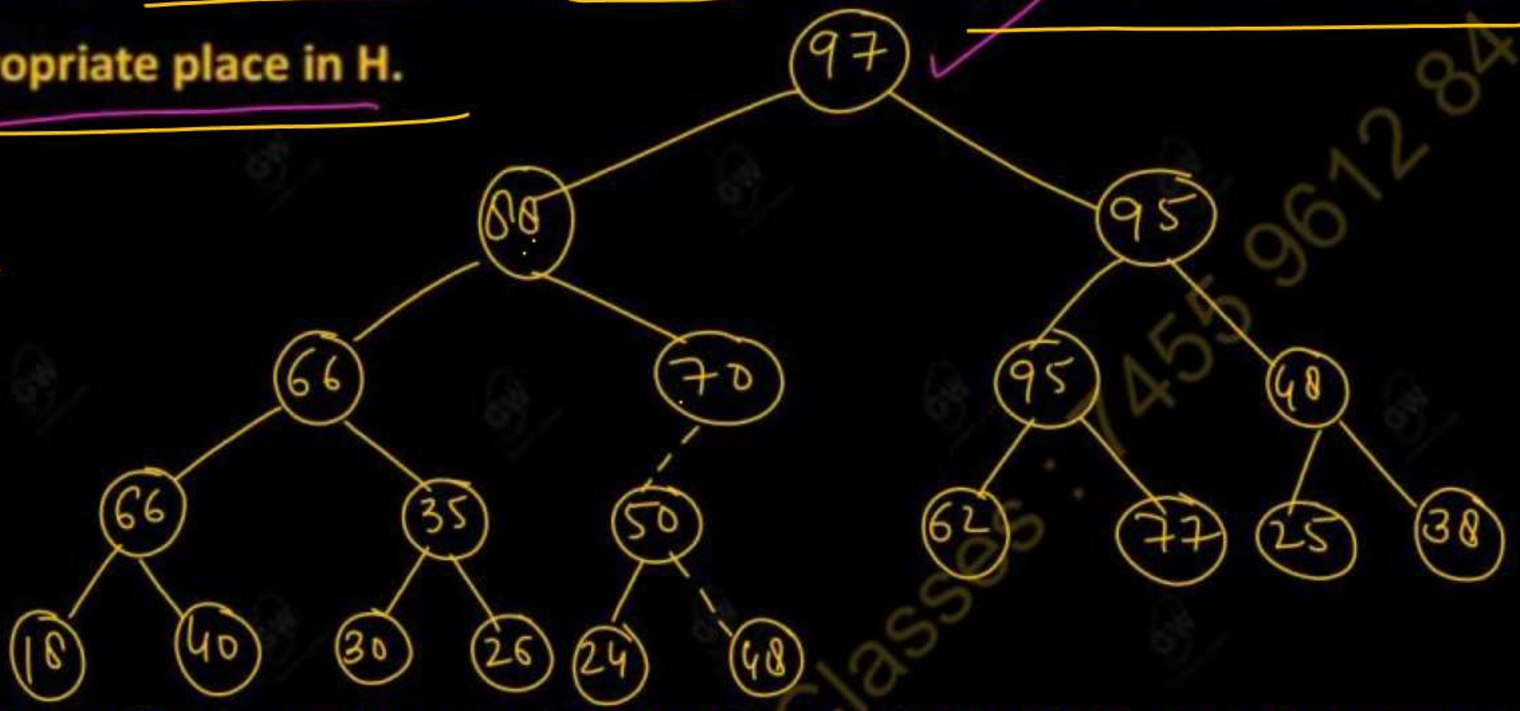
not a maxheap.

(a) Compare 70 with its new parent, 55. sine 70 is greater than 55, interchange 70 and 55 as follows



This is max heap

(c) Compare 70 with its new parent, 88. Since 70 does not exceed 88, ITEM = 70 has risen to its appropriate place in H.



This is the final max-heap and dotted line indicates that an exchange has taken place.

Example : - Suppose we want to build a heap H from the following list of members:

44, 30, 50, 22, 60, 55, 77, 55

Solution:-

item: 44

Root node

Max heap

a

b

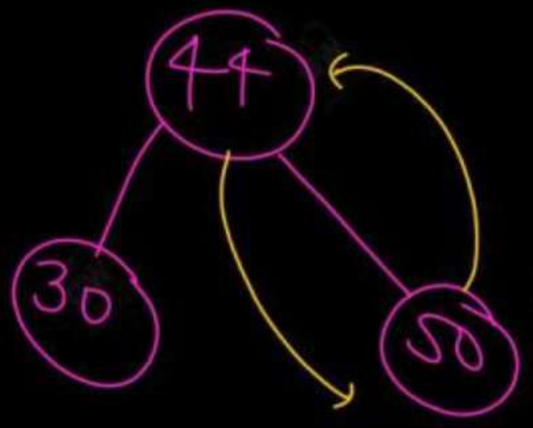
item = 30

CBTree

Maxheap

c

item = 50

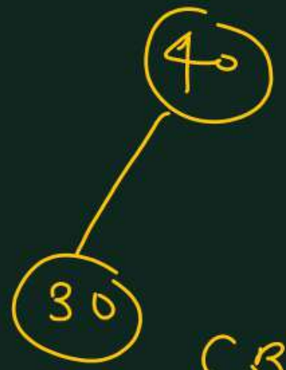


Complete BT

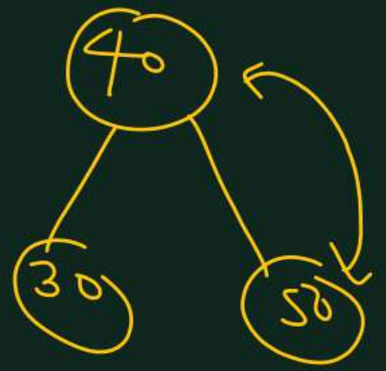
Max heap = 1

44, 30, 50, 22, 60, 55, 77, ~~55~~ 55

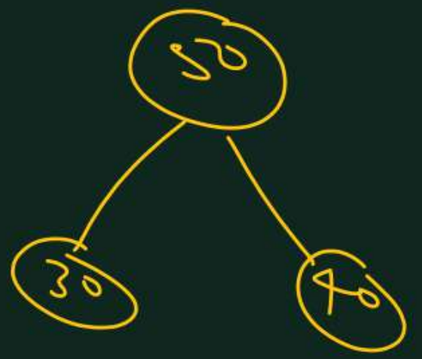
maxheap



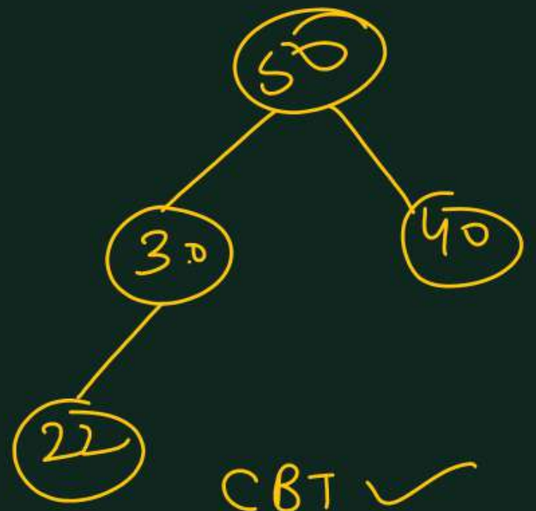
CBT
maxheap



CBT
maxheap = X

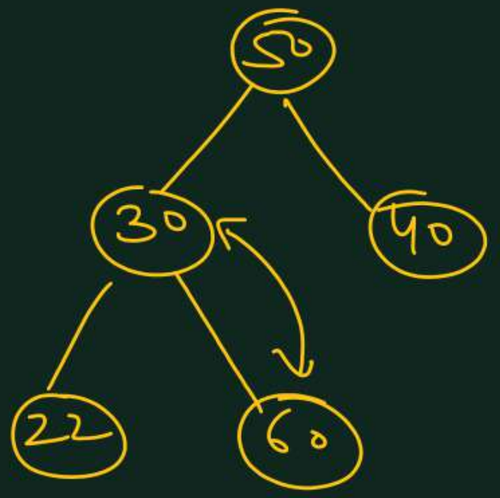


CBT
maxheap.



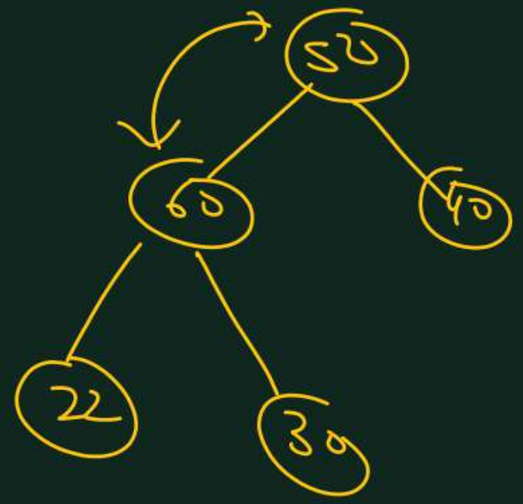
CBT ✓

Maxheap.

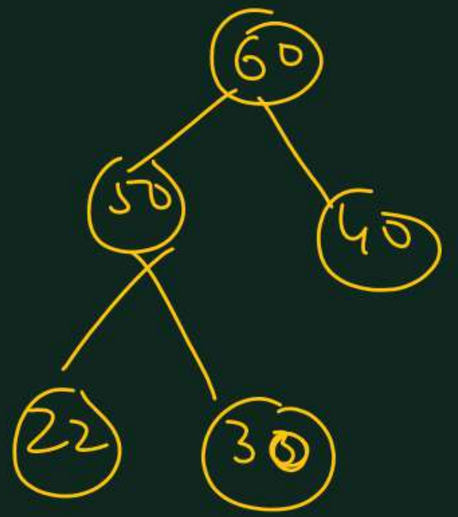


CBT ✓

Maxheap ×

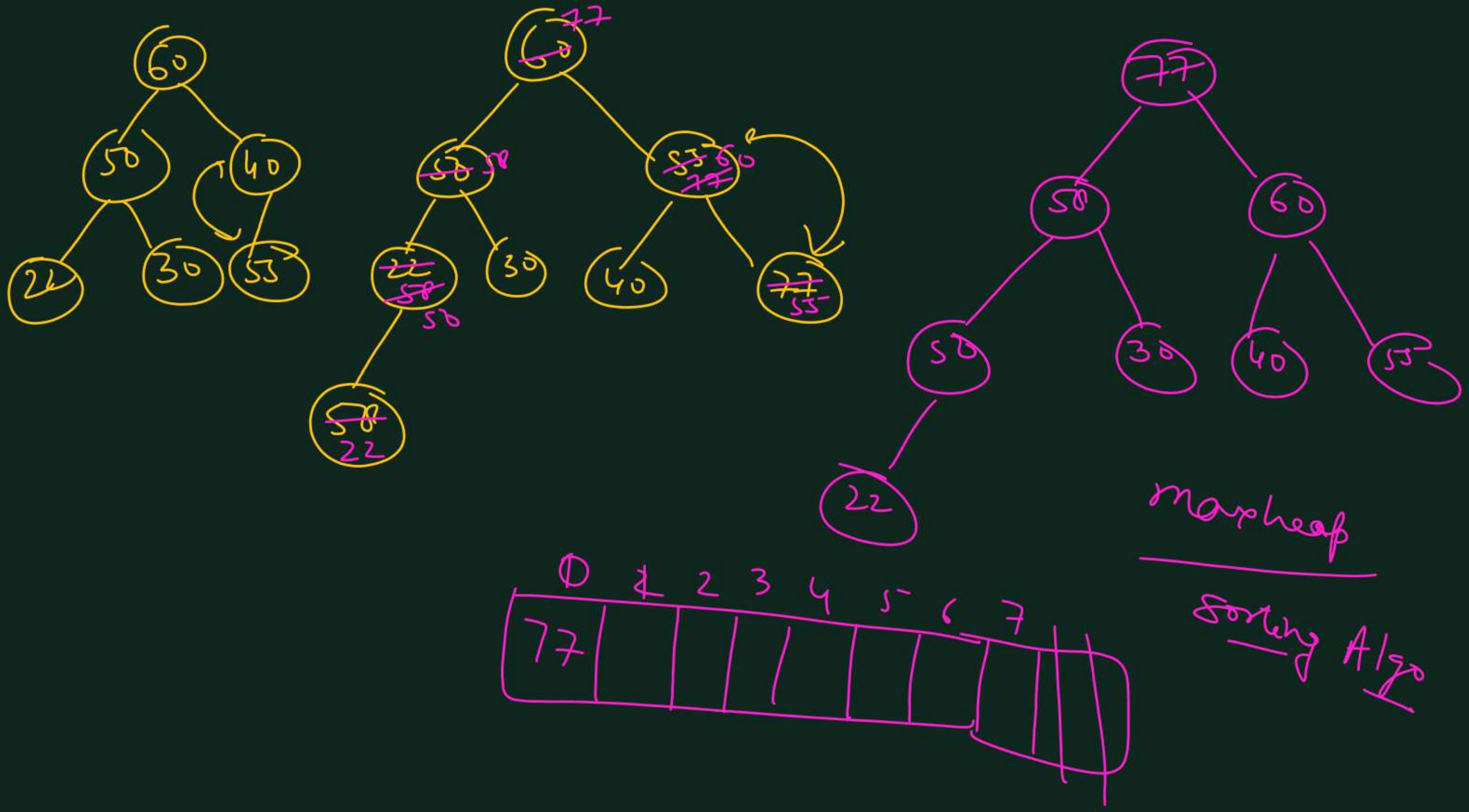


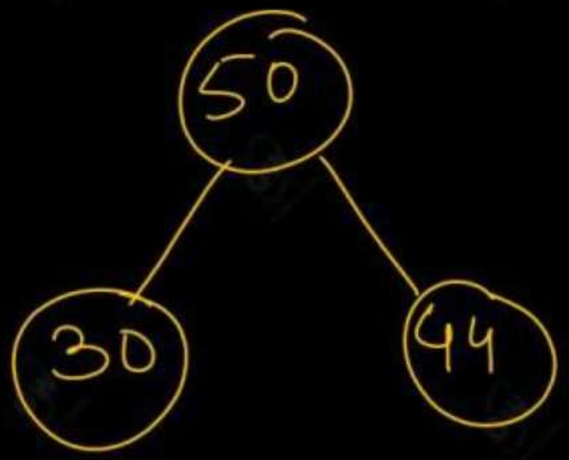
Maxheap ×



CBT ✓

Maxheap

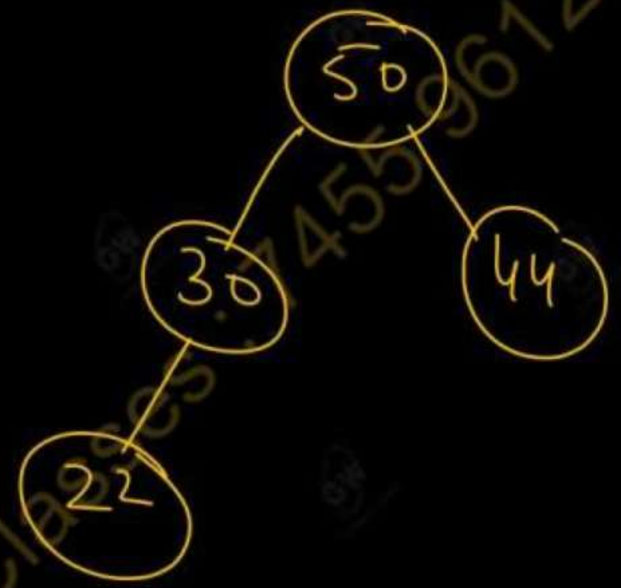

60
50
40
24
30
55
77
50
50
55
60
77
50
30
40
77
55
50
22
50
22
77
50
60
50
30
40
55
22
0 1 2 3 4 5 6 7
77
maxheap
Sorting Algo



CBT

Maxheap.

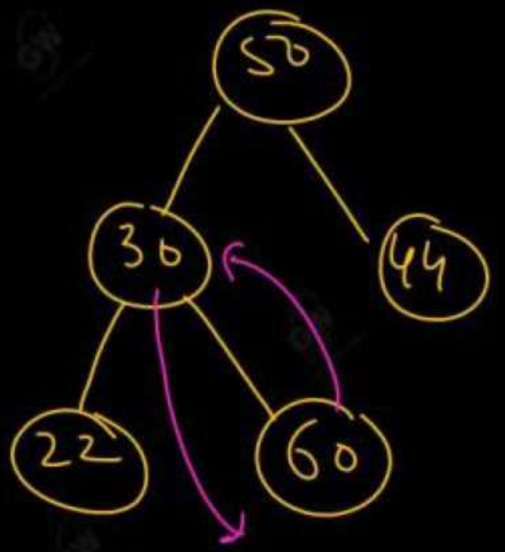
(d) item = 22



CBT ✓

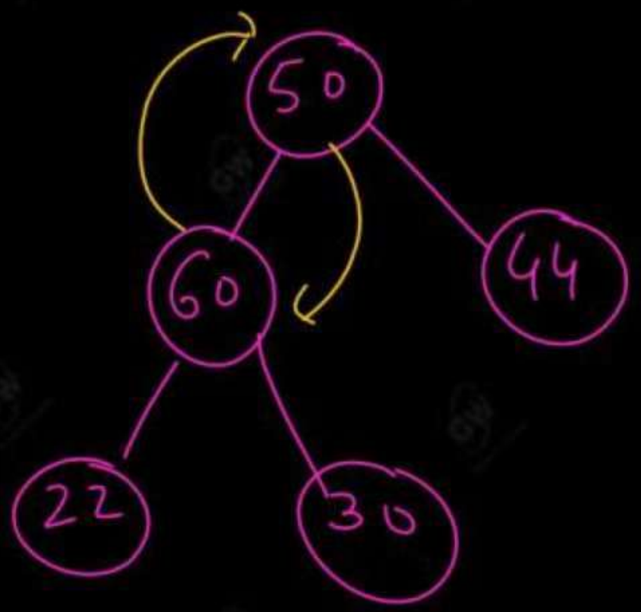
maxheap = ✓

(e) item = 60



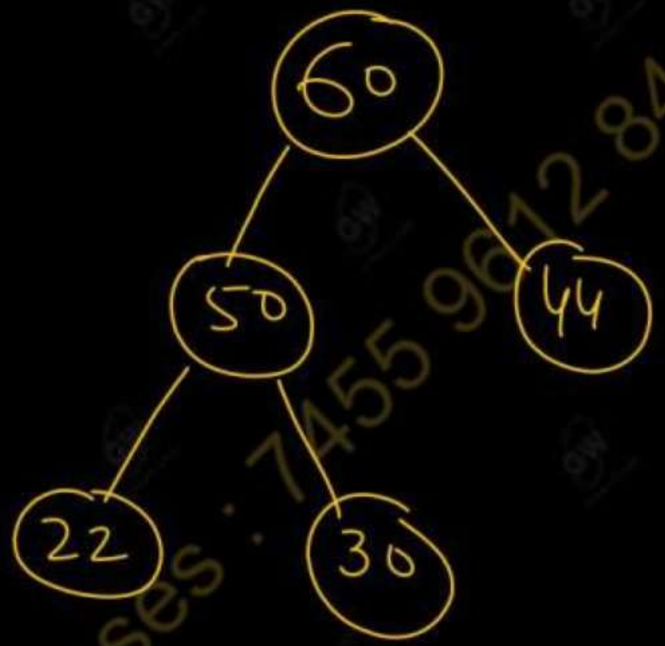
CBT ✓

Maxheap ✗



CBT ✓

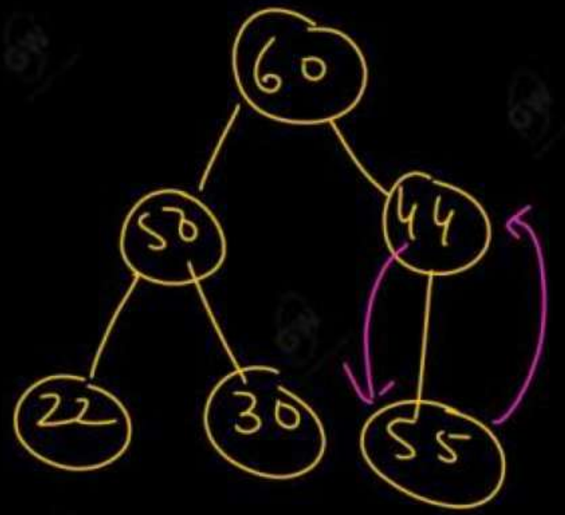
Maxheap = ✗



CBT - ✓

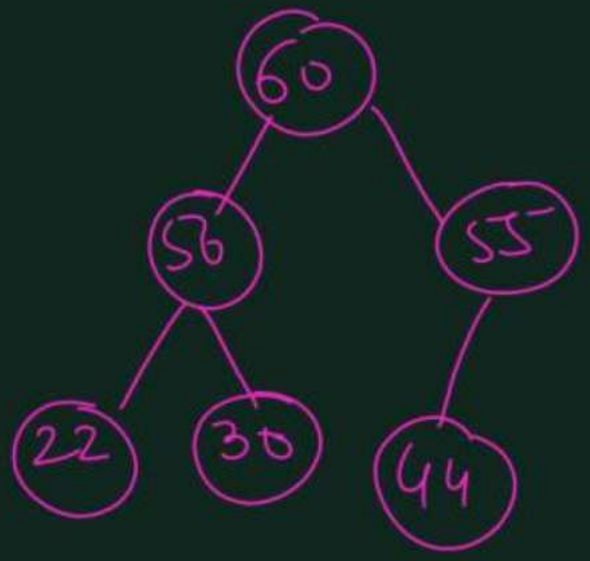
Maxheap = ✓

f item = 55



CBT = ✓

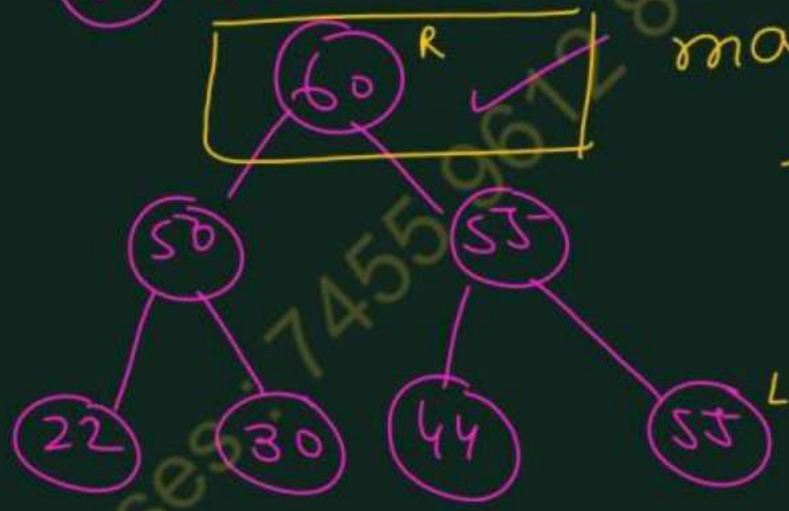
Maxheap = ✗



CBT = ✓

Maxheap = ✓

(g) item = 55

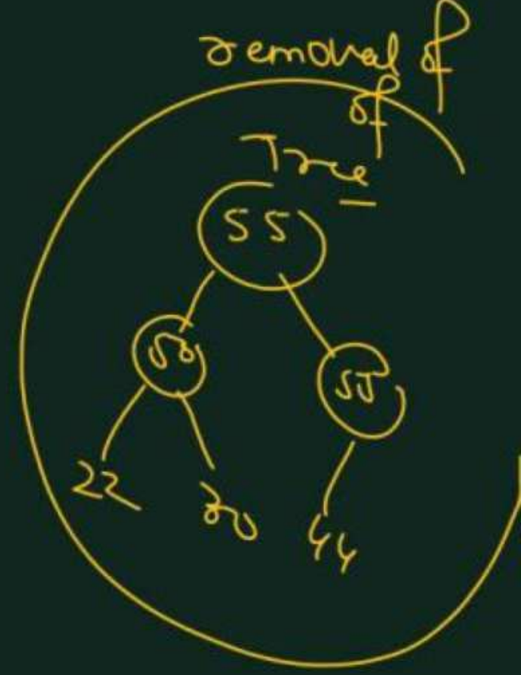


max element of the tree

CBT = ✓ H

Maxheap = ✓

removal of



Deleting the Root of the Heap

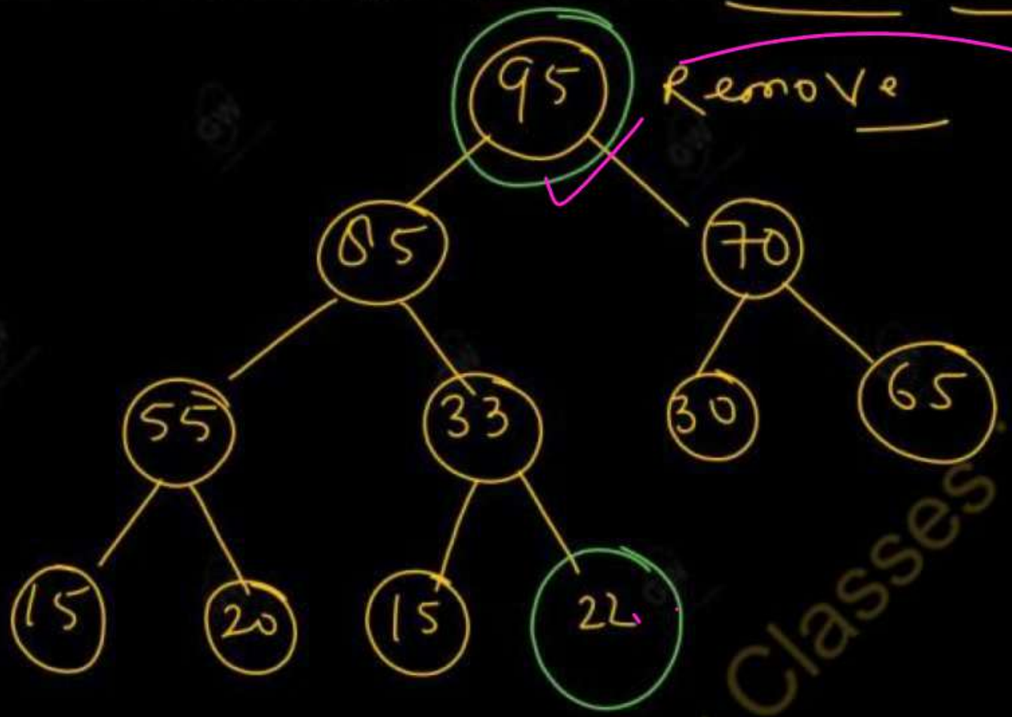
Suppose H is a heap with N elements, and suppose we want to delete the root R of H. This is accomplished as follows:

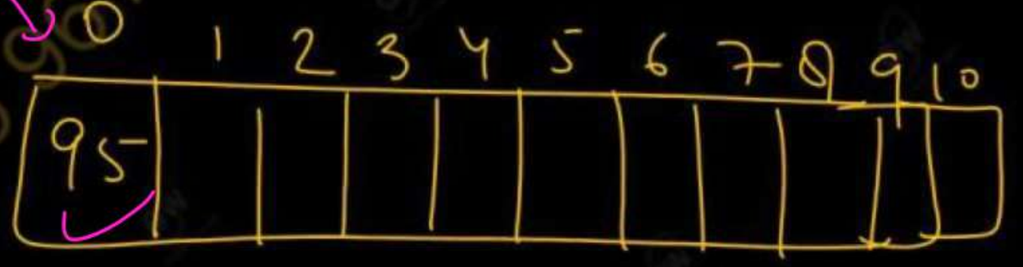
1. Assign the root R to some variable ITEM.
2. Replace the deleted node R by the last node L of H is still a complete tree, but not necessarily a heap.
3. (Reheap) Let L sink to its appropriate place in H so that H is finally a heap.

ITem = 60

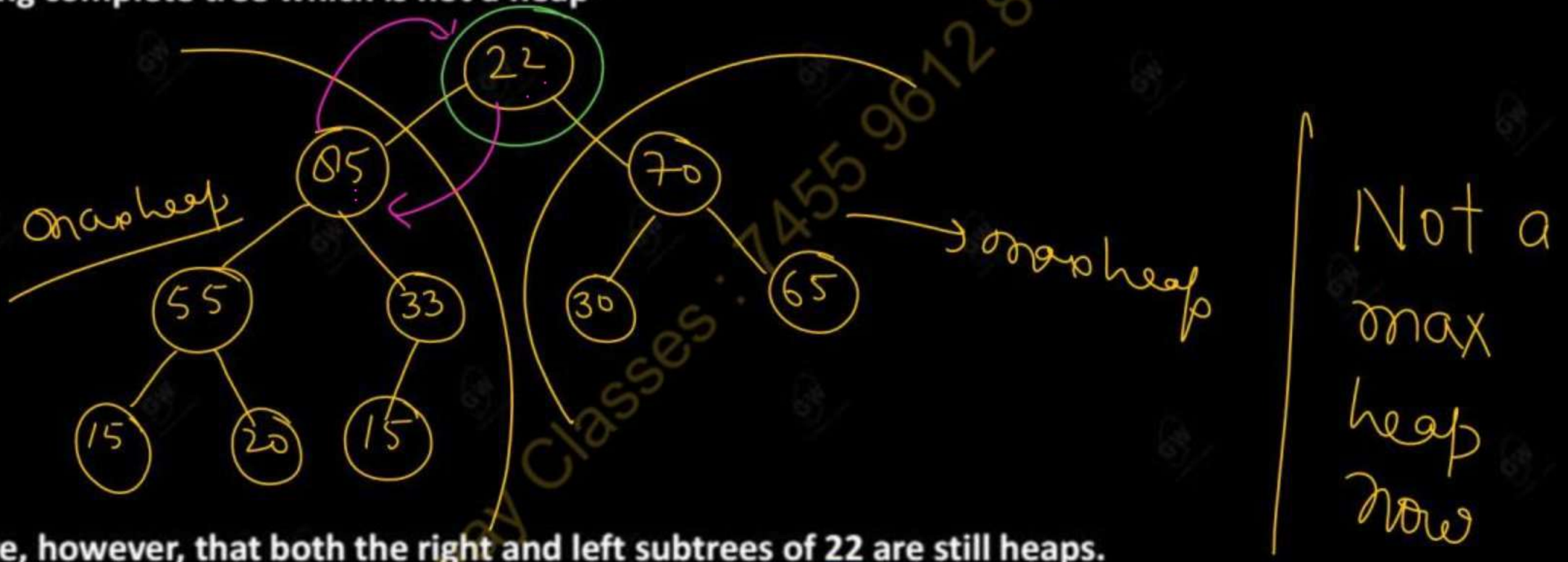
Example:-

Consider the following heap H, where R = 95 is the root and L = 22 is the last node of the tree.





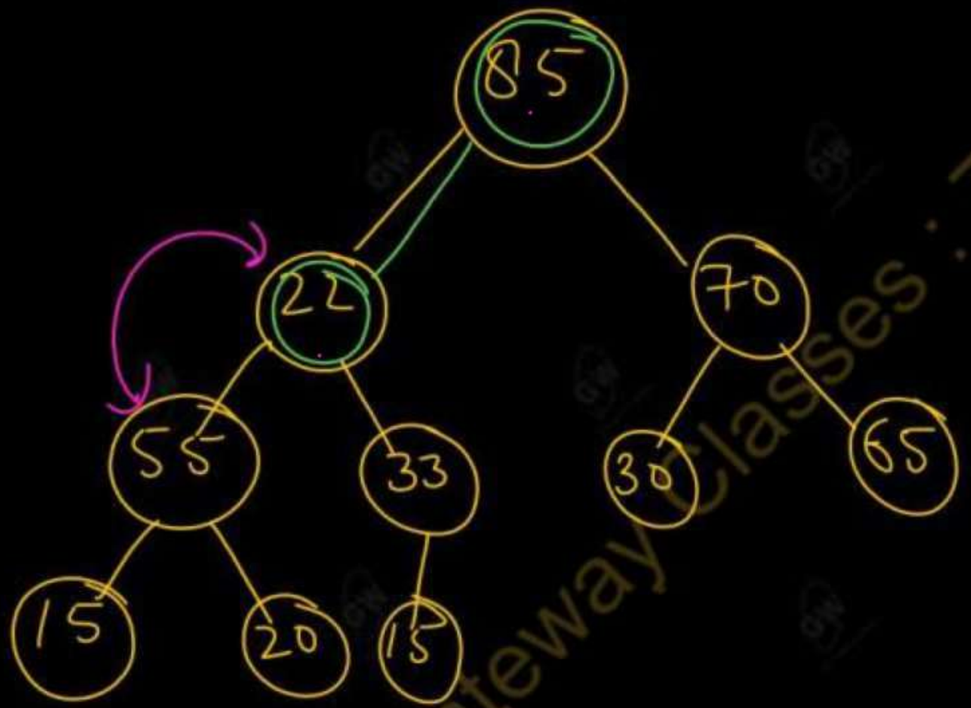
Step 1 of the above procedure deletes R = 95, and step 2 replaces R = 95 by L = 22. This gives the following complete tree which is not a heap –



Observe, however, that both the right and left subtrees of 22 are still heaps.

Applying step 3, we find the appropriate place of 22 in the heap as follows:

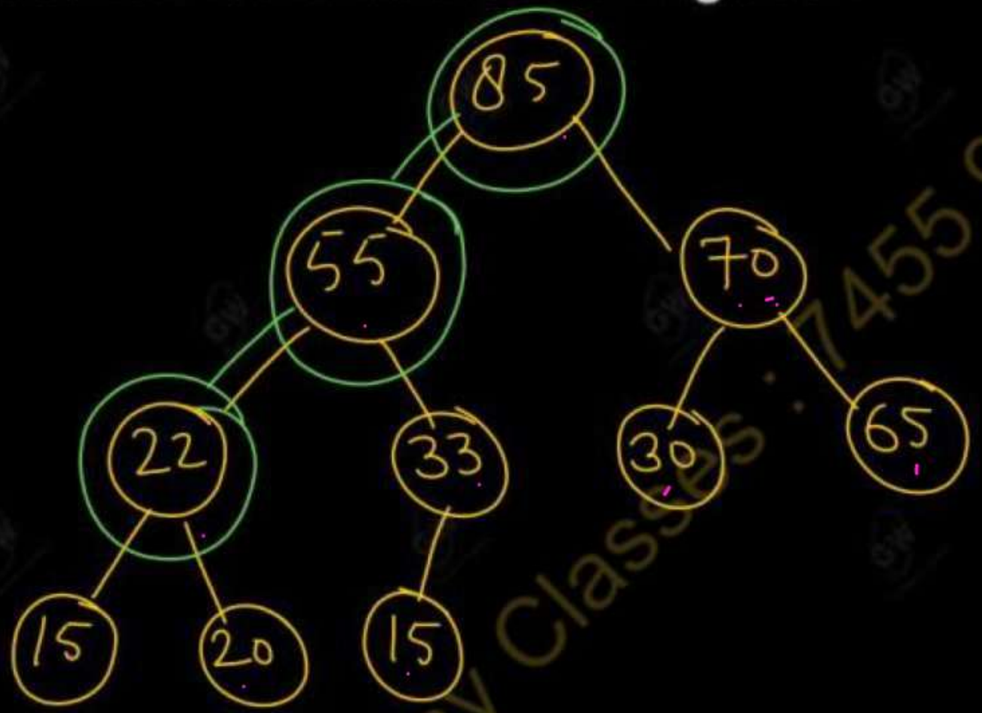
(a) Compare 22 with its two children, 85 and 70. Since 22 is less than the larger child, 85, interchange 22 and 85 so the tree now looks like the following tree –



CBT = ✓

Max heap =

(b) Compare 22 with its new children, 55 and 33. Since 22 is less than the larger child, 55, interchange 22 and 55 so the tree now looks like the following tree –



CBT = ✓

Maxheap = ✓

(c) Compare 22 with its new children, 15 and 20. Since 22 is greater than both children, node 22 has dropped to its appropriate place in H.

Thus previous tree is the required heap H without its original root R.

Algorithm INSHEAP (TREE , N , ITEM)

A heap H with N elements is stored in the array TREE, and an ITEM of information is given. This procedure inserts ITEM as a new element of H. PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the ITEM.

Step 1: [Add new node to H and initialize PTR.]

Set N = N + 1 and PTR = N.

Step 2: [find location to insert ITEM]

Repeat steps 3 to 6 while PTR > 1.

Step 3: Set PAR = [PTR / 2]. [Location of parent node.]

Step 4: If ITEM < = TREE [PAR] , then:

Set TREE [PTR] = ITEM, and Return.

[End of If Structure]

Example –

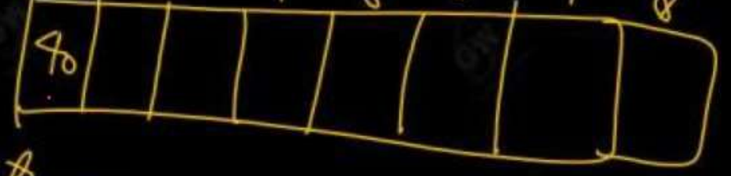
40, 30, 50, 22, 60, 55, 75, 58

item = 40

(1) N = 0 + 1 = 1, PTR = 1

(2) Repeat steps 3 to 6 while 1 > 1 false

(7) Tree[1] = 40



(8) return.

Step 5. Set TREE [PTR] = TREE [PAR]. [Moves node down]

Step 6. Set PTR = PAR [Update PTR]

[End of Step 2 loop]

Step 7. [Assign ITEM as root of H]

Set TREE [1] = ITEM

Step 8. Return

item = 30

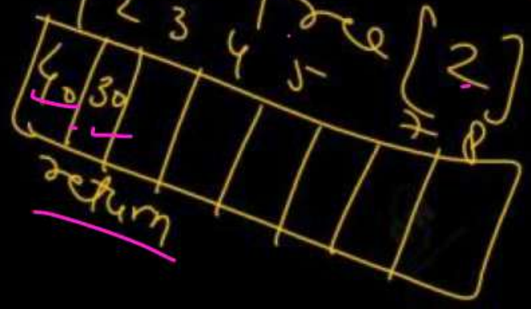
Step ① N = 1+1 = 2 PTR = 2

② 3 to 6 — PTR > 1
2 > 1 True

③ PAR = PTR / 2 = 2 / 2 = 1

④ if item <= Tree[PAR]
30 <= Tree[1]
30 <= 40 True

Tree[PTR] = item
Tree[2] = 30



ITEM = 50

1	2	3	4	5	6	7	0
50	30	40					

① N = 2+1 = 3, PTR = 3

② repeat steps 3 to 6 while PTR > 1
3 > 1
True

③ PAR = [PTR/2] = 3/2 = 1

④ if item <= Tree[PAR]
50 <= Tree[1]
50 <= 40 false

⑤ Tree[PTR] = Tree[PAR]
Tree[3] = Tree[1] = 40

⑥ PTR = PAR = 1

② repeat step 3 to 6
while PTR > 1
1 > 1 false

⑦ set tree[1] = 50

⑧ return

item = 22

1	2	3	4	5	6	7	8
50	30	40	22				

① N = 3+1 = 4, PTR = 4

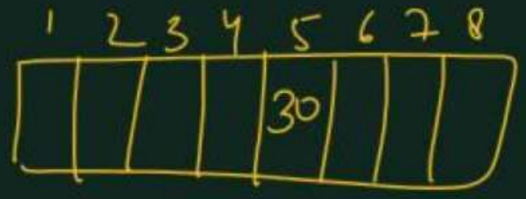
② repeat steps 3 to 6 while PTR > 1
4 > 1 True

③ PAR = PTR / 2 = 4 / 2 = 2

④ if item <= Tree[PAR]
22 <= Tree[2]
22 <= 30 True ✓
Tree[PTR] = item
Tree[4] = 22
return

⑤ Tree[PTR] = Tree[PAR]
Tree[4] = Tree[2]
= 30

item = 60



① N = N + 1 = 4 + 1 = 5 PTR = 5

② repeat steps 3 to 6 while PTR > 1

5 > 1 True

③ PN = PTR/2 = 5/2 = 2

④ if 60 <= Tree[2]

60 <= 30 False

⑤ Tree[PTR] = Tree[PAR]

Tree[5] = Tree[2] = 30

⑥ PTR = PAR = 2

② repeat steps 3 to 6 while PTR > 1

2 > 1 True

③ PAR = PTR/2 = 2/2 = 1

④ if 60 <= Tree[1]

60 <= 50 False

item = 55

① $\Lambda = 6$ PTR = 6

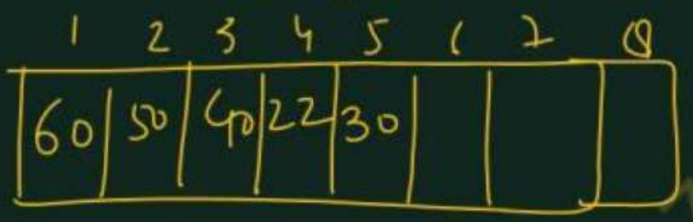
② repeat steps 3 to 6 while PTR > 1
6 > 1 T

③ PAR = PTR/2
= 6/2 = 3

④ if item <= Tree[PAR]
55 <= Tree[3]
55 <= 40 False

⑤ T[PTR] = T[PAR]
T[6] = T[3]
= 40

⑤ Tree[PTR] = Tree[PAR]
Tree[2] = Tree[1]
= 50

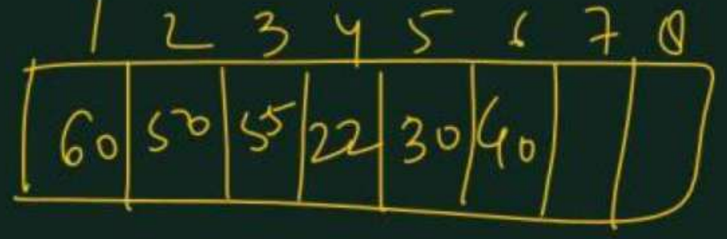


⑥ PTR = PAR = 1

② repeat steps 3 to 6 while PTR > 1
1 > 1 False

⑦ Tree[1] = 60

⑧ return



⑥ PTR = PAR = 3

② repeat steps 3 to 6 while PTR > 1

3 > 1 True

③ PAR = PTR / 2 = 3 / 2 = 1

④ if item <= T[PAR]

55 <= T[1]

55 <= 60 T

T[PTR] = item

T[3] = 55

⑤ T[PTR] = T[PAR]

T[3] = T[1]

= 60

return

item = 75

① N = 7 PTR = 7

② repeat step 3 to 6 while PTR > 1

7 > 1 T

③ PAR = PTR | 2 = 7 | 2
= 3

④ if item <= T[PAR]
75 <= T[3]
75 <= 55 F

⑤ T[PTR] = T[PAR]
T[7] = T[3]
= 55

1	2	3	4	5	6	7	8
75	50	60	22	30	40	55	

⑥ PTR = PAR
= 3

② repeat step 3 to 6 while
PTR > 1
3 > 1 T

③ PAR = PTR | 2
= 3 | 2 = 1

④ if item <= T[PAR]
75 <= T[1]
75 <= 60 F

⑤ T[PTR] = T[PAR]
T[3] = T[1]
= 60

(6) PTR = 1 AR = 1

(2) repeat steps 3 to 6 while PTR > 1
1 > 1 False

(7) Tree[1] = 75

(8) return

---

item = 58

(1) N = N + 1 = 7 + 1 = 8, PTR = 8

(2) repeat 3 to 6 while 8 > 1

(3) PAR = PTR / 2
= 8/2 = 4

(4) if 58 <= T[PAR]
58 <= T[4]
58 <= 22 False

(5) T[PTR] = T[PAR]
T[8] = T[4]
T[8] = 22

1	2	3	4	5	6	7	8
75	58	60	50	30	40	55	22

(6) PTR = PAR = 4

(2) repeat steps 3 to 6 while PTR > 1
4 > 1 T

(3) PAR = PTR / 2 = 4 / 2 = 2

(4) if SD <= T[2]
SD <= SQR

(5) Tree[PTR] = T[PAR]
T[4] = T[2]
= SD

(6) PTR = PAR
= 2

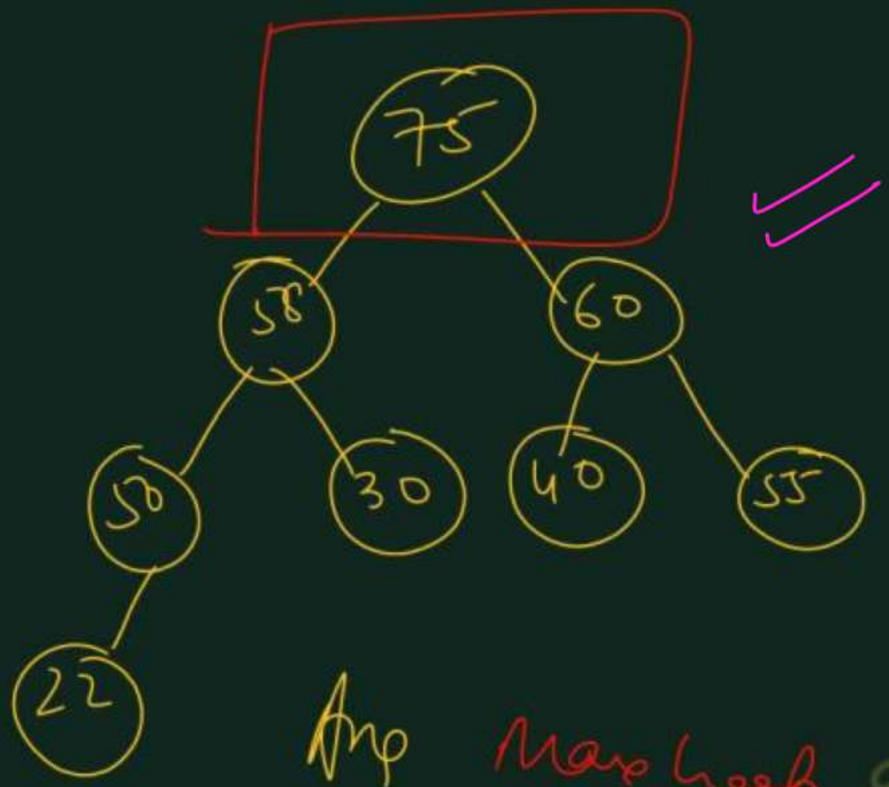
(2) repeat steps 3 to 6
while PTR > 1
2 > 1 T

(3) PAR = PTR / 2 = 2 / 2 = 1

(4) if SD <= T[PAR]
SD <= T[1]
SD <- 75 T

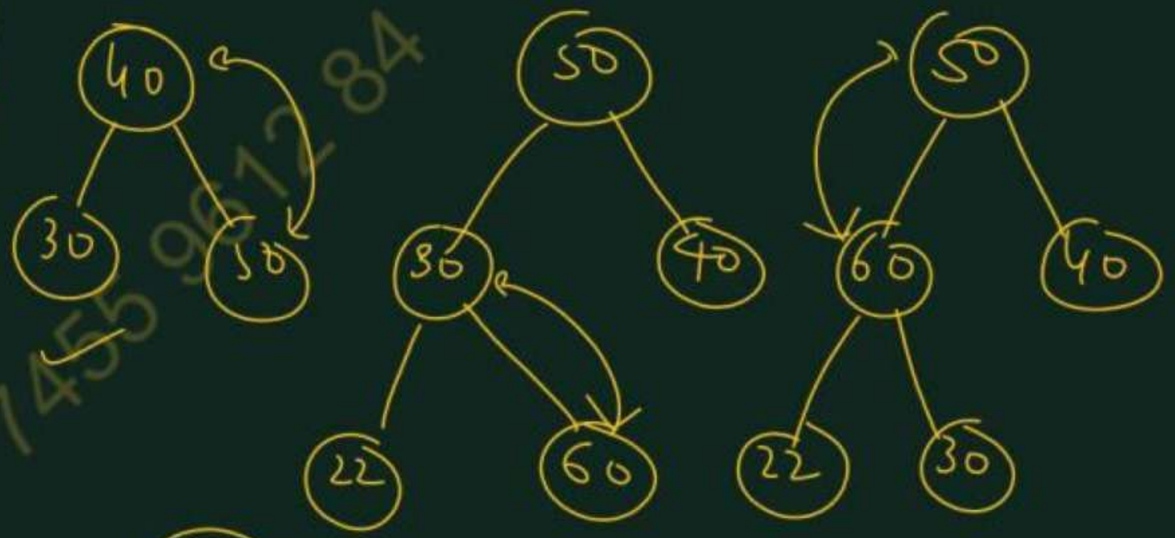
Set T[PTR] = item
T[2] = SD

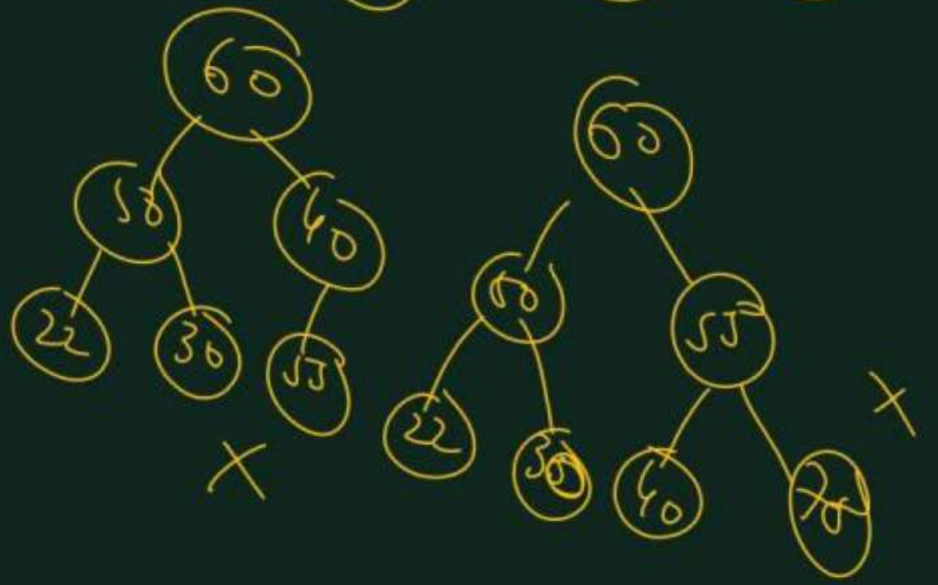
return

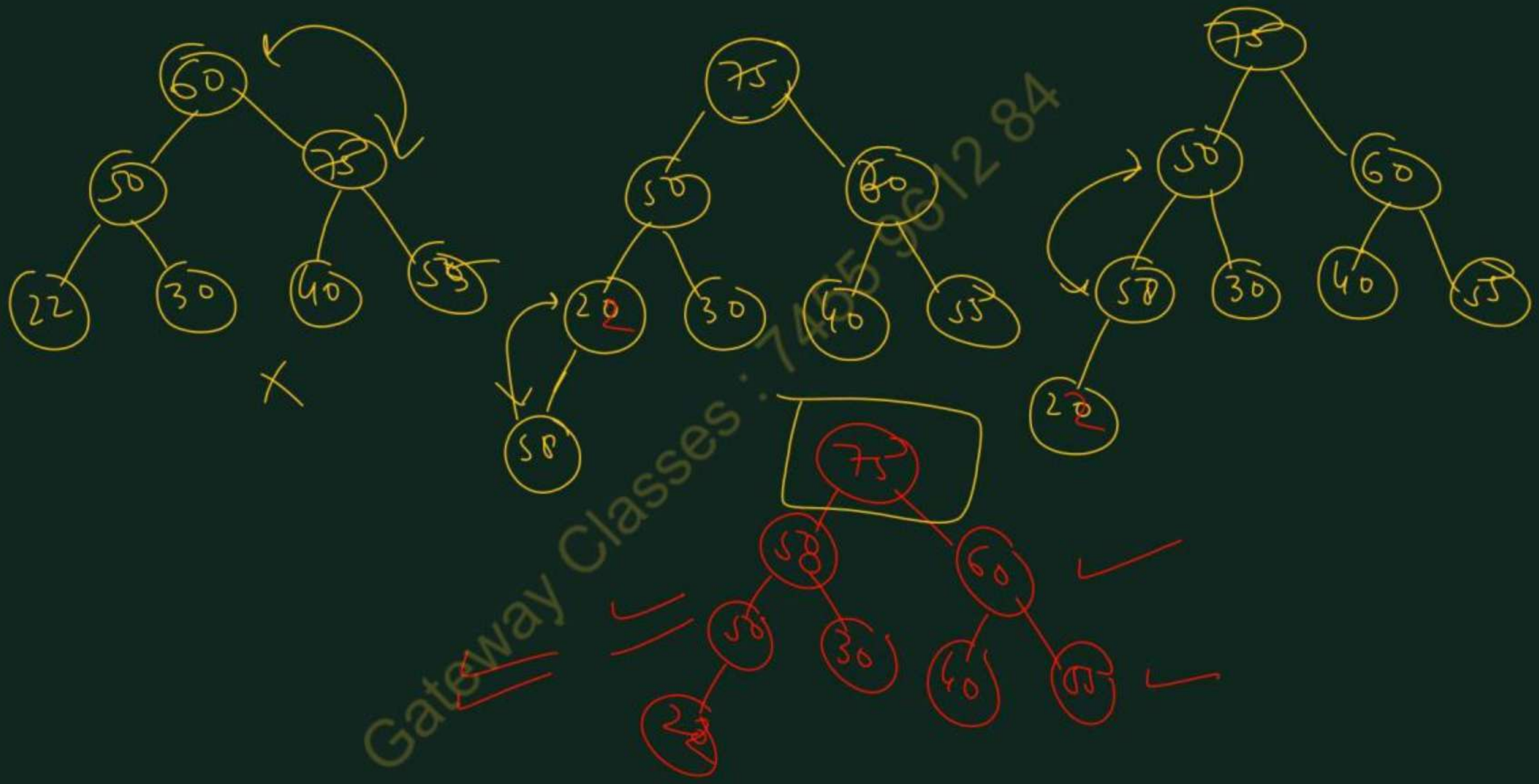


Ans Max heap

40, 30, 50, 22, 60, 55, 75, 58






60
50
75
22
30
40
55
X
75
50
60
20
30
40
55
50
75
50
60
20
30
40
55
20
Gateway Classes : 7405 96 12 84

Algorithm Heap Sort

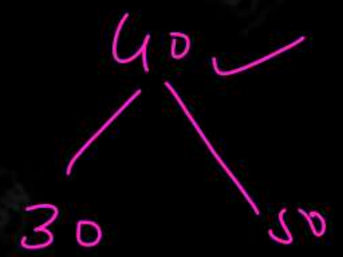
Step 1 : First build the heap (max heap, if desired order is descending). Insertheap (max/min heap)

Step 2 : Remove the root node one by one and place in an empty array and heapify the tree in the same manner as done in step 1. Delheap (Remove 1 node)

Step 3 : The resultant array is desired sorted array in the descending order.

Two Methods of Creating Heap -

To implement Heap sort we should have idea about how to create heap (max heap or min heap) and how to delete item from heap.



Consider the following elements in an order.

There are two methods to create a heap –

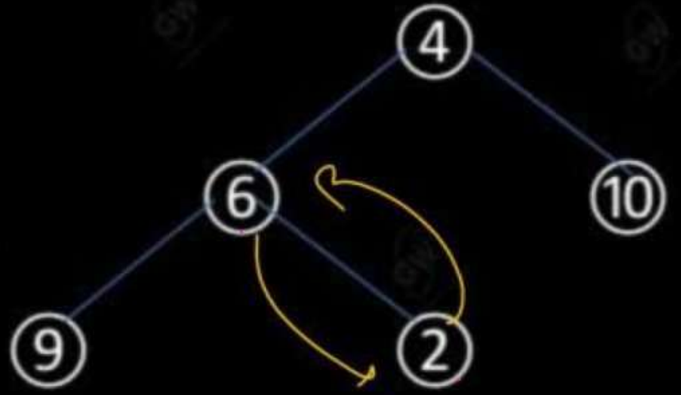
(1) One by one key insertion method and

(2) Heapify method

Let us create a heap by heapify method for the following numbers

4, 6, 10, 9, 2

Just create a complete binary tree in this manner

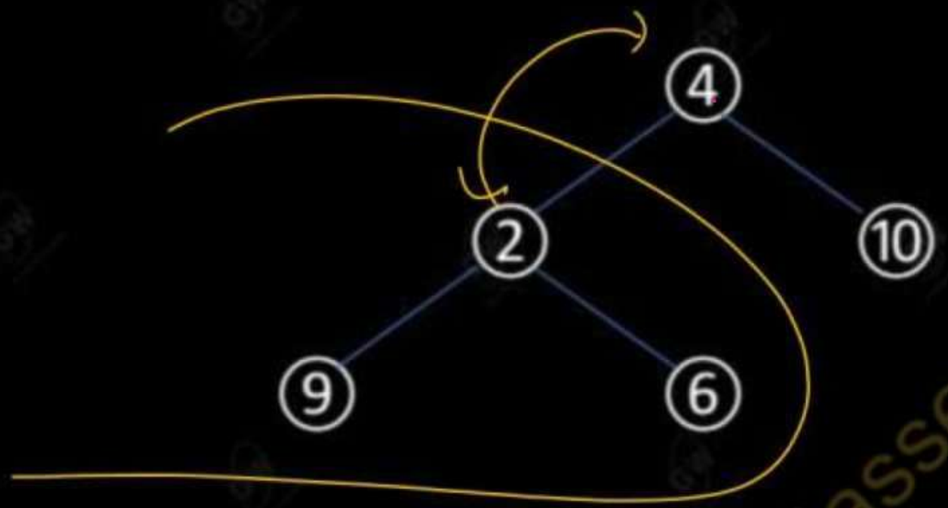


Complete BT

Creating a min heap will give ascending order of sorted elements.

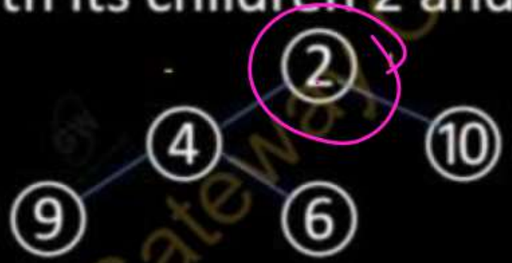
Let us implement the concept of heapify, no need to touch element at leaf nodes (last nodes)

Give right position to 6 , Compare 6 with its children i.e. with 9 and 2 and replace 6 and 2 to obtain the following *tree*.



It is not min heap

Now Compare 4 with its children 2 and 10 and replace 4 and 2 to obtain the following tree



This is now min heap because parents node value is lesser then its children value.

NOTE - Complexity of one by one key insertion method to create a heap is O (n log n) and the complexity of heapify method is O(n).

Now start deleting element from heap i.e. delete 2 i.e. root node of the tree and place this element in an empty array.

If we have removed root of the tree 2, we have to insert the last element of the tree at root to obtain the following free-

Which is not a min heap.

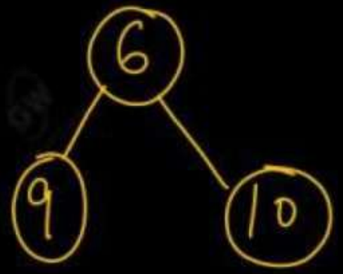
Compare 6 and 4 i.e. root value is greater than its children, now interchange 6 and 4 to obtain the following tree-

Now this is min heap. Now remove root node 4 and place this value in the array

2	4			

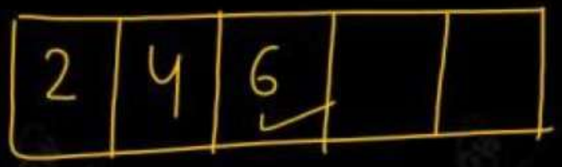
Now insert the last element i.e.9 at root.

This is not min heap because root node value is greater then child node value i.e. implement heapify method *and interchange 6 and 9 to obtain the following tree-*

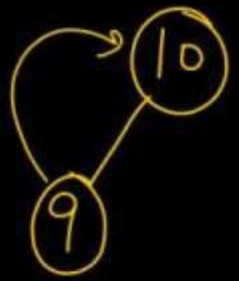


This is min heap.

Now remove root node 6 and place in array.



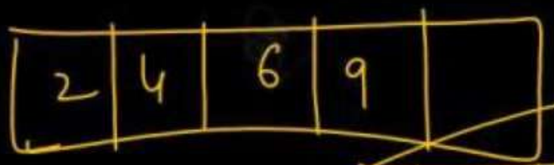
Now, last node will be the root node to obtain the following tree:



This is not min heap.

Interchange 10 and 9 (heapify).

remove root node 9 and place in array.



Now there is only one element which is itself sorted and place in array.

This is the sorted list of elements.

Algorithm DELHEAP (TREE , N , ITEM)

A heap H with N elements is stored in the array TREE. This procedure assigns the root TREE [1] of H to the variable ITEM and then reheaps the remaining elements. The variable LAST saves the value of the original last node of H. The pointers PRT, LEFT and RIGHT gives the locations of LAST and its left and right children as LAST sinks in the tree.

Step 1: Set ITEM = TREE [1]. [Removes root of H.]

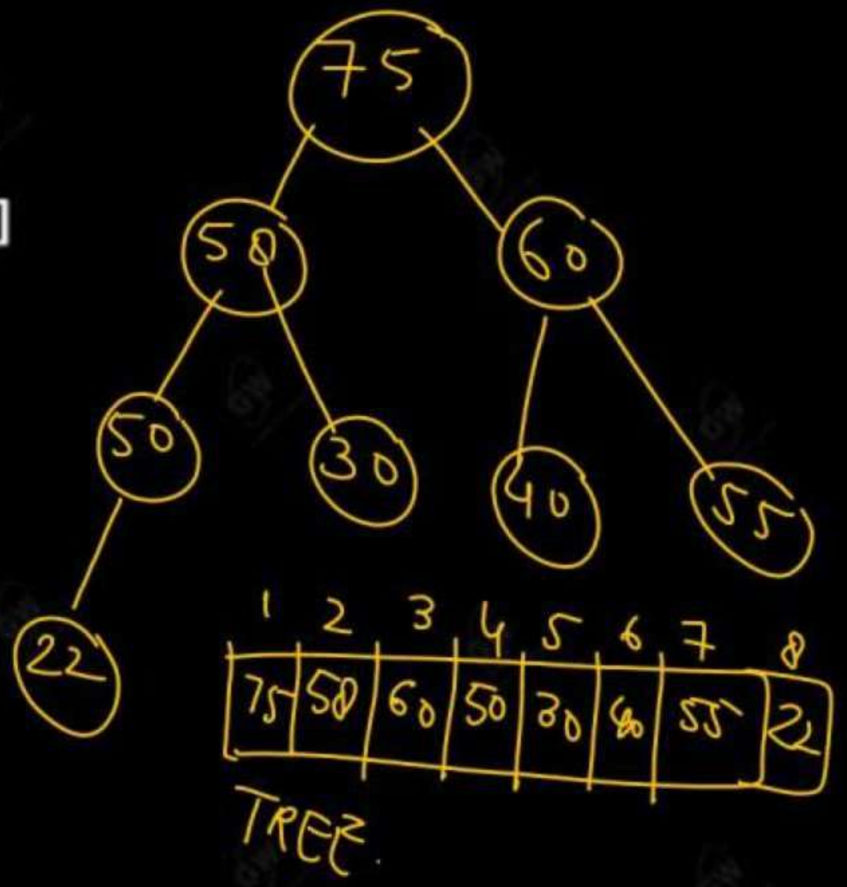
Step 2: Set LAST = TREE [N] and N = N – 1. [Remove last node of H]

Step 3: Set PTR = 1, LEFT = 2 and RIGHT = 3. [Initialize pointers.]

Step 4: Repeat Steps 5 to 7 while RIGHT < = N:

Step 5: If LAST > = TREE [LEFT] and LAST > = TREE [RIGHT], then:

Set TREE [PTR] = LAST and Return. [End of If Structure].



Step 6: If TREE [RIGHT] < = TREE [LEFT], then:

Set TREE [PTR] = TREE [LEFT] and PTR = LEFT

Else:

Set TREE [PTR] = TREE [RIGHT] and PTR = RIGHT.

[End of If Structure]

Step 7: Set LEFT = 2 * PTR and RIGHT = LEFT + 1.

[End of Step 4 loop.]

Step 8: If LEFT = N and if LAST < TREE [LEFT], then : Set PRT = LEFT.

Step 9: Set TREE [PTR] = LAST

Step 10. Return.

① ITEM = TREE[1] = 75 , N=8

② Last = Tree[N] = Tree[8] = 22
N = 7

③ PTR = 1.
left = 2, Right = 3

④ repeat steps 5 to 7 while Right <= N
3 <= 7
True

⑤ if last >= Tree[left] and

22 >= Tree[2]

22 >= 50

False

⑥ if Tree[Right] <= Tree[left]

Tree[3] <= Tree[2]

60 <= 58 false

else

Tree[PTR] = Tree[right]

Tree[1] = Tree[3]

1	2	3	4	5	6	7
60	58	55	56	30	40	22

PTR = Right = 3

⑦ left = 2 × ptr

= 2 × 3

= 6

Right = left + 1 = ~~2+1=3~~ 6+1 = 7

④ Repeat steps 5 to 7

While right <= N

7 <= 7

True

⑤ if last >= Tree(left) and

22 >= Tree[6]

22 >= 40 false

⑥ if Tree[Right] <= Tree[left]

Tree[7] <= Tree[6]

55 <= 40 false

else

Tree[PTR] = Tree[Right]

Tree[3] = Tree[7]

= 55

PTR = right = 7

(7) left = 2 × ptr

= 2 × 7 = 14

right = left + 1 = 14 + 1

= 15

(4) repeat steps 5 to 7 while right <= N

15 <= 7 False

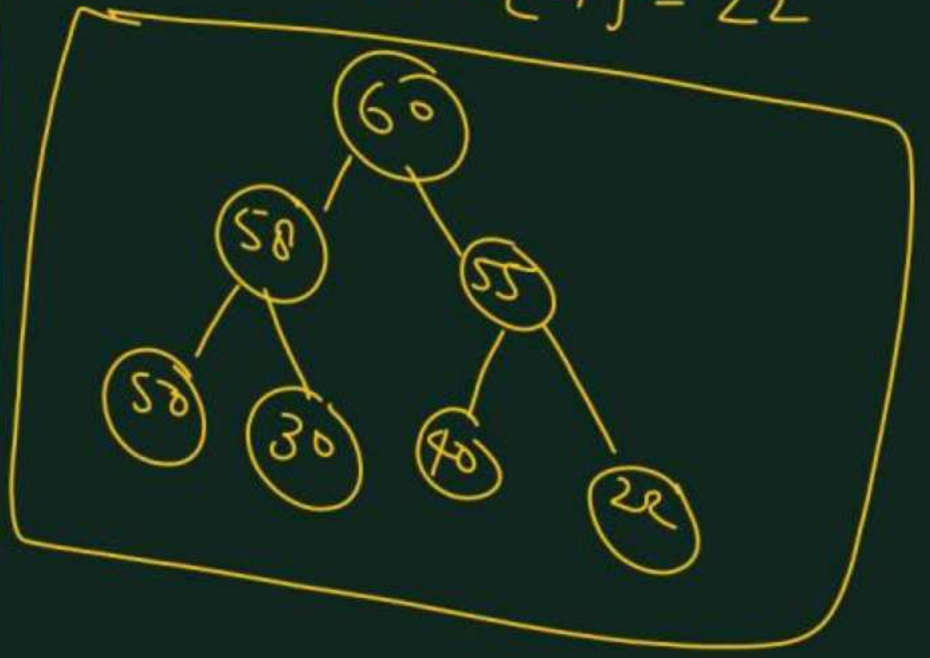
(8) if left = N

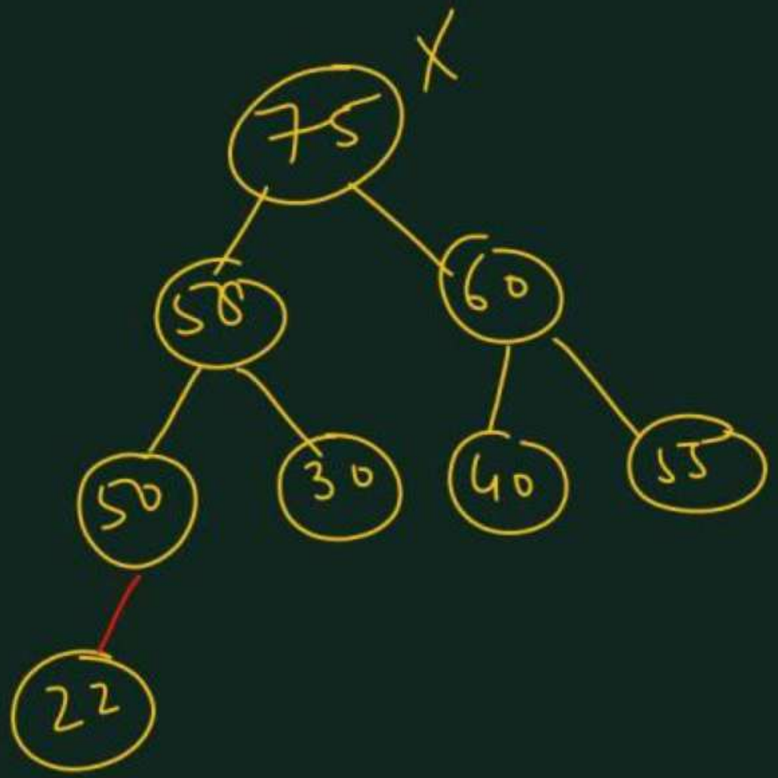
14 = 7

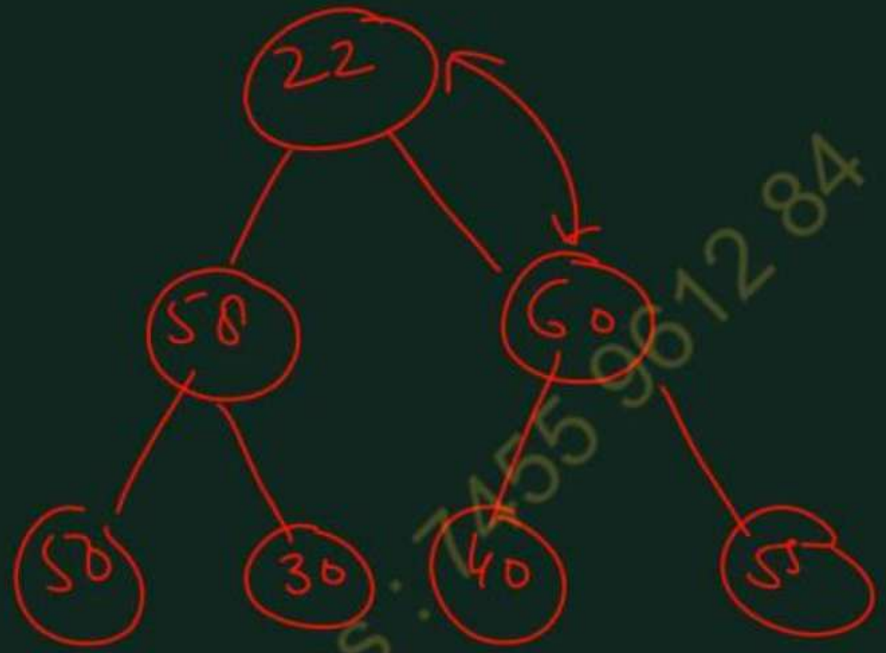
false

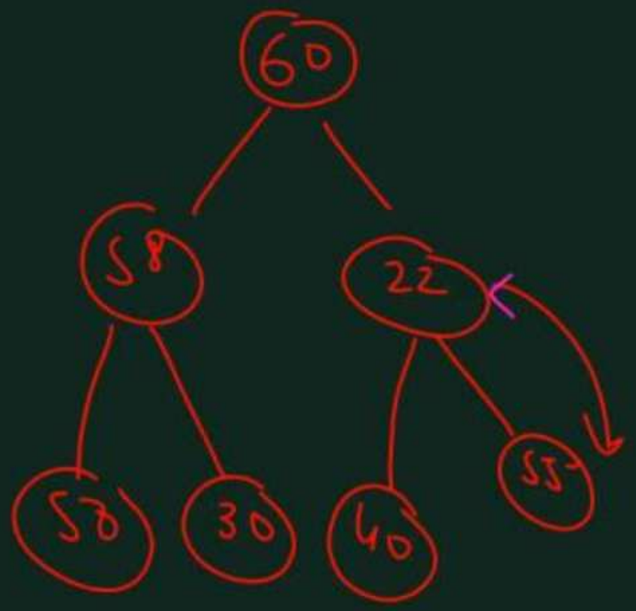
(9) Set Tree[PTR] = last

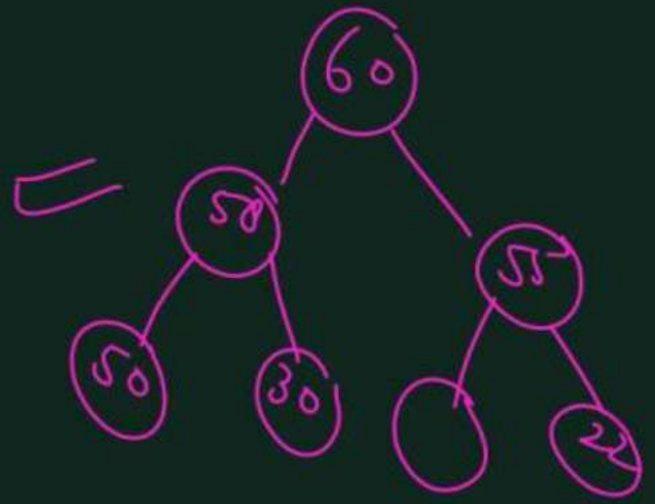
Tree[7] = 22











ALGORITHM : HEAPSORT (A, N)

An array A with N elements is given. This algorithm Sorts the element of A.

STEP 1 : [Build a heap H using INSHEAP]

Repeat for J = 1 to N-1 :

Call INSHEAP (A, J, A[J+1])

[END OF LOOP]

STEP 2: [Sort A by repeatedly deleting the root of H, using procedure DELHEAP].

Repeat while N >= 1:

(a) Call DELHEAP (A, N, ITEM)

(b) Set A[N+1] := ITEM.

[END OF LOOP]

STEP 3 : EXIT.

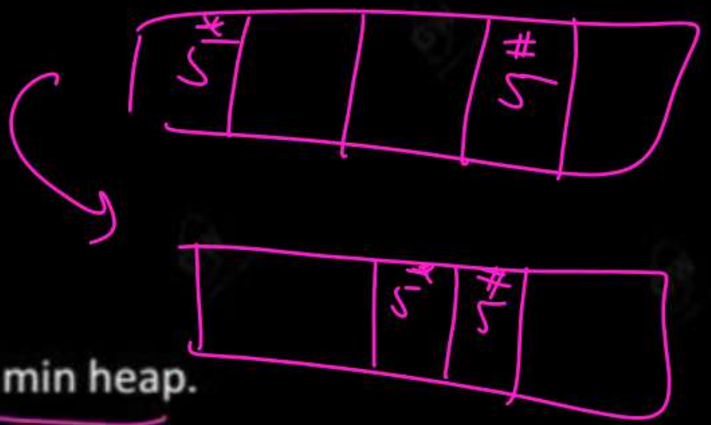
- **Heap sort is unstable because it does not preserve the relative ordering of duplicate elements. This can be understand by the following example.**

Consider following example

Complete Binary Tree

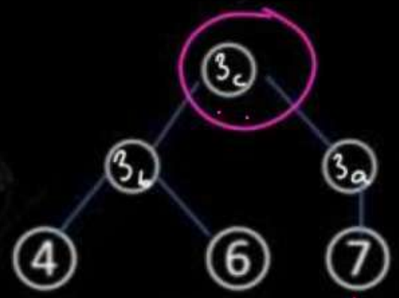
This is not min heap

Compare 6 and 3_b and interchange



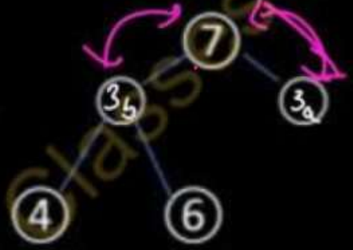
This is min heap.

Now remove root element and place in empty array and place the last element as root to obtain the following tree-



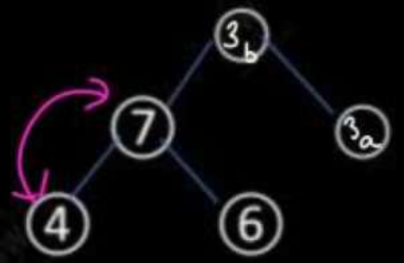
This is min heap.

Now remove root i.e. 3_c and place in *array* and the last node will become the root node to obtain the following Tree .



This is not min heap.

Interchange 7 and 3_b to obtain the following tree.



This is not min heap.

Interchange 4 and 7 to obtain the following tree -

This is min heap.

Now remove the roof 3_b and place in array -

2	3_c	3_b				

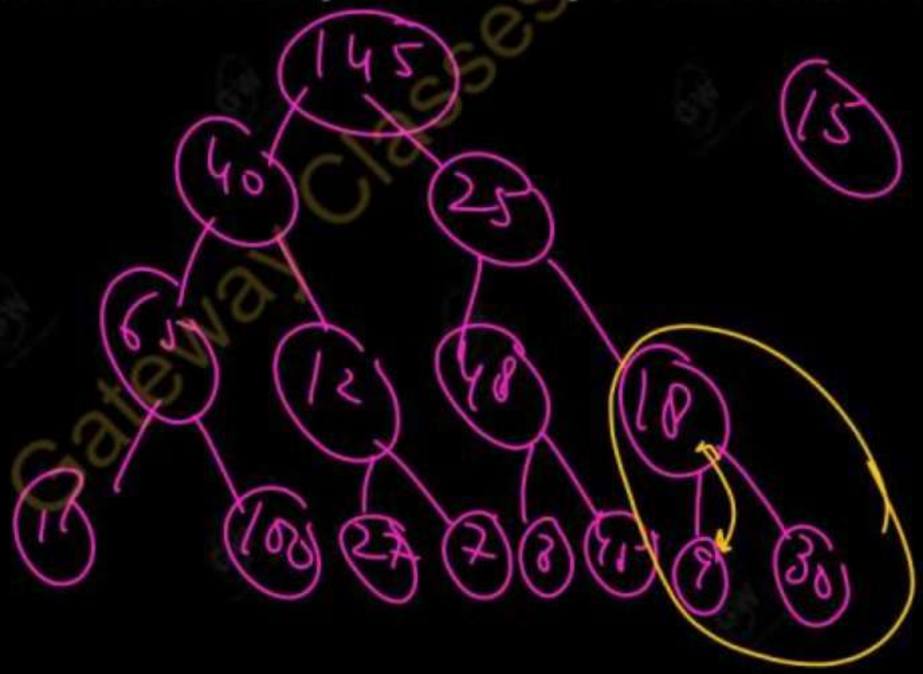
Now we can observe that in the sorted array 3_b is placed after 3_c which should come before 3_c to maintain the relative ordering of duplicate elements. But in this sorting algorithm it does not preserved the relative ordering of duplicate elements i.e. it is unstable sorting.

Heapify method

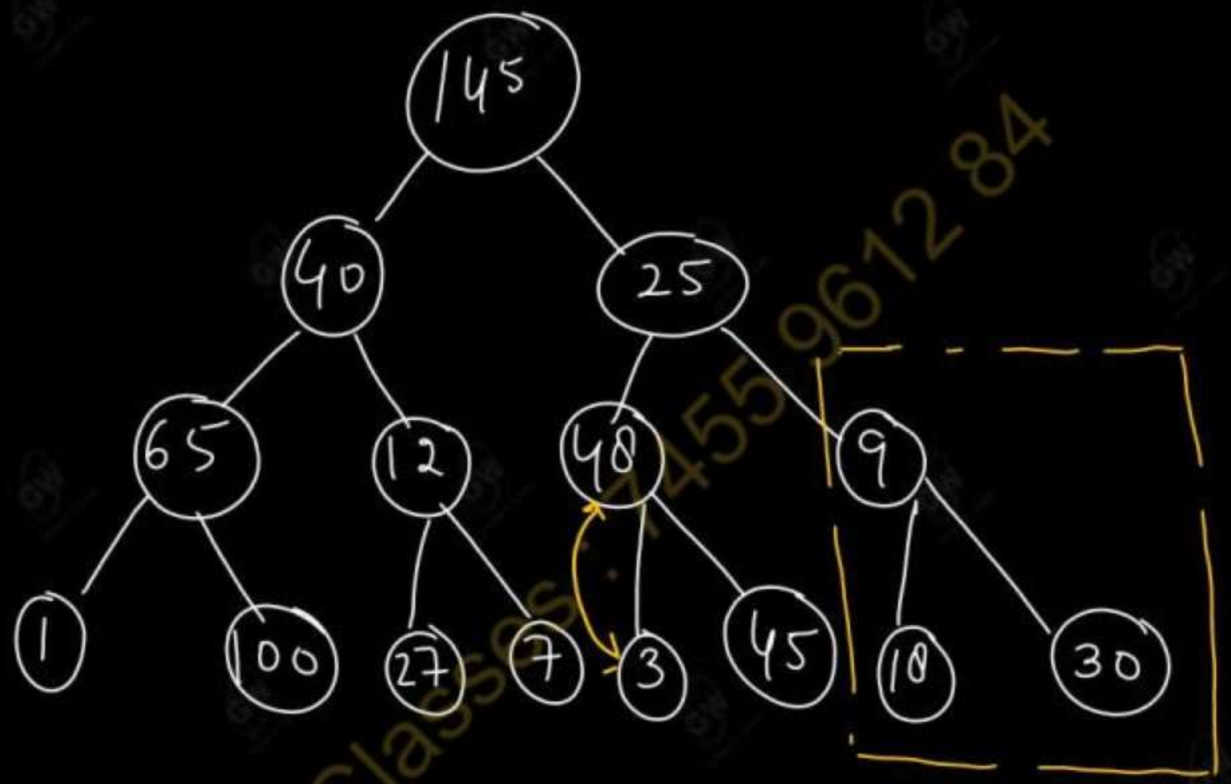
- In one by one key insertion method the complexity is n log n to create a heap for n elements.
- In Heapify method the home complexity is o(n) which is less then one by one key insertion method.
- Consider the following list of elements in the given order –

  145, 40, 25, 65, 12, 48, 18, 1, 100, 27, 7, 3, 45, 9, 30

Now this heapify method create a complete binary tree to obtain the following tree

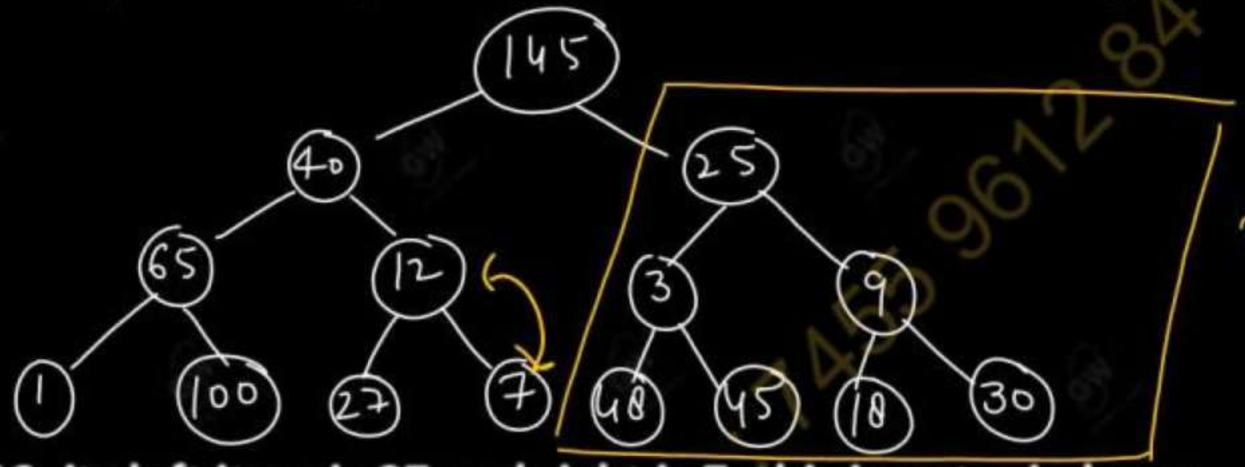


- Now convert this tree into min heap.
- In this heapify method, no need to perform any operation on element at leaf node i.e. no swapping is required on leaf node. (i.e. ignore the elements at leaf node i.e. number of swapping required for elements at leaf node = 0).
- It is to be observed that if number of elements in a tree are n then leaf node will contain n/2 elements.
- As in the previous tree number of elements were 15 (n) and at leaf node 8 (n / 2) elements are there i. e. ignoring half of the elements will effect the time complexity of the algorithm.
- Now consider the last element of non leaf nodes.
- The element is 18, its left child is 9 and right child is 30. This is not min heap. To make it min heap interchange, 9 and 18 to obtain the following tree.



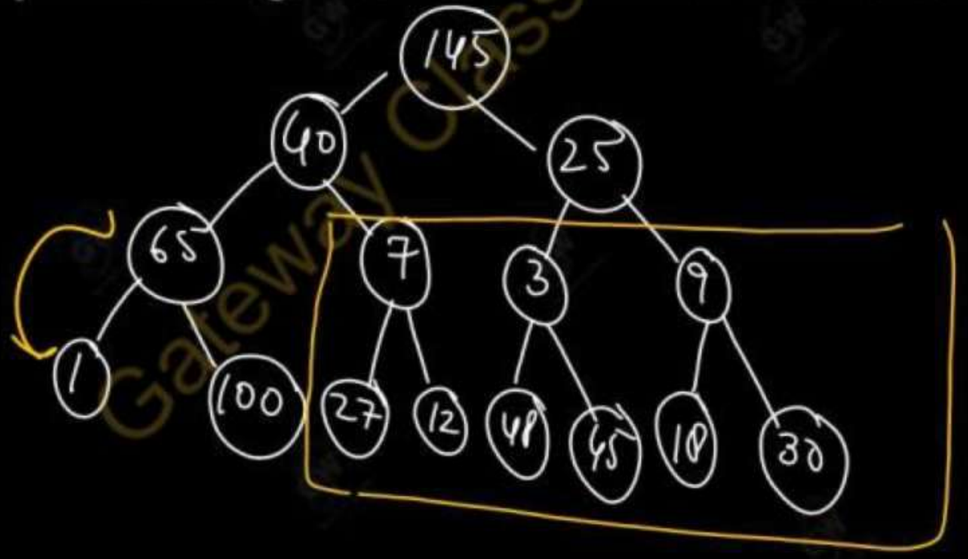
min heap

Now Consider 48, its left element is 3 and right element is 45, it is not min heap.
To make it min heap, interchange 3 and 48 to Obtain the following tree -



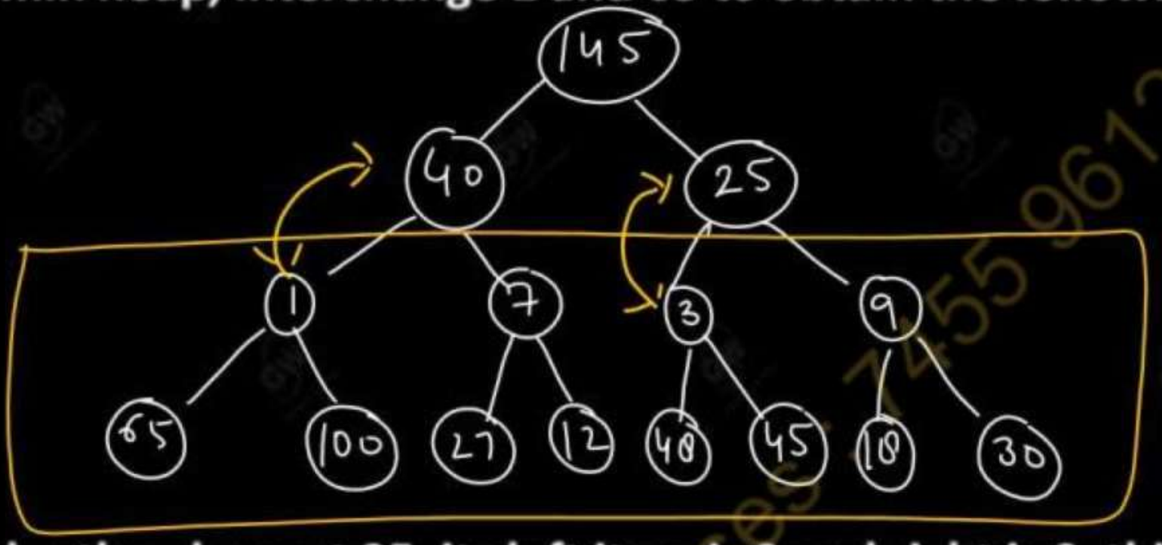
min heap

Now consider 12, its left item is 27 and right is 7, this is not min heap.
To make it min heap interchange 12 and 7 to obtain the following free-



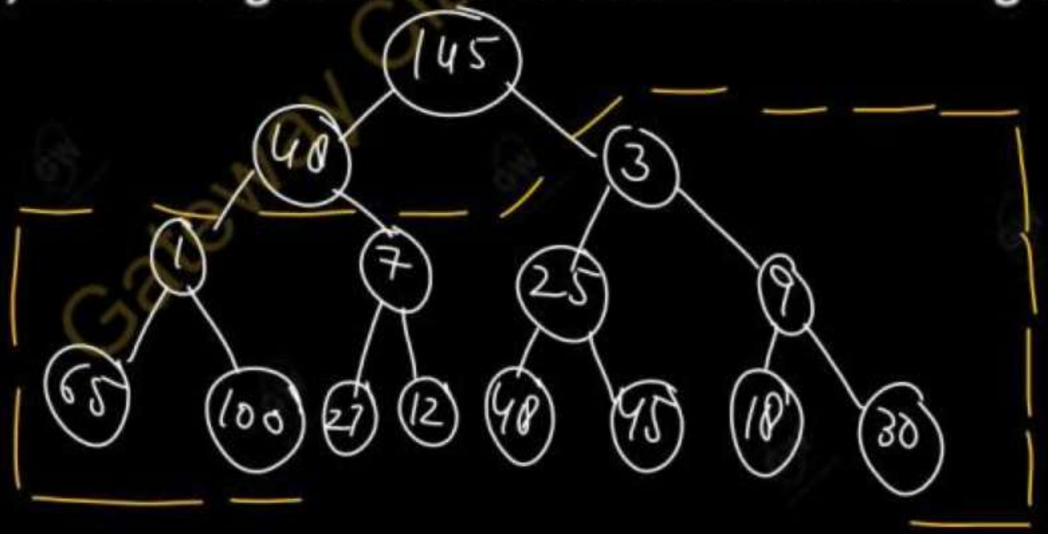
min heap.

Now Consider 65, its left element is 1 and 100 is its right element.
To make it min heap, interchange 1 and 65 to obtain the following tree-



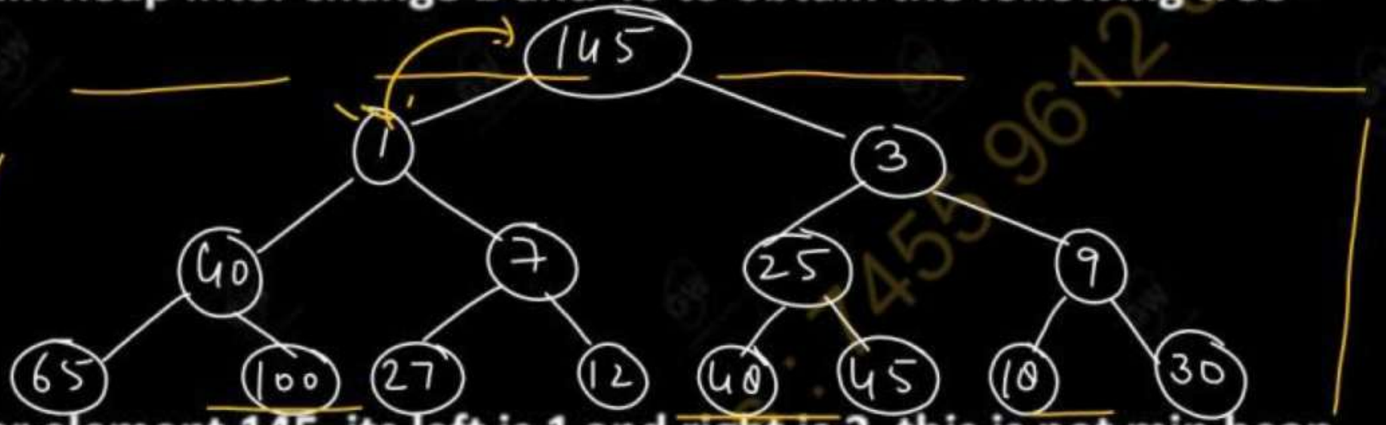
min heap

Now consider the element 25, its left item is 3 and right is 9, this is not a min heap.
To make it min heap, interchange 3 and 25 to obtain the following tree-



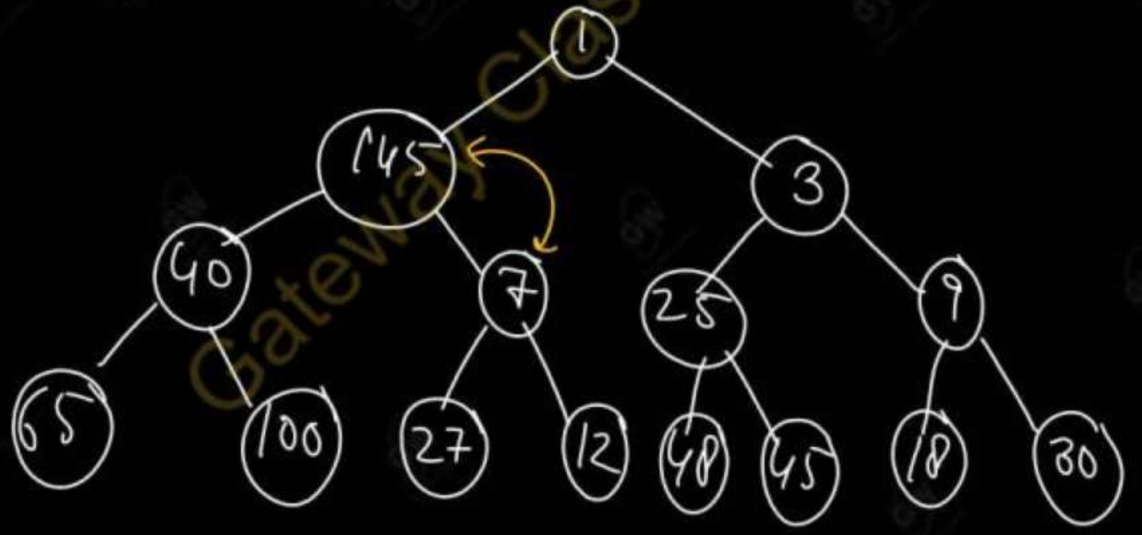
min heap

Now consider 40, its left is 1 and right is 7. This is not min heap.
To make it min heap inter change 1 and 40 to obtain the following tree-



min heap

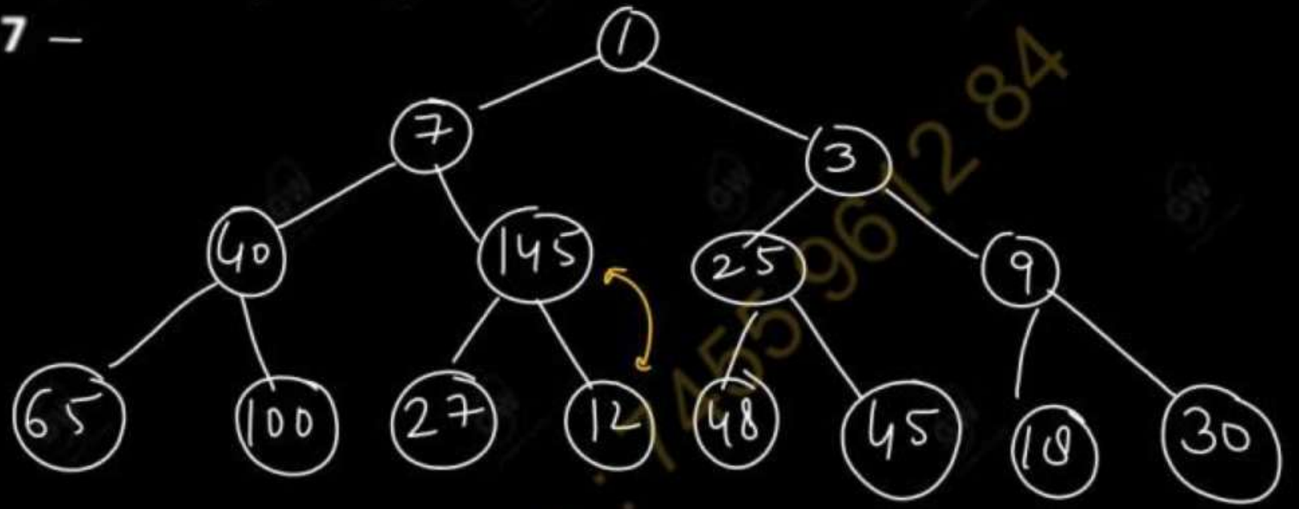
Now consider element 145, its left is 1 and right is 3, this is not min heap.
To make it min heap, interchange 1 and 145 to obtain the following tree -



not a min heap.

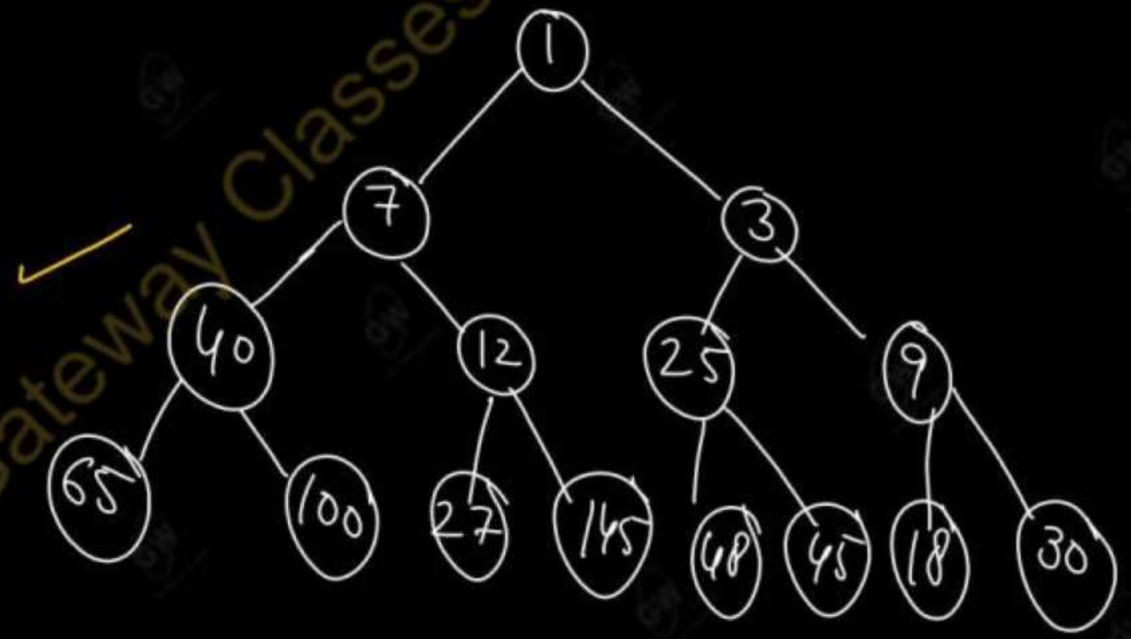
Interchange may be maximum up to highest level.

Interchange 145 and 7 –



not a min heap.

Interchange 145 and 12



min heap

Merge Sort

- It is a divide and conquer technique.
- Suppose the array A contains 14 elements as follows:-

Each pass of the merge sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows:-

Pass 1. Merge each pair of elements to obtain the following list of sorted pairs:

33, 66 22, 40 55, 88 11, 60 20, 80 44, 50 30, 77

Pass 2. Merge each pair of pairs to obtain the following list of sorted quadruplets:

22, 33, 40, 66 11, 55, 60, 88 20, 44, 50, 80 30, 77

Pass 3. Merge each pair of sorted quadruplets to obtain the following two sorted subarrays:

11, 22, 33, 40, 55, 60, 66, 88 20, 30, 44, 50, 77, 80

Pass 4. Merge each pair of sorted quadruplets to obtain the following two sorted subarrays:

11, 20, 22, 30, 33, 40, 44, 50, 55, 60, 66, 77, 80, 88

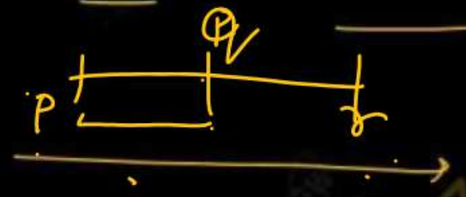
The original array A is now sorted.

sorted list of elements

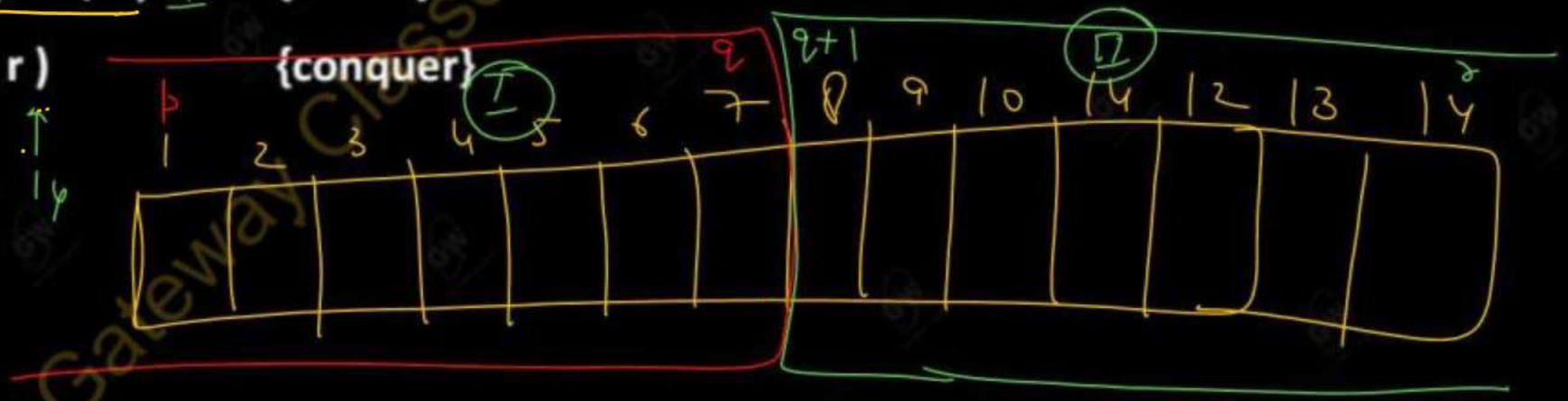
ALGORITHM : Merge Sort (A, p, r)

Where A = list of elements , p = index of first element , and r = index of last element

1. If (p < r) then { if p is equal to r then there is only one element, no sort operation is required in that case }
2. q = (p + r) / 2
3. Mergesort (A, p, q) I {divide}
4. Mergesort (A, q + 1 , r) II {divide}
5. Merge (A, p , q , r) {conquer}



$$q = \frac{1+14}{2} = \frac{15}{2} = 7$$



Merge (A, p, q , r)

1. n1 = q – p + 1 {calculating the size of first part }
2. n2 = r – q { Calculating the size of second part }
3. Create array L [1....n1 + 1] and R [1 ..... n2 + 1] {creating two arrays of plus 1 size}

{ Now copy value of original array to L and R }

4. for i = 1 to n1
5. do L [i] = A [p + i – 1]

copy 1 array

6. for j = 1 to n2
7. do R [j] = A [p + j]

second array

{ place ∞ to have bigger value for comparison }

8. L [n1 + 1] = ∞
9. R [n2 + 1] = ∞

{ set i and j at first index of two arrays}

10. i = 1
11. J = 1

{place elements in sorted order in resultant array}

12. for k = p to r
13. do if L [i] < = R [j]
14. then A [k] = L [i]
15. Set i = i + 1
16. else A [k] = R [j]
17. Set j = j + 1

Example:- Use merge sort algorithm to sort the following elements:-

	1	2	3	4	5	6	7	8
A:	15	10	5	20	25	30	40	35

Solution –

Mergesort(A,p,r)

p=index of I element = 1

r= index of last element=8

1. If (p<r) i.e. 1<8 true
2. q=(p+q)/ 2 =(1+8)/2 = 4
3. Mergesort(A,1,4)----------divide
4. Mergesort(A,5,8)----------divide
5. Merge(A,1,4,8)----------conquer

Call mergesort(A,1,4)

P=1,r=4

1. if(p<r) (1<4) true
2. q=(p+r)/2=(1+4)/2=2
3. Mergesort(A,p,q) (A,1,2)
4. Mergesort(A,q+1,r) (A,3,4)
5. Merge(A,1,2,4)

Now call mergesort(A,1,2) p=1,r=2

1. If p<r 1<2 true
2. q=(p+r)/2=1+2/2=1
3. Mergesort(A,2,2)
4. Merge(A,1,1,2)

Now call mergesort (A,1,1)

P=1,r=1

1. If p<r 1<1 false

Then call mergesort(A,2,2)

P=2,r=2

1. If p<r 2<2 false

Now call merge(A,1,1,2)

i.e. p=1,q=1,r=2

1. n1=q-p+1=1-1+1=1

2. n2=r-q=2-1=1

3. Create arrays L[1……..n1+1] and R[1……..n2+1]

L[1……..2] and R[1……..2]

4. For i=1 to n1 i=1 to 1

5. L[i]=A[p+i-1]

L[1]=A[1+1-1]

L[1]=A[1]=15 L[1]=15

6. For j=1 to n2

J=1 to 1

7. Do R[j]=A[q+j]

R[1]=A[1+1]

R[1]=A[2] R[1]=10

8. L[1+1]= ∞ L[2]=

9. R[1+1]= ∞ R[2]=

10. i=1

11. j=1

12. For k=p to r

k=1 to 2

13. Do if L[i]<=R[j]

L[1]<=R[1]

15<=10 false

16. A[K]=R[j]

A[1]=R[1]

A[1]=10

10	15

17. j=i+1=2

12. k=2 to 2

13. Do if L[i]<=R[j]

L[1]<=R[2]

15<= true

14. Then A[k]=L[i]

A[2]=L[1]

A[2]=15

15. i=1+1=2

Now mergesort (A,1,2) is completed.

Now call mergesort(A,3,4)

In the same manner execution will take place.

Merge Sort Through Recursion

```
#include<stdio.h>
#define MAX 20
int a[MAX];
void merge(int low, int mid, int high)
{
int temp[MAX];
int i=low,j=mid+1,k=low;
while((i<=mid) && (j<=high))
{
if(a[i]<a[j])
{
temp[k]=a[i];
k++;
i++;
}
else
{
temp[k]=a[j];
k++;
j++;
}
}
while(i<=mid)
{
temp[k]=a[i];
k++;
i++;
}
while(j<=high)
{
temp[k]=a[j];
k++;
j++;
}
for(i=0;i<=high;i++)
a[i]=temp[i];
}
```

```
void mergesort(int low, int high)
{
int mid;
if(low!=high)
{
mid=(low+high)/2;
mergesort(low,mid);
mergesort(mid+1,high);
merge(low,mid,high);
}
}
void main()
{
int i,n;
clrscr();
printf("\n enter the number of elements:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\n enter element %d:",i+1);
scanf("%d",&a[i]);
}
printf("\n Undorted list is:");
for(i=0;i<n;i++)
printf("\n %d",a[i]);
mergesort(0,n-1);
printf("\n sorted list is:");
for(i=0;i<n;i++)
printf("\n %d",a[i]);
getch();
}
```

mid divide divide

Radix Sort

- It is non- comparison based algorithm.
- Quick sort, merge sort, selection sort Bubble sort etc. are comparison bared sorting algorithm.
- Consider the following elements - 904, 46, 5, 74, 62, 1

Number	LSB	10' Place	100 Place (MSB)
904	I Pass: 001	II Pass: 001	III Pass: 001
046	062	904	005
005	904	005	046
074	074	046	062
062	005	062	074
001	046	074	904

Sorted array

- The biggest number is of 3 digits so insert 0 with all numbers (prior to the number)

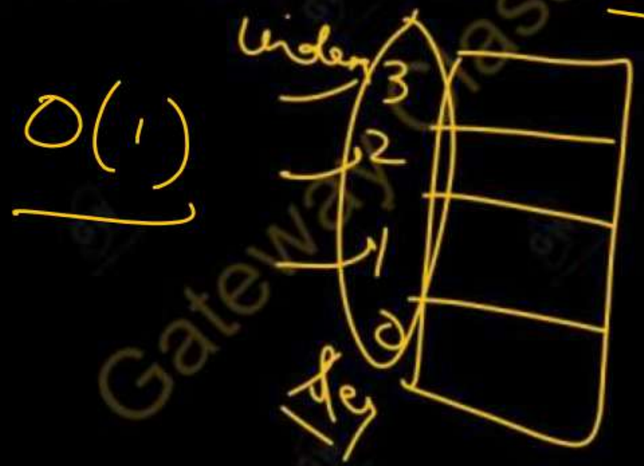
- Radix means base i.e. first consider least significant bit or number at unit place to sort numbers at unit place.
- do not change relative position of duplicate number i.e. in the previous example 904 and 074, 904 will come first in the list.
- Radix sort is stable sorting technique.

ALGORITHM -

1. Identify the maximum number of the list.
2. Count the number of digits in in the maximum number. This gives us the of passes required for the sorting process.
3. For each pass, starting from the least significant digit (rightmost), sort the numbers by grouping them according to their digit values.
4. Repeat the above step for each digit, moving from right to left (i.e. from the least significant digit to the most significant digit).
5. *At* the end of the last pass, the list is sorted.

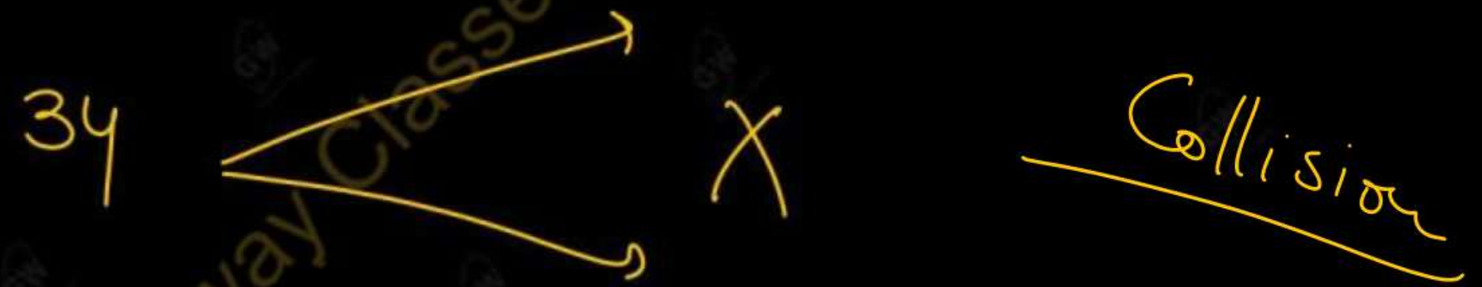
What is Hash Table?

- Hash tables support one of the most efficient types of searching i.e. **hashing.**
- A hash table consists of an array in which data is accessed via a special index called a key.
- The primary idea behind a hash table is to establish a mapping between the set of all possible keys and positions in the array using a hash function.
- A hash function accepts a key and returns its hash coding i.e. hash value.
- Since both computing a hash value and indexing into an array can be performed in constant time, **the beauty of hashing is that we can use it to perform constant time searches.**



O(1)

- When a hash function can guarantee that no two keys will generate the same hash coding, the resulting hash table is said to be directly addressed.
- This is ideal, but direct addressing is rarely possible in practice.
- Consequently, most hash functions map some keys to the same position in the table.
- When two keys map to the same position, they collide.
- A good hash function minimizes collisions, but we must still be prepared to deal with them.



What is Hashing?

- We have seen different searching techniques where search time is basically dependent on the number of elements.
- Sequential search, binary search and all the search trees are totally dependent on number of elements and so many key comparisons are also involved.

For example,

Consider the Unsorted sequential array implementation-

- insert: add to back of arrays; O (1)
- find: search through the keys one at a time, potentially all of the keys; O(n)
- remove: find and replace removed node with last node; O(n)

Consider the Sorted sequential array implementation

- insert: add in sorted order; O(n)
- find: binary search $O(\log_2 n)$
- remove: find, remove node and shuffle down; O(n)

Consider the Linked list (unsorted or sorted) implementation

- insert: add to front: O(1) or O(n) for a sorted list
- **find:** search through potentially all the keys, one at a time; O(n) still O(n) for a sorted list
- **remove:** find, remove using pointer alternations; O(n)

- Our need is to search the element in constant time and less key comparisons should be in the range 0 to N-1.
- Now we are storing the records in array based on the key where array involved.
- Suppose all the elements are in array size *N*.
- Let us take all the keys are unique and index and keys are same.
- Now we can access the record in constant time and there no key comparisons are involved.

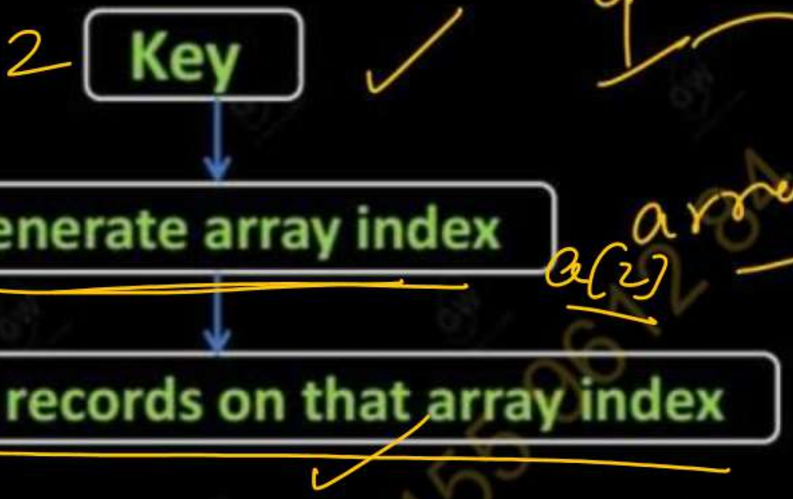
- Let us take the 5 records where keys are:

Key: 9 4 6 7 2

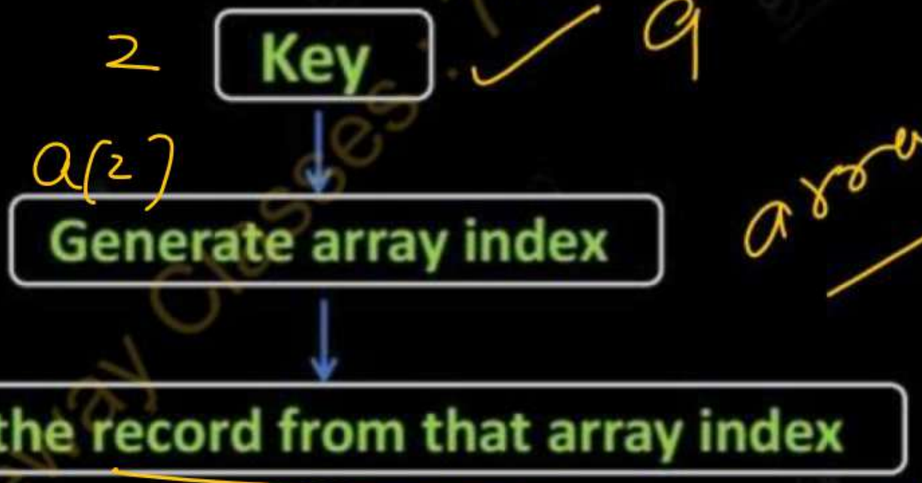
- It will be stored in array as:

- Here we can see the record which has key value '2' can be directly accessed through array index array[2].
- Now the idea that comes in picture is hashing where we will convert the key into array index and put the records in array and in the same way for searching the record, convert the key into array index and get the records from array.
- This can be described as in the next figure-

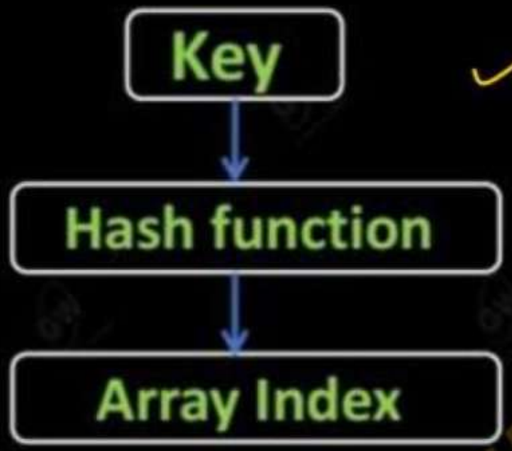
- For storing record:



- For accessing record:



- The generation of array index uses hash function, which converts the keys into array index and the array which supports hashing for storing record or searching record called hash table.



- So we can say each key is mapped on a particular array index through hash function.
- Collision - If each key is mapped on a unique hash table address then this situation is ideal situation but there may be possibility that our hash function is generating same hash table address for different keys, this situation is called "collision".
- So our hash function should generate unique address but it is not possible, only one thing we can do is to take the good hash function for minimizing collision.

HASH FUNCTIONS

- Hash functions is a function which, when applied to the key, produced an integer which can be used as an address in a hash table.
- The intent is that elements will be relatively randomly and uniformly distributed.
- **Perfect Hash Function** is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions.
- **Good Hash Function** minimizes collisions by spreading the elements uniformly throughout the array.
- There is no magic formula for the creation of the hash function. It can be any mathematical transformation that produces a relatively random and unique distribution of values within the address space of the storage.

Characteristics of a Good Hash Function

There are *four* main characteristics of a good hash function:

1. The hash value is fully determined by the data being hashed. - If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.

2. The hash function uses all the input data. - If the hash function doesn't use all the input data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions.

3. The hash function "uniformly" distributes the data across the entire set of possible hash values. - If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.

4. The hash function generates very different hash values for similar strings. - In real world applications, many data sets contain very similar data elements. We would like these data elements to still be distributable over a hash table.

However, finding a perfect hash function that works for a given data set can be extremely time consuming and very often it is just impossible.

Therefore, we must live with collisions and learn how to handle them.

Types of Hash Functions

Some common Hash functions are:

1. **Division Method -**

- **Choose a number m larger than the number n of keys in K.**
- In the division method for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m.
- That is, the hash function is: $h(k) = k \bmod m$ or $h(k) = k \bmod m + 1$

For example, if the hash table has size $m = 12$ and the key is $k = 100$, the $h(k) = 100 \bmod 12 = 4$.

{ K mod 10 may also be a hash function.}

- The second formula is used when we want the hash addresses to range from 1 to m rather than 0 to m-1.
- Since it requires only a single division operation, hashing by division is quite fast.

2. Midsquare Method -

- In midsquare method we square the key, after getting number we take some digits from the middle of that number as an address.
- Then the hash function is defined as - $H(k) = l$
- Where l is obtained by deleting digits from both ends of k^2.
- Let us take some 4 digit number as a key: -

1337	1273	1391	1026

- Now square these keys: -

1787569	1620529	1934881	1052676

- Now take the 3rd and 4th digit from each number and that will be the hash address of these keys.
- Let us assume here table size is 1000. So the hash address for these keys will be 75, 05, 48 and 26.

3. Folding Method -

- The key k is partitioned into a number of parts where each part, except possibly the last, has same number of digits as the required address.
- Then the parts are added together, ignoring the last carry.
- That is, H(k) = k1 + k2 + ...
- One easiest way to compute the key is break the key into pieces, add them and get the hash address.
- Let us take some 8 digits key:

  82394561, 87139465, 83567271, 85943228

- Now chop them in pieces of 3, 2 and 3 digits and add them.

- So the address will be -

  82394561 = 823 + 94 + 561 = 1478

  87139465 = 871 + 39 + 465 = 1375

  83567271 = 835 + 67 + 271 = 1173

  85943228 = 859 + 43 + 228 = 1130

- Now truncate them up to the digit based on the size of hash table. Suppose the table size is 1000 so the hash address can be from 0 to 999. So we will truncate here the higher digit of number. Now the hash address of keys will be as:

  H (82394561)= 478

  H (87139465)= 375

  H (83567271) = 173

  H (85943228) = 130

Example – Consider the company with 68 employees, is assigned a unique 4 digit employee number to each employee which is used as the primary key in company's employee file. Suppose L consists of 100 two digits addresses: 00,01,02,.....99. Apply the hash functions to each of the following employee numbers:

3205, 7148, 2354

Solution –

1. **Division Method –** Choose a prime number m close to 99, such as m=97. Then

   H (K) = K mod m , H(3205) = 3205 mod 97 = 4

   H (K) = K mod m, H(7148) = 7148 mod 97 = 67

   H (K) = K mod m, H(2345) = 2345 mod 97 = 17

In the case that the memory addresses begin with 01 rather 00, we choose that the function –

H(k) = k (mod m) + 1 to obtain

$H(K) = K \pmod m + 1$, $H(3205) = 3205 \bmod 97 + 1 = 5$

$H(K) = K \pmod m + 1$, $H(7148) = 7148 \bmod 97 + 1 = 68$

$H(K) = K \pmod m + 1$, $H(2345) = 2345 \bmod 97 + 1 = 18$

2. Midsquare Method:- The following calculations are performed –

k:	3205	7148	2345
k^2:	10272025	51093904	5499025
H(k)	72	93	90

Observe that the 4th and 5th digits, counting from the right, are chosen from the hash address.

3. Folding Method:- Chopping the key k into two parts and adding yields the following hash addresses:

$H(3205) = 32 + 05 = 37$

$H(7148) = 71 + 48 = 119$

$H(2345) = 23 + 45 = 68$

Observe that the leading digit 1 in H(7148) is ignored.

Alternatively, one may want to reserve the second part before adding, thus producing the following hash addresses:

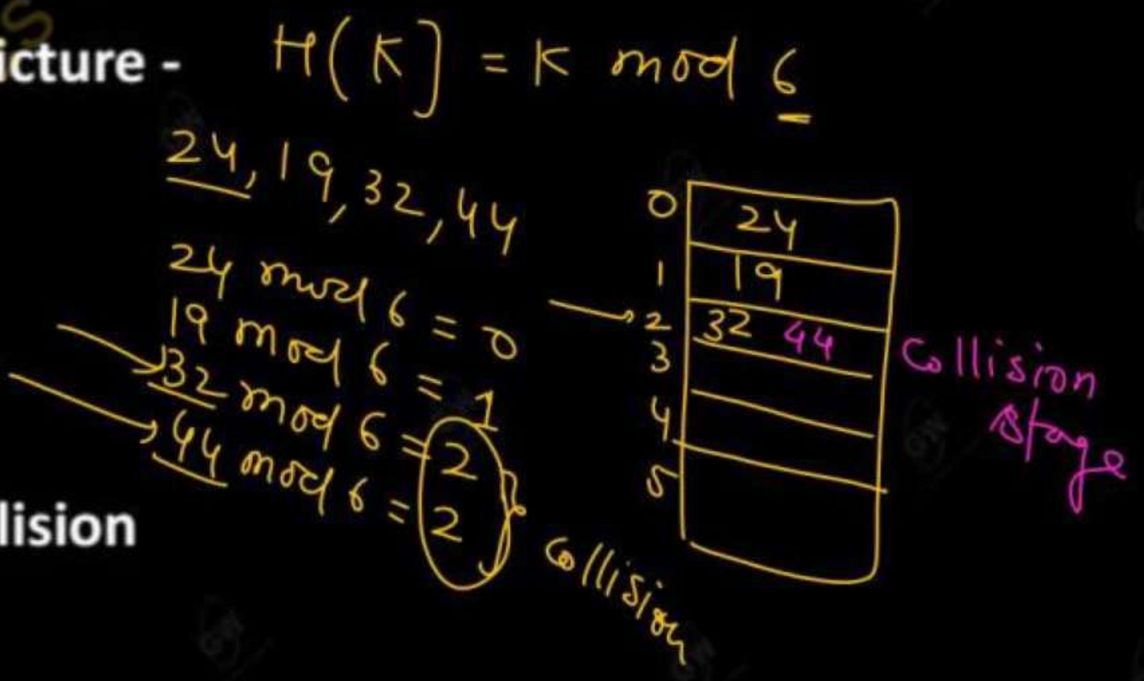
H(3205) = 32 + 50 = 82

H(7148) = 71 + 84 = 55

H(2345) = 23 + 54 = 77

COLLISION RESOLUTION TECHNIQUES

- Suppose we want to add a new record R with key k to file *F*, but suppose the memory location address H (k) is already occupied. This situation is called collision.
- Lets take an example – consider the hash function – H (K) = K mod 6 for the following key values : - 24, 19, 32, 44 , Because mod value is 6 then hash table will be from 0 to n – 1 , i.e., 0 to 5 as shown in picture -
- Key = 24, H (24) = 24 mod 6 = 0
- Key = 29, H (19) = 19 mod 6 = 1
- Key = 32, H (32) = 32 mod 6 = 2
- Key = 44, H (44) = 44 mod 6 = 2 , i.e. collision



$H(K) = K \bmod 6$

24, 19, 32, 44

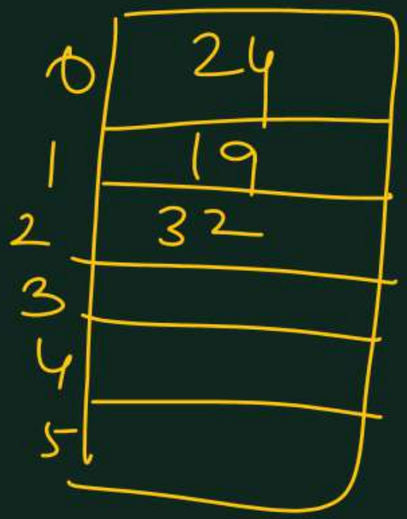
$H(24) = 24 \bmod 6 = 0$

$H(19) = 19 \bmod 6 = 1$

$H(32) = 32 \bmod 6 = 2$

$H(44) = 44 \bmod 6 = 2$

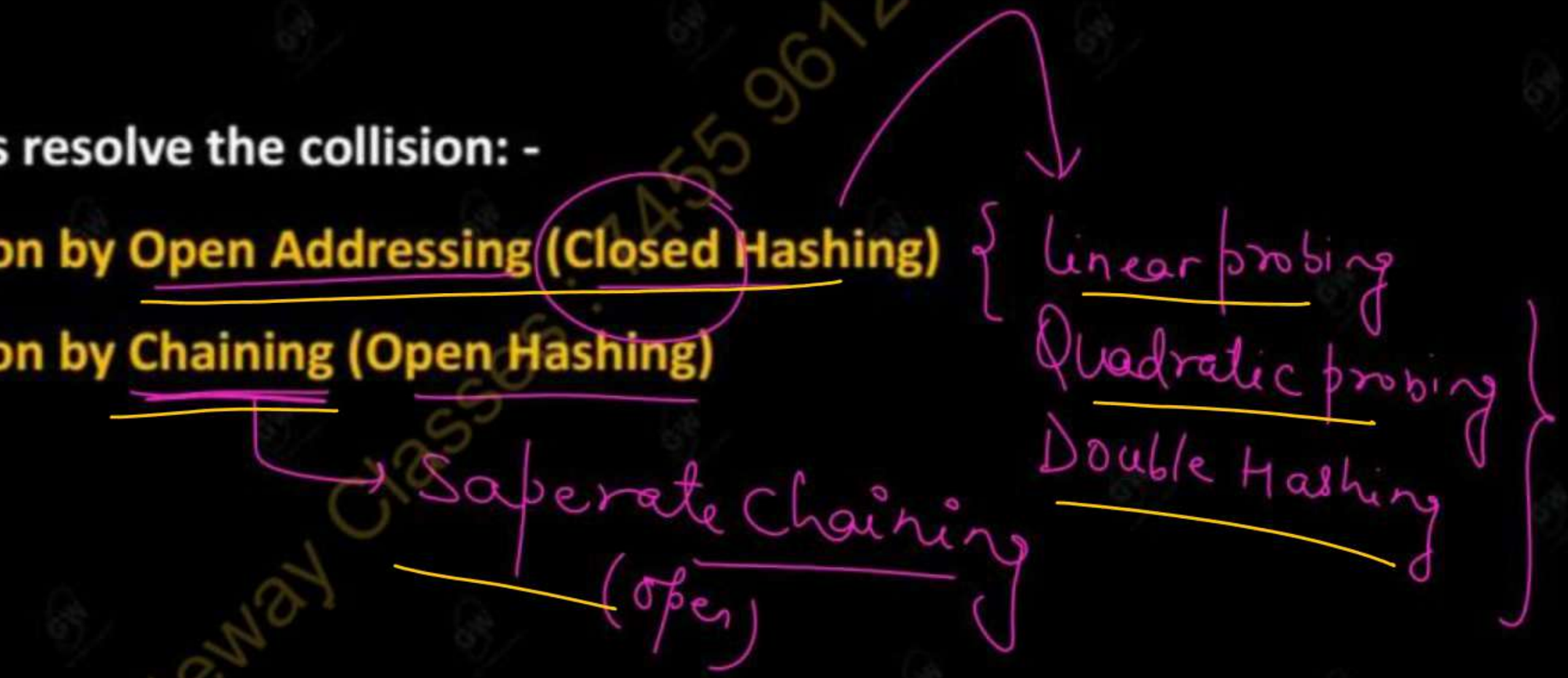
Collision



COLLISION RESOLUTION TECHNIQUES

- That is, a collision occurs when more than one keys map to same hash value in the hash table.
- The following ways resolve the collision: -

1. Collision Resolution by Open Addressing (Closed Hashing)
2. Collision Resolution by Chaining (Open Hashing)



HASHING WITH OPEN ADDRESSING

- In open addressing, all elements are stored in the hash table itself.
- When searching for an element, we systematically examine table slots until the desired element is found or it is clear that the element is not in the table.
- The advantage of open addressing is that it avoids pointers altogether (as it is in chaining). Instead of follow pointers, we compute the sequence of slots to be examined.
- The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.
- The process of examining the locations in the hash table is called 'Probing'.
- To perform insertion using open addressing, we successively examine, or probe, the hash table until we find an empty slot in which to put the key.

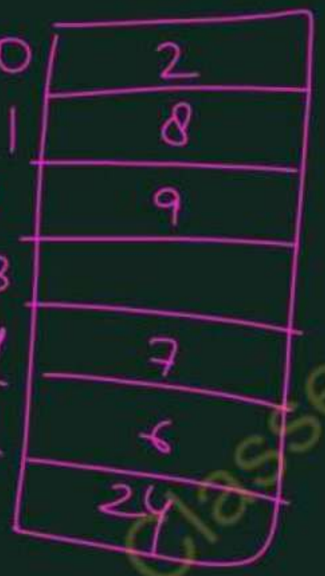
Hash fun^{n}

$24 \% 7 = 3$ (Probe = 1)

$24 - 4$ (Probe 2)

$24 - 5$ (Probe 3)

$24 - 6$ (Probe 4)



Three techniques are commonly used to compute the probe sequences required for open addressing:

1. Linear Probing,
2. Quadratic Probing, and
3. Double Hashing

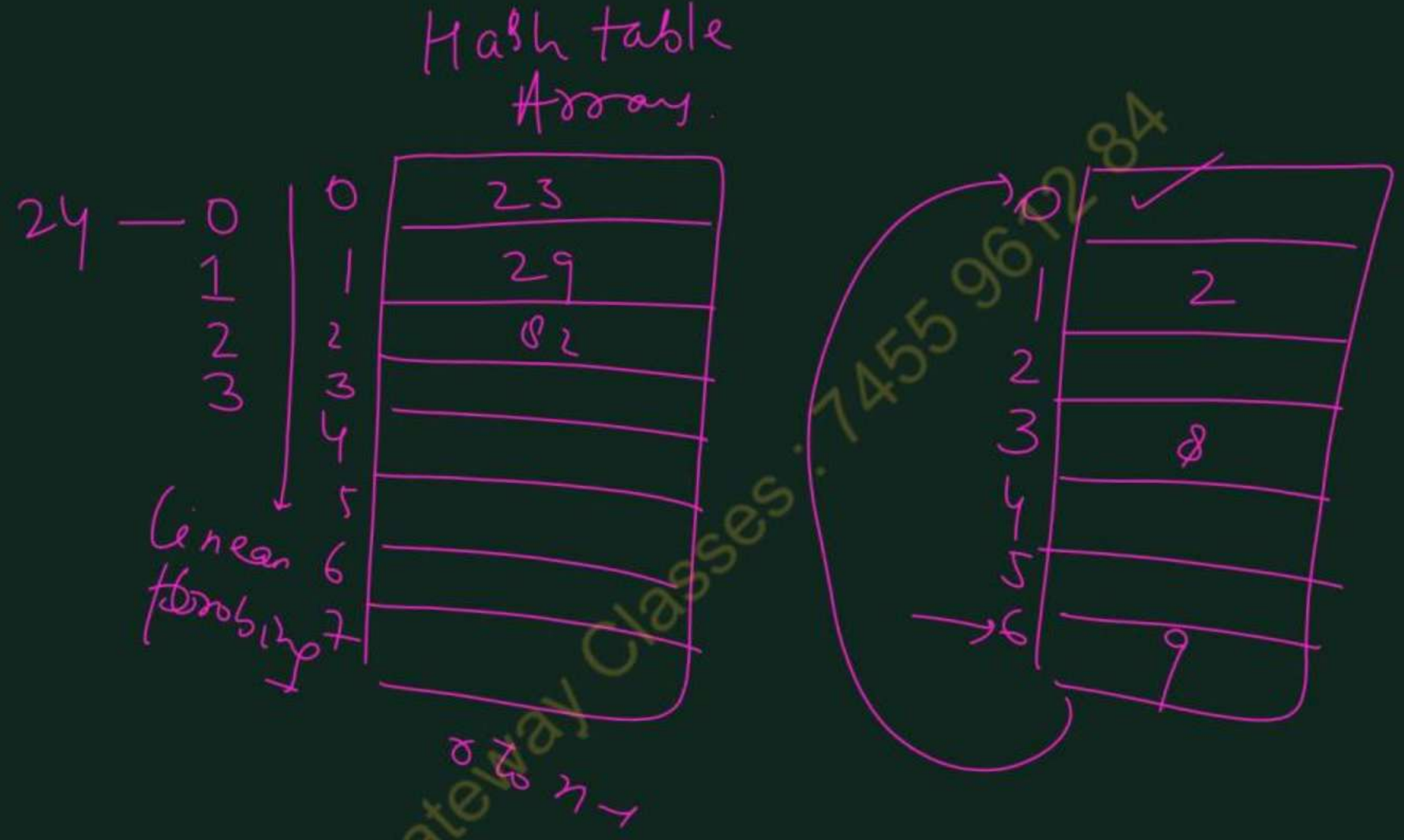
Let us discuss all three techniques one by one.

1. Linear Probing

- Given an ordinary hash function h': U [0, 1,..... m – 1], the method of linear probing use the hash function.

$$h(k,i) = (h'(k) + i) \bmod m$$

- where 'm' is the size of the hash table and h' (k) = k mod m (i.e. basic hash function).
- For i = 0, 1....., m – 1. Given key k, the first slot probed is T [h' (k)].
- We next probe slot T [h' (k) + 1] , and so on up to the slot T [m – 1].
- Then we wrap around to slots T [0] , T [1],....., until we finally probe slot T [h' (k) – 1].
- Since the initial probe position determines the entire probe sequence, only m distinct probe sequences are used with linear probing.



EXAMPLE 1 :- Consider inserting the key 26, 37, 59, 76, 65, 86 into a hash-table of size m = 11 using linear probing, consider the primary hash function is h' (k) = k mod m.

Solution: Initial state of the hash table is as follows : -

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 26. We know h (k, i) = [h' (k) + i] mod m

Now h (26, 0) = [26 mod 11 + 0] mod 11 = (4 + 0) mod 11 = 4 mod 11 = 4

Since T [4] is free, insert key 26 at this place.

2. Insert 37. Now h (37, 0) = [37 mod 11 + 0] mod 11= [4 + 0] mod 11 = 4

Since T [4] is not free, the next probe sequence is computed as -

h (37, 1) = [37 mod 11 + 1] mod 11 = [4 + 1] mod 11 = 5 mod 11 = 5

T [5] is free, insert key 37 at this place.

Insert 26

$H(k) = k \bmod m$

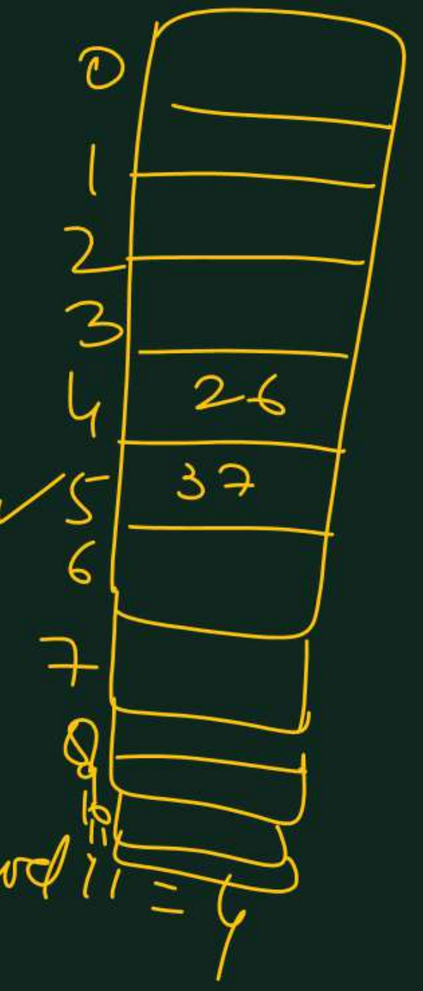
$H(26) = 26 \bmod 11$

$h(K, i) = (h(K) + i) \bmod m$

$h(26, 0) = (h(26) + 0) \bmod 11$

$= (26 \bmod 11 + 0) \bmod 11$

$= (4 + 0) \bmod 11 = 4 \bmod 11 = 4$



Insert 37

$$h(K, i) = (h(K) + i) \bmod m$$

$$h(37, 0) = (37 \bmod 11 + 0) \bmod 11$$

$$= 4 \bmod 11 = 4$$

Can not store the Collision

$$h(K, i) = ((37 \bmod 11) + 1) \bmod 11$$

$$= 4 + 1$$

$$5 \bmod 11 \Rightarrow 5$$

m = 11

26, 37, 59, 76, 65, 86

① Insert 26

$h(k) = h(k) \bmod m$

$h(26) = 26 \bmod 11$

$= 4$

② Insert 37

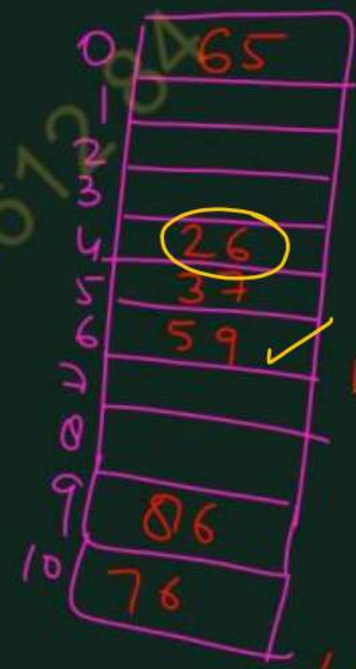
$h(37) = 37 \bmod 11$

$= 4$ Collision

$h(k,i) = (h(k) + i) \bmod m$

$h(37,1) = (4+1) \bmod 11 = 5 \bmod 11$

$= 5$



③ Insert 59

$h(k) = h(k) \bmod m$

$h(59) = 59 \bmod 11$

$= 4$ Collision

$h(59,1) = (h(59)+1) \bmod 11$

$= (4+1) \bmod 11$

$= 5$ Collision

$h(59,2) = (h(59)+2) \bmod 11$

$= (4+2) \bmod 11$

$= 6 \bmod 11 = 6$

3. Insert 59. Now h (59, 0) = [59 mod 11 + 0] mod 11 = [4 + 0] mod 11 = 4

T [4] is not free, so the next probe sequence is computed as -

H [59, 1] = [59 mod 11 + 1] mod 11 = 5

T [5] is also not free, so the next probe sequence is computed as -

H (59, 2) = (59 mod 11 + 2) mod 11 = 6 mod 11 = 6

T [6] is free. Insert key 59 at this place.

4. Insert 76. h (76, 0) = (76 mod 11 + 0) mod 11 = (10 + 0) mod 11 = 10

T [10] is free, insert key at this place.

5. Insert 65. h (65, 0) = (65 mod 11 + 0) mod 11 = (10 + 0) mod 11 = 10

T [10] is occupied, the next probe sequence is computed as -

H (65, 1) = (65 mod 11 + 1) mod 11 = (10 + 1) mod 11 = 11 mod 11 = 0

T [**0**] is free, insert key 65 at this place.

6. Insert 86. h (86, 0) = (86 mod 11 + 0) mod 11 = 9 mod 11 = 9

T [9] is free, insert key 86 at this place.

Thus,

0	1	2	3	4	5	6	7	8	9	10
65	I	I	I	26	37	59	I	I	86	76

$h(K) = h(K) \bmod 11$

$h(65) = 65 \bmod 11$

$= 10$

$h(65, 1) = (10+1) \bmod 11$

$= 11 \bmod 11$

$= 0$

$h(86) = 86 \bmod 11 = 9$

2. Quadratic Probing

- Suppose a record R with key k has the hash address H (k) = h then instead of searching the locations with address h, h +1, h + 2, we linearly search the locations with addresses.

  h, h +1, h + 4, h + 9,....., h + i^2, ...

- Quadratic probing uses a hash function of the form -

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

- Where (as in linear probing) h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and i = 0, 1, ..., m-1.
- The initial position probed is T (h' (k)); later positions probe are offset by amounts that depend in a quadratic manner on the probe number i.
- This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2 and m are constrained.

EXAMPLE 2: Consider inserting the key 76, 26, 37, 59, 21, 65 into a hash table of size m = 11 using quadratic probing with $c_1 = 1$ and $c_2 = 3$. Further consider that the primary hash function is h' (k) = k mod m.

Solution: For quadratic probing, we have -

$$h(k, i) = [k \bmod m + c_1 i + c_2 i^2] \bmod m$$

0	1	2	3	4	5	6	7	8	9	10
/	/	/	/	/	/	/	/	/	/	/

This is the initial state of the hash table.

Here $c_1 = 1,\ c_2 = 3$

$$h(k, i) = [k \bmod m + i + 3\, i^2] \bmod m$$

Insert(76)

$$h(76,0) = (76 \bmod 11 + 1 \times 0 + 3 \times 0^2) \bmod 11$$

$$= (10 + 0 + 0) \bmod 11 = 10$$

1. Insert 76. h (76, 0) = (76 mod 11 + 0 + 3 x 0) mod 11

= (10 + 0 + 0) mod 11 = 10

T [10] is free, insert the key 76 at this place.

2. Insert 26. h (26, 0) = (26 mod 11 + 0 + 3 x 0) mod 11 = (4 + 0 + 0) mod 11 = 4

T [4] is free, insert the key 26 at this place.

3. Insert 37.

h (37, 0) = (37 mod 11 + 0 + 3 x 0) mod 11 = (4 + 0 + 0) mod 11 = 4 Collision

T [4] is not free, so next probe sequence is computed as -

h (37, 1) = (37 mod 11 + 1 + 3 x 1^2) mod 11

= (4 + 1 + 3) mod 11 = 8 mod 11 = 8

T [8] is free. Insert the key 37 at this place.

37

$h(37,0) = (37 \bmod 11 + 1 \times 0 + 3 \times 0) \bmod 11$

$= 4 \bmod 11 = 4$

$h(37,1) = (37 \bmod 11 + 1 \times 1 + 3 \times 1) \bmod 11$

$= (4 + 1 + 3) \bmod 11$

$= 8 \bmod 11 = 8$

4. Insert 59.

h (59, 0) = (59 mod 11 + 0 + 3 x 0) mod 11 = (4 + 0 + 0) mod 11 = 4 mod 11 = 4 Collision.

T [4] is not free, so next probe sequence is computed as -

h (59, 1) = (59 mod 11 + 1 + 3 x 1^2) mod 11 = (4 + 1 + 3) mod 11 = 8 mod 11 = 8 Collision

T [8] is also not free, so the next probe sequence is computed as -

h (59, 2) = (59 mod 11 + 2 + 3 x 2^2) mod 11 = (4 + 2 + 12) mod 11 = 18 mod 11 = 7

T [7] is free, insert key 59 at this place.

5. Insert 21.

h (21, 0) = (21 mod 11 + 0 + 3 x 0) mod 11 = (10 + 0 + 0) mod 11 = 10 mod 11 = 10

T [10] is not free, the next probe sequence is computed as -

h (21, 1) = (21 mod 11 + 1 + 3 x 1^2) mod 11 = (10 + 1 + 3) mod 11 = 14 mod 11 = 3

T [3] is free, so insert key 21 at this place.

6. Insert 65. h (65, 0) = (65 mod 11 + 0 + 3 x 0) mod 11 = (10 + 0 + **0**) mod 11 = 10 mod 11 = 10

Collision

Since, T [10] is not free, the **next** probe sequence **is** computed as -

h (65, 1) = (65 mod 11 + 1 + 3 x 1^2) mod 11 = (10 + 1 + 3) mod 11 = 14 mod 11 = 3

Collision

T [3] is not free, so the next probe sequence is computed as -

h (65, 2) = (65 mod 11 + 2 + 3 x 2^2) mod 11 = (10 + 2 + 12) mod 11 = 24 mod 11 = 2

T [2] is free, insert the key 65 at this place.

Thus, after inserting all keys, the hash table is:

0	1	2	3	4	5	6	7	8	9	10
l	*l*	*65*	*21*	*26*	*l*	*l*	*59*	*37*	*l*	*76*

3. Double Hashing

- Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutation.
- Double hashing uses a hash function of the form:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

$h(76, 0) = 76 \bmod 11 + 0 \times 76 \bmod 11 \bmod 11 = 10$

- where h_1, and h_2, are auxiliary hash functions and m is the size of the hash table.
- h_1 (k) = k mod m or h_2 (k) = k mod m'. Here m' is slightly less than m (say m - 1 or m – 2)
- The initial position probed is T (h, (k)) successive probe positions are offset from previous positions by the amount h_2(k) modulo m.
- Thus, unlike the case of linear or quadratic probing the probe sequence here depends in two ways upon the key k, since the initial probe position the offset, or both, may vary.

- Double hashing represents an improvement over linear or quadratic probing in that probe sequences are used, rather than θ (m), since each possible (h_1 , (k) , h_2 (k)) pair yields a distinct probe sequence, and as we vary the key, the initial probe position h_1 (k) and the offset h_2 (k) may vary independently.
- As a result, the performance of double hashing appears to be very close to the performance of the " ideal " scheme of uniform hashing.

EXAMPLE 3. Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size m = 11 using double hashing. Consider that the auxiliary hash functions are $h_1 (k) = k \bmod 11$ and $h_2 (k) = k \bmod 9$.

Solution: Initial state of hash table is:

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/ 76

1. Insert 76.

$h_1 (76) = 76 \bmod 11 = 10$

$h_2 (76) = 76 \bmod 9 = 4$

$h (76, 0) = (10 + 0 \times 4) \bmod 11 = 10 \bmod 11 = 10$

T [10] is free, so insert key 76 at this place.

2. Insert 26.

$h_1(26) = 26 \bmod 11 = 4$

$h_2(26) = 26 \bmod 9 = 8$

$h(26, 0) = (4 + 0 \times 8) \bmod 11 = 4 \bmod 11 = 4$

T [4] is free, so insert key 26 at this place.

3. Insert 37.

$h_1(37) = 37 \bmod 11 = 4$

$h_2(37) = 37 \bmod 9 = 1$

Collision

$h(37, 0) = (4 + 0 \times 1) \bmod 11 = 4 \bmod 11 = 4$

T [4] is not free, the next probe sequence is computed as -

$h(37, 1) = (4 + 1 \times 1) \bmod 11 = 5 \bmod 11 = 5$

T [5] is free, so insert key 37 at this place.

$h(26,0) = (26 \bmod 11 + 0 \times \_\_\_) \bmod 11$

$= 4 \bmod 11 = 4$

$h(37,0) = (37 \bmod 11 + 0 \times \_\_\_) \bmod 11$

$= 4 \bmod 11 = 4$

$h(37,1) = (37 \bmod 11) + 1 \times$

$= (4 + (37 \bmod 9)) \bmod 11$

$= (4 + 1) \bmod 11$

$= 5 \bmod 11 = 5$

4. Insert 59.

$h_1 (59) = 59 \bmod 11 = 4$

$h_2 (59) = 59 \bmod 9 = 5$

$h (59, 0) = (4 + 0 \times 5) \bmod 11 = 4 \bmod 11 = 4$ Collision

Since, T [4] is not free, the next probe sequence is computed as -

$h (59, 1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$

T [9] is free, so insert key 59 at this place.

5. Insert 21.

$h_1 (21) = 21 \bmod 11 = 10$

$h_2 (21) = 21 \bmod 9 = 3$

$h (21, 0) = (10 + 0 \times 3) \bmod 11 = 10 \bmod 11 = 10$ Collision

T [10] is not free, the next probe sequence is computed as -

$h (21, 1) = (10 + 1 \times 3) \bmod 11 = 13 \bmod 11 = 2$

T [2] is free, so insert key 21 at this place.

6. Insert 65.

$h_1 (65) = 65 \bmod 11 = 10$

$h_2 (65) = 65 \bmod 9 = 2$

$h (65, 0) = (10 + 0 \times 2) \bmod 11 = 10 \bmod 11 = 10$ Collision

T [10] is not free, the next probe sequence is computed as-

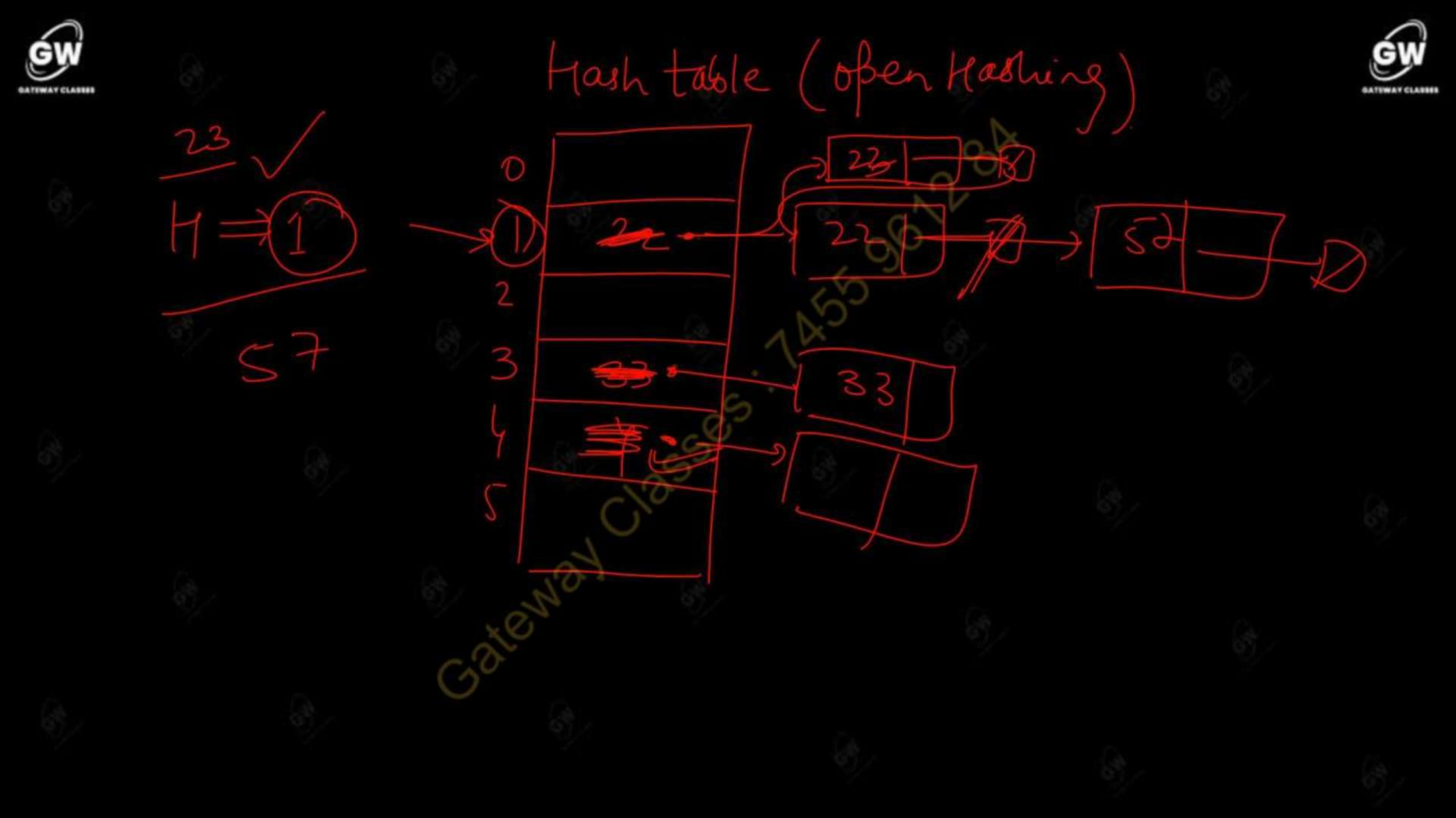
$h (65, 1) = (10 + 1 \times 2) \bmod 11 = 12 \bmod 11 = 1$

T [1] *is* free, so insert key 65 at this place. Thus, after insertion of all keys the final hash table is:

0	1	2	3	4	5	6	7	8	9	10
I	*65*	*21*	*I*	*26*	*37*	*I*	*I*	*I*	*59*	*76*

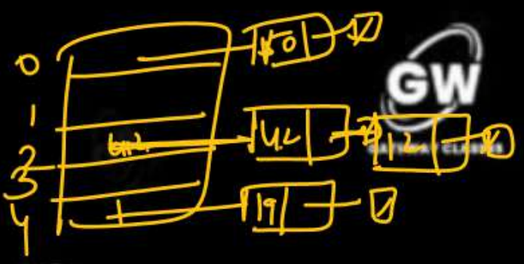
HASHING WITH CHAINING

- This method maintains the chain of elements which have same hash address.
- We can take the hash table as an array of pointers.
- Size of hash table can be number of records.
- Here pointer will point to one linked list and the elements which have same hash address will maintained in the linked list.
- We can maintain the linked list in sorted order and each element *of* linked list will contain the whole record with key as shown in the next figure.



Example of Chaining (Open Hashing)

- Let us take an example and consider the following keys – 42 , 19 , 10 , 12
- Our hash function is – h (K) = K mod 5
- i.e. hash table will be of 0 to n - 1 i.e. from 0 to 4 -
- h (42) = 2
- h (19) = 4
- h (10) = 0
- h (12) = 2



$h(K) = K \bmod 5 = 42 \bmod 5$
$h(19) = K \bmod 5 = 19 \bmod 5 = 2$
$h(10) = K \bmod 5 = 10 \bmod 5 = 0$
$h(12) = K \bmod 5 = 12 \bmod 5 = 2$

$h(42) = 42 \bmod 5 = 2$
$h(19) = 19 \bmod 5 = 4$
$h(10) = 10 \bmod 5 = 0$
$h(12) = 2$

- For inserting one element, first we have to get the hash vales through hash function which will map in the hash table, then that element will be inserted in the linked list.
- Searching a key, is also same, first we will get the hash key value in hash table through hash function, then we will search the element in corresponding linked list.
- Deletion a key, contains first search operation then came as delete operation of linked list.

- The main advantage of this hashing is saving of memory space and perfect Colleton resolution.
- Here hash table contain only pointers which point to the linked list, containing space because of empty hash table and the back records, So there is no wastage of memory space because of empty hash table and the hash table size can be taken as number of records.
- Hash table space is also minimized because records are stored in linked list.
- So memory space can be allocated for keeping records at the which at the time of need.
- From collision resolution point of view, it gives perfect solution.
- The elements which have same hash address will be in same linked list.
- So we should use good hash function which distributes elements on different hash address.
- Basically linked Bets will be small, so searching will be fast.