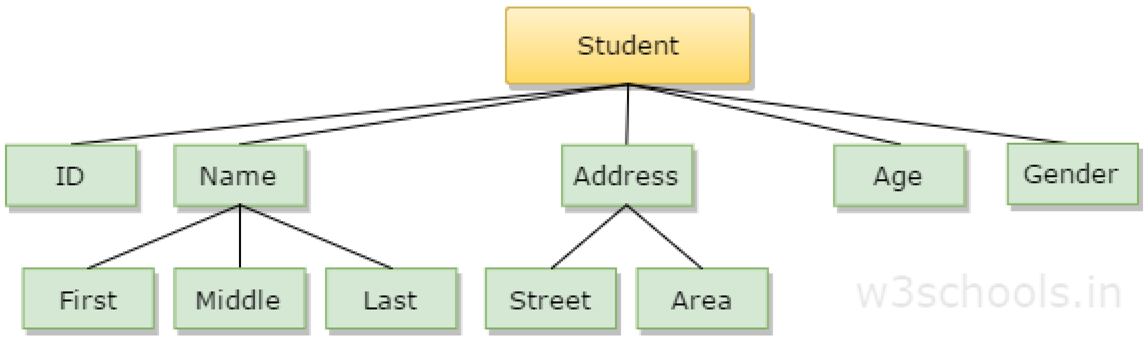
Introduction: Data Structures

A data structure is a special way of organizing and storing data in a computer so that it can be used efficiently. Array, LinkedList, Stack, Queue, Tree, Graph etc are all data structures that stores the data in a special way so that we can access and use the data efficiently.

In the modern world, Data and its information is an essential part, and various implementations are being made to store in different ways. Data are just a collection of facts and figures, or you can say data are values or a set of values that are in a particular format.

A data item refers to a single set of values. Data items are then further categorized into sub-items which are the group of items which are not being called a plain elementary form of items. Let us take an example where the name of the student may be divided into three sub-items namely: first name, middle name and last name. But the ID that is assigned to a student would normally be considered as a single item.



What is DS:

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure. The variety of a specific data model depends on the two factors -

Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real world object.

Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Why is Data Structure important?

- Data structure is important because it is used in almost every program or software system.
- It helps to write efficient code, structures the code and solve problems.
- Data can be maintained more easily by encouraging a better design or implementation.
- Data structure is just a container for the data that is used to store, manipulate and arrange.
- It can be processed by algorithms.

  For example, while using a shopping website like Flipkart or Amazon, the users know their last orders and can track them. The orders are stored in a database as records.
  However, when the program needs them so that it can pass the data somewhere else (such as to a warehouse) or display it to the user, it loads the data in some form of data structure.

Classification (Primitive and Non-Primitive)

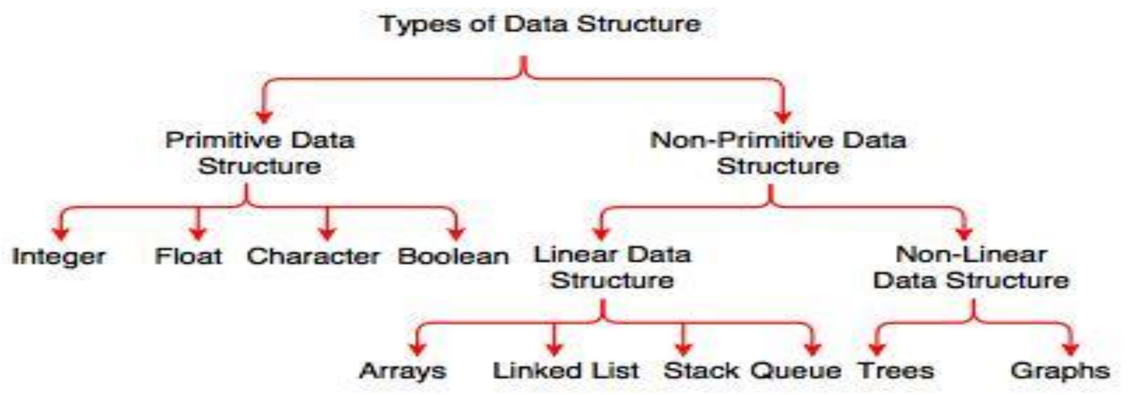


Fig. Types of Data Structure

Data Type:

A Data type defines a domain of allowed values and the operations that can be performed on those values. Forexample in C, int datatype can take values in a range and operations that can be performed are addation substraction, ultiplication, division, bitwise operations etc.

Primitive Data Type

- Primitive data types are the data types available in most of the programming languages.
- These data types are used to represent single value.
- It is a basic data type available in most of the programming language.

Data type	Description
Integer	Used to represent a number without decimal point.
Float	Used to represent a number with decimal point.
Character	Used to represent single character.
Boolean	Used to represent logical values either true or false.

Non-Primitive Data Type

- Data type derived from primary data types are known as Non-Primitive data types.
- Non-Primitive data types are used to store group of values.

DATA STRUCTURES OPERATIONS

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely in the Frequency with which specific operations are performed.

The following four operations play a major role in this text:

· **Traversing:** accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “visiting” the record.)

· **Searching:** Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.

· **Inserting:** Adding a new node/record to the structure.

· **Deleting:** Removing a node/record from the structure.

The data structure can be subdivided into major types:

- Linear Data Structure & Non-linear Data Structure
- Static and Dynamic Data Structure

A data structure is said to be linear if its elements combine to form any specific order. There are two techniques of representing such linear structure within memory.

Linear Data Structure

a data structure is linear if all the elements are arraigned in a linear order. In a linear data structures, each element has only one successor and only one predecessor.

The only exceptions are the first and the last elements; first element does not have a predecessor and last element does not have a successor.

The examples of the linear data structure are:

- Arrays
- Queues
- Stacks
- Linked lists

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

Non-linear Data Structure

In non linear data structures there is no linear order in the arrangement of the elements.

Examples of Non-Linear Data Structures are listed below:

- Graphs
- The family of trees and table of contents

Tree: In this case, data often contain a hierarchical relationship among various elements. The data structure that reflects this relationship is termed as a rooted tree graph or a tree.

Graph: In this case, data sometimes hold a relationship between the pairs of elements which is not necessarily following the hierarchical structure. Such a data structure is termed as a Graph.

Static Data Structures

In static data structures the memory is allocated at compilation time only, therefore the maximum size is fixed and it can’t be changed at run time. Static data structures allow fast access to elements but insertion and deletion is expensive. Array is the best example for static data structures

Dynamic Data Structures

In dynamic data structures the memory is allocated at runtime therefore these data structures has flexible size dynamic data structures allow fast insertion and deletion of elements but access to elements slow. Linked list is an example of dynamic data structures.

Algorithms:

An algorithm is a procedure having well defined steps for solving a particular problem. The data stored in the data structures is manipulated by using different algorithms, so the study of data structures includes the study of algorithms. Some of the common approaches designed are:-

Greedy algorithm: a greedy algorithm works by taking a descion that appears best at the moment, without thinking about the future. The descion once taken is never reconsidered. this means that a local optimum is chosen at every step in hope of getting a global optimum at the end

It is necessary that the greedy algorithm always produce optimum solutions

Divide and conquer algorithm: solves a problem by dividing it into sub problems the solution of these smaller problems are then combined to get the solution of the given problem..

The example for divide and conquer algorthiim are merge sort, quick sort, binary search

Backtracking algorithm: in some problems we have several options, where any one might lead to solutions.

We will take an option and try, and if we do not reach the solution, we will undo our action and select another one. The steps are retraced if we do not reach the solution, it is a trial and error process the example of backtracking is eight queens problem.

Randomized algorithm:

In a randomized algorithm, random numbers are used to make some decisions. The running time of such algorithm depends on the input as well as the random numbers that are generated.

The running time may be varied from same input data. An example of randomized algorithm is a version of quick sort , where a random number is choose as the pivot

Arrays:

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like **char, int or float.** The element of the array shares the same variable name but each element has a different index number known as subscript.

Consider a situation when we want to store and display the age of 100 employees. We can take an array variable **age** of type **int**.

The size of this array variable is 100 so it is capable of storing 100 integers values. The individual element of this array are

Age[0], age[1], age[2], age[3],………………age[98],age[99]

Here subscript starts with zero, so the age[0] is the first element, age[1] is the second element of the array

Array can be single dimensional or multidimensional. The number of subscripts determine the dimensions of array. A one dimension array has one subscript; two dimensional array has two subscript and so on, the two dimensional array are known as matrix.

One dimensional array

Like other simple variables, array should also be declared before they are used in the program. The syntax for declaration of an array is

Data_type array_name[size];

Here array_name denotes the name of the array and it can be any valid C identifier, data_type is the data type of the elements of array. The size of the array specifies the number of elements that can be stored in the array. It may be a positive integer constant or constant integer expression. Here are some examples of array decleration.

int age[100];

float salary[15];

char grade[20];

Here **age** is integer type array, which can store 100 elements of integer type.

The array salary is float type array of size 15, can hold float value

Character type array of size 20, can hold characters

Age[0], age[1], age[2]……age[99]

Salary[0], salary[1], salary[2]………..salary[15]

Grade[0], grade[1], grade[3]…………grade[19]

Two dimensional arrays:

The syntax of decleration of a 2-D array is similar to that of 1-D arrays, but here we have subscripts

Data_ type array_name[rowsize] [columnsize];

Here *arr* is a 2-D array with 4 rows and 5 columns. The individual elements of this array can be accessed by applying two subscripts, where the first subscript denotes the row number and the second subscript denotes the column number.

The starting element in the array are rowsize* column size for

Ex: int arr[4][5]

Here arr is a 2-D array with 4 rows and 5 columns. The individual elements of this array can be accessed by applying two subscripts,

Where the first subscript denotes the row number and the second subscript denotes the column number

The starting element of this array id arr[0][0]

The last element is arr[3][4]

The total number of elements in this array is 4*5=20

Linked list (only introduction part detailed concept will be explained in coming units)

List of collection of similar type of elements. There are two ways of maintaining the list in the memory

1. First way is to store the element of the list in the array. But arrays having some restrictions and disadvantages.
2. The second way of maintaining a list in memory is through linked list

Single linked list:

A single linked list made up of nodes where each node has two parts.

- The first one is the info part that contains the actual data of the list

- The second one is link part that points to the next node of the list or we can say that it contains the address of the next node

INFO	LINK

STACKS AN QUEUES

- In linked list and arrays, insertion and deletion can be performed at any place of the list
- There can be situation when there is a need of data structure in which operations are allowed only on the ends of the list and not in the middle.
- Stacks and ques are the data structures which fulfills these requirements.
- Stack is a linear list in which insertion and deletion are allowed only at one end
- While queue is a linear list in which insertion performed on one end deletion is performed on the other end.

Pointers:

A pointer is a variable that stores the memory address, Like all other variable it also has a name, has to be declared and occupied space in memory.

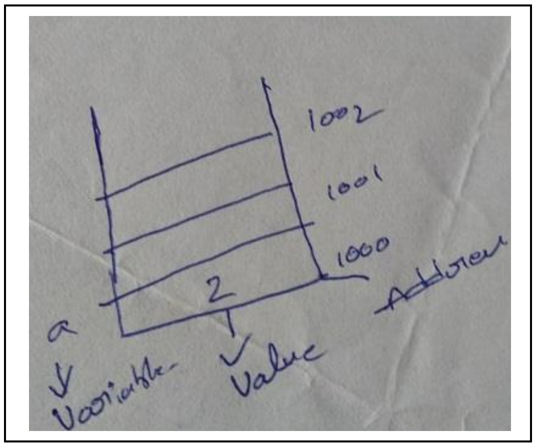
It is called pointers because it points to a particular location in memory by storing the address of the location.

The use of pointers makes the code more efficient and compact.

Ex: if we declare any variable

Data type **variable name**

$$\begin{tikzcd}
\textbf{Int} \arrow[d] & \textbf{‘A’} \arrow[r] & \boxed{\text{Which holds the integer value}} \\
\boxed{\text{Fo ex: 2 bytes in the memory}} & &
\end{tikzcd}$$



- Here variable is declared and some memory is allocated to the give variable
- If we initialize any value, the value will store in particular address.

✓ **Pointers is a variable which holds address of another variable**

✓ **Pointer represent** ‘*’ → value operator

‘&’ → address operator

Declaring pointers:

Data type *Variable name

(For storing address of a variable we have to use pointers concept)

Int a=2;

Int *P;

(to declare pointer variable we have to add **‘*’ prefix**)

P=&A

(P= pointers)

(&A= address)

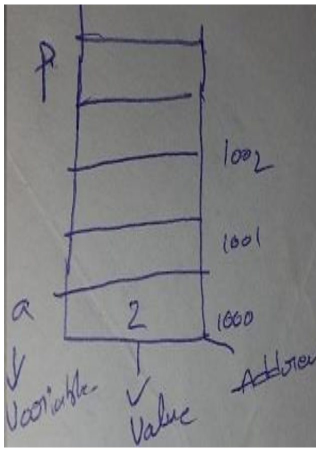
If we print

P→ &A=1000

*P→ value at address (1000)

=2

*P can also be written as *(&a)



Example: Sum of two numbers

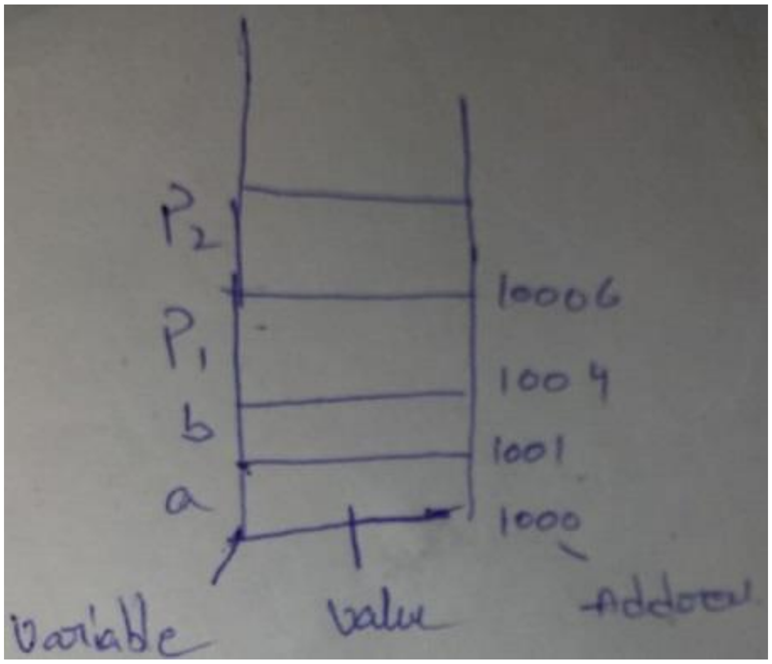
Int A,B,sum=0

Int *P1,*P2;

A=5; B=6;

P1=&A;

P2=&B;



Sum = *P1+*P2

= *(1000)+*(1002)

=5+6

Sum= 11

Representation of Linear Arrays in Memory:

- The elements of the array are stored in the contiguous memory location suppose we have an array of type **int.**

  int arr[5] = {1,2,3,4,5};

  suppose it is stored in memory at address 5000

Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]
1	2	3	4	5
5000	5004	5008	5012	5016

- Here 5000 is the address of the first element
- And since the element of type **int** takes 4 bytes, address of the next element is 5004 and so on.
- The address of the first element is also known as base address of the array
- It is clear that the elements of the array are stored sequentially in memory one after another.

For example:

Int arr[5] = {5,10,15,20,25}

Here **arr** is an array that has 5 elements each of type **int.**

Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]
5	10	15	20	25
2000	2004	2008	2012	2016

- We can get the address of an element of array by applying '**&**' operator in front of subscript variable name.
- Hence **&arr[0]** gives address of 0th element.
- &arr[1] gives the address of first element and so on. Since the array subscript start from 0, we refer the first element of array as 0th element and so on.

The name of the array is a constant pointer, according to the pointer arithmetic when an integer is added to a pointer then we get the address of the next element of the same type.

1. **Write a program to print the value and address of the element of the array**

```
# include<stdio.h>
Main()
{
        Int arr[5] ={5,10,15,20,25}
        Int I;
        For (i=0;i<5;i++)
{
        Printf("value of array[%d]=%d\t"I,arr[i]);
        Printf("address of arr[%d]= %p/n",I,arr[i]);
}
}
```

arr → points to 0th element → &arr[0] → 2000

arr+1 →points to 1st element → &arr[1] →2004

arr+2 →points to 2nd element → &arr[2] →2008

arr+3 →points to 3rd element → &arr[3] →2012

arr+4 →points to 4th element → &arr[4] →2016

- In general we can write as the pointer expression (arr+1) denotes the same address as &arr[i]
- No if we dereference arr, then we get the 0th element of array, i.e expression *arr or *(arr+0) represents the 0th.
- arr → points to 0th element → &arr[0] → 5
- arr+1 →points to 1st element → &arr[1] → 10
- arr+2 →points to 2nd element → &arr[2] → 15
- arr+3 →points to 3rd element → &arr[3] → 20
- arr+4 →points to 4th element → &arr[4] → 25

2. Program to print the value and address of element of an array using pointers.

```
# include<stdio.h>
Main()
{
        Int arr[5] ={5,10,15,20,25};
        For (i=0;i<5;i++)
{
        Printf("value of array[%d]=%d\t"i,(arr+i));
        Printf("address of arr[%d]= %p/n",i,arr+i));
```

}
}

Dynamic memory allocation:

It is important to develop algorithm based applications like **stacks and queues.**

Static Memory or fixed size:

- Declaring a primative variable, **int a;**
- Memory allocated at runtime only
- We call it as static means memory size is fixed

  Ex: int arr(100)

  Here we cannot store more than 100, so it is called fixed size. These are called static memory allocation.

Dynamic memory allocation:

- Size of the array may be increased or decreased based on the elements storing and deleting.
- To allocate the memory dynamically predefined function is used [**Stdlib.h**]library
- **Stdlib.h provides 4 important functions to allocate or deallocate functions.**
- **Pointers conceot is used**
  1. **Malloc:** used to allocate the memory to structures
  2. **Calloc :** used to allocate the memory to arrays
  3. **Realloc :** used to increase or decrease size of the array
  4. **Free:** used to delete the memory

1. **Malloc:** used to allocate the memory to structures.

   Void ptr = malloc(Size_type size);

   Void ptr: return type= generic pointer

   Size_t: argument= unsigned= positive integer value

   Size: variable

   - Whenever we allocate memory we should pass only positive integer value
   - On success : it returns base address of the memory block

- Onfailure: it returns the null pointer

Memory allocation using pointers

Here we don't know the exact size if it is 16 bit **int** will be 2 bytes or if it is 32 bit 4 bytes

So we use struct emp

```
Struct emp
{
Int eno
Char ename[20]
Float esal;
};
Void ptr=(struct emp*)malloc(size of(struct epm));
If(ptr==null)
{
Printf("out of memory error\n");
}
Else
{
Printf("enter employee details");
Scanf("%d,%s,%p; ptr→eno; ptr→ ename;ptr→esal");
}
```

Calloc()

- Memory allocation to array
- Void *calloc(size_t n, size_t size) ['n' is array size, size:size of the element in array]
- On success it returns address of the memory block
- On failure ; it returns null pointer
- Using calloc function we can allocate memory dynamically
- We cannot increase or decrease the size of the array
- So calloc is failed
- Calloc taing the realloc function to increase or decrease the size of the array

Realloc();

Increase or decrease the size of the array

Void* realloc(void *ptr, size_t size)

Generic pointer it can access any array size

Polynomials and sparse matrix

Sparse matrix:

Sparse matrixes are those matrixes which have the majority of elements equal to zero.

Ex:

0	0	0
1	0	0
0	1	0

- **WHY WE USE SPARSE MATRIX INSTEAD OF SIMPLE MATRIX**

  ***STORAGE**: There are less non zero elements than zero elements.
  so less memory can be used to store the elements.

  ***Computing Time:** computing time can be saved by traversing only non-zero elements.

  *** storing non-zero elements with triples: (**Row, column, value**)**

- **Sparse matrix can be represented in following ways.**
  1. **Array representation**
  2. **Linked representation**

1. **Array representation :**

   Two dimensional array is used to represent a sparse matrix, in which there are three rows named as

➔ Row: index of the row,

➔ column: index of the column

➔ Value: value of the non zero element located at index (row & column)

Example: array representation of sparse matrix:

0	0	4	0	5
0	0	0	3	6
0	0	0	2	0
0	2	3	0	0

4*5

ROW	**0**	**0**	**1**	**1**	**2**	**3**	**3**
COLUMN	**2**	**4**	**3**	**4**	**3**	**1**	**2**
VALUE	**4**	**5**	**3**	**6**	**2**	**2**	**3**

2. LINKED REPRESENTATION:

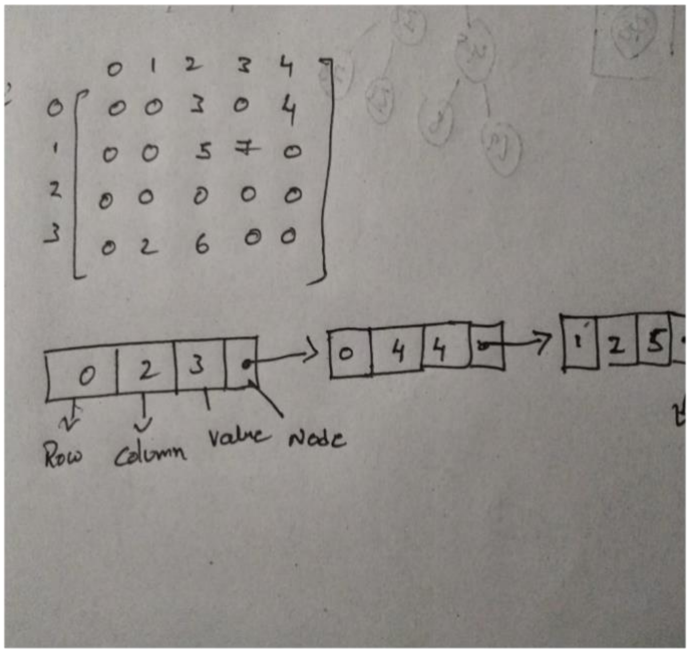
In linked representation here we have four fields

➔ROW: index of row, where non zero element is located

➔ column: index of the column

➔ Value: value of non zero element located at index of (row, column)

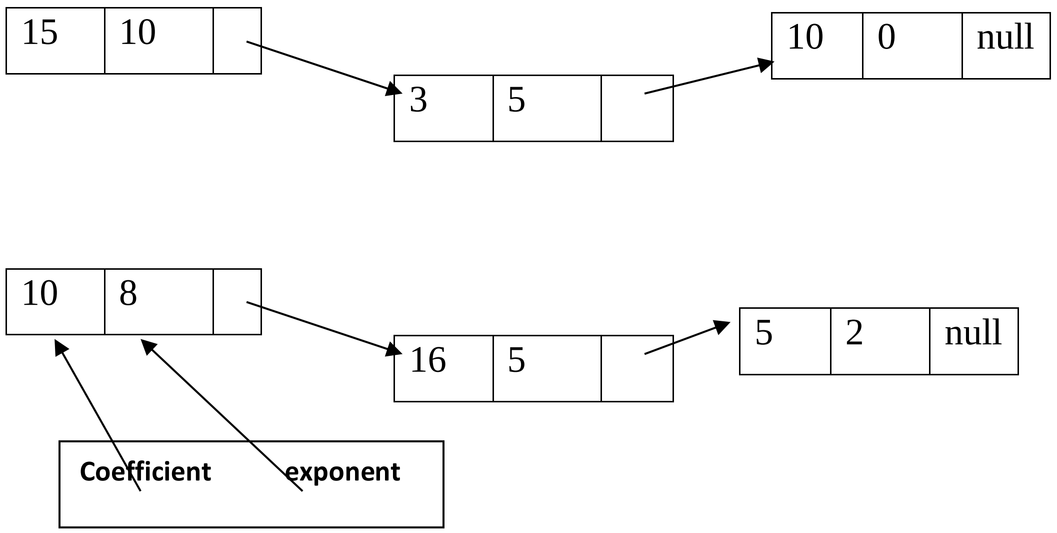
➔ Next node : address of the next node



- **Polynomial representation using linked list**

P(x) =15x^10+3x^5+10

Q(x)=10x^8+16x^5+5x^2



Example: polynomial addition using linked list.

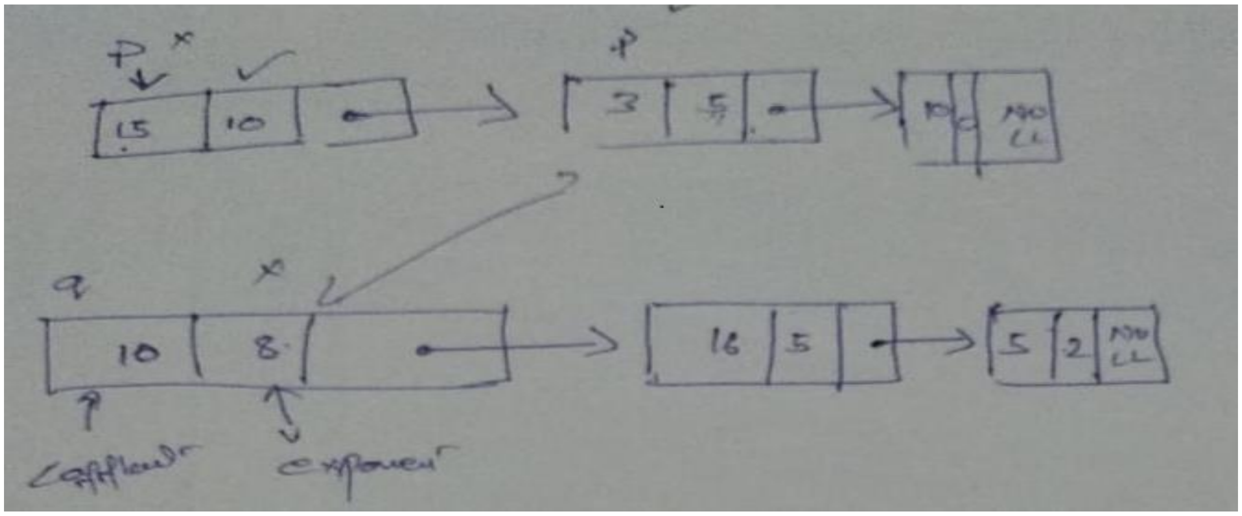
P(x) +Q(x) = 15x^10+10x^8+19x^5+5x^2+10

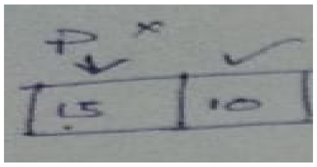
Here we written like 3x^5+16x^5 both elements are same so we write it as

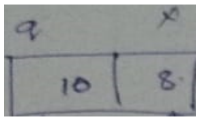
3x^5+16x^5 =19x^5

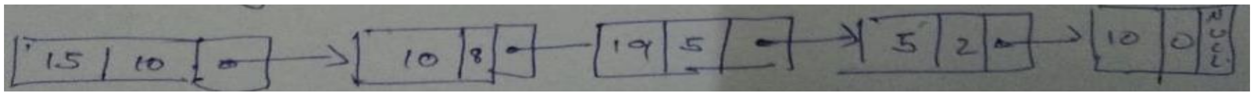
10x^0=10

5x^2=5x^2



First compare **P & Q, 10** is highest so we are copying 

Now **'p'** shifted to next node comparing with the exponents **5 & 8** so we are copying  because exponent **'8'** is highest.

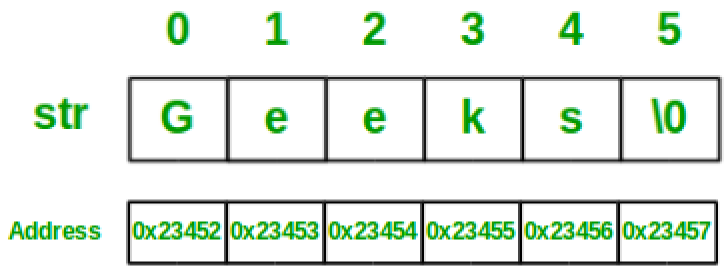


String Data Structure:

Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character '\0'.

Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string in **C programming** language.

char str_name[size];



Srting operations and related operations:

- Length
- Substring (string, initial, length) {input augment}
- Concatenation (a series of interconnected things)
- Indexing (text, pattern) {input augment}{searching the sub string from the main string}

Example:

Str1 = "hyderabad"

Srt2 ="university"

- ➔ **Length(str1)=9**
- ➔ **Length (str2)=10**

- **Substring:** Substring (string, initial, length)

  Substring(str2,4,3) = university

  Substring(str2,ver)

  Substring(str1,6,100)= Hyderabad

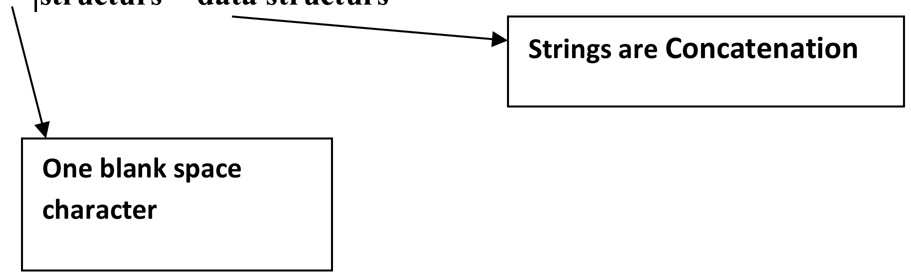
  Substring(str1,abad) {here traversal is 100 but here the length of the elements is **abad, so we write it as "ABAD"**}

- **Concatenation (a series of interconnected things)**

  Example:

  Str1 |" "| str2= "Hyderabad university"

  data|" "|structurs="data structurs"



- **INDEXING(TEXT, PATTERN) {** searching the sub string from the main string}

  Str1=Hyderabad

  Str2= university

  Index(str1, "erabad")=4

  Index(str2,"iver")=3

  Index (str2,"india")=0

  →Searching the "erabad from string hyderabad"

  →An index (str2,"india")=0 here searching india so no (india) is available so it is "0".

  ➔ If searching is unsuccessful it will written as zero.

PATTERN MATCHING ALGORTHIM:

Pattern matching algorithm is an act of checking a given sequence of token for the presence of the constituents of some pattern.

- Strings are sequence of characters
- Strings are more often used than numbers

- Some of important operations on the string are searching a word, find and replace operations etc.
- String matching is most important problem
- String matching consists of searching a query string or pattern.
  EX: “P” in a given text “T”
- Generally size of the pattern to be searched is smaller than the given text
- There are several applications for the string matching some of these are
  Text editor, search engine’s, biological applications
- Since matching algorithms are used extensively, these should be efficient in terms of time and space.
- Let P[1..M] is the pattern to be searched and its size is “M”
- Assume that pattern occurs in “T” at position the output of the matching algorithm will be the integer I {where 1<=i<=n-m}if there are multiple occurrence of the pattern in the text, then some times.

  Let pattern “P”=CAT

  TEXT=ABABNACATM
- Then there is a match with the shift in the text “T”

1	2	3	4	5	6	7	8	9	10
A	B	A	B	N	A	**C**	**A**	**T**	M

PATTERN P= CAT

BRUTE FORCE STRING MATCHING ALGORTHIM

this is a simple and one in which we can compare a given pattern “p” with each of the surroundings of the text “T”, moving from left to right until match is found.

- Let Si is the substring of “T” beginning at the ith postion and where length is same as pattern “P”.
- We compaer “P” character by character with the first substring s1
- If all the corresponding characters are same then the pattern “p” appears in ‘T’ at shift”1”
- If the same of the character of s1 not matched with the corresponding characters of “p”,then we try for next substring s2
- This procedure continues till the input text exhausted
- In this algorithm we have to compare “p” with n-m+1 substring of T

Example:

Let P=aba

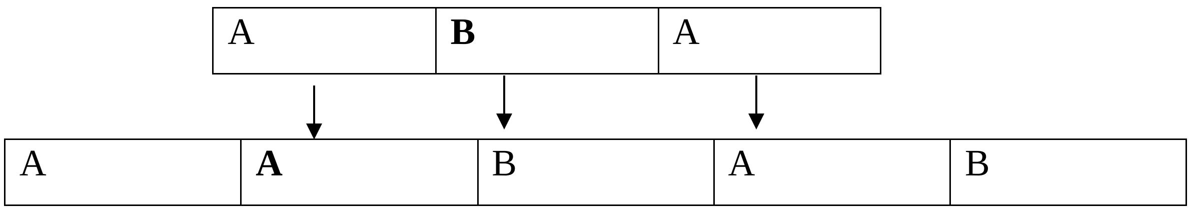
T=aabab

- Comparing “P” with 1st substring of “T”

A	**B**	A

A	**A**	B	A	B

- Mismatching second character at “T”
- Compare ‘p’ with 2nd



Since the corresponding characters are same there is a match at shift “1”

Algorithm for brute force string matching algorithm:

Let p[o,m] is a given pattern and T[o,n]is the next

```
1. I=1 [substring 1]
2. Repete step 3 to 5 while i<=n-m+1 do
3. For j=1 to m [for each character of P]
4. For j=1 to m[for each character of p]
   If p[j]!=T[i+j-1] then
   Go to step 5
5. Print "pattern founds at shift i"
6. I=i+1
7. Exit
```

SORTING

- **Sorting:** sorting means rearranging the elements into either ascending or descending order within the array.
- Sorting is one of the most common data structure processing applications.
- We use to arrange in ascending order or descending order.

Example: 7,39,8,36,24

We have to arrange this in ascending order or in descending order

Ascending order: 7,8,24,3,39

Descending order: 39,36,24,8,7

$$\begin{tikzcd}
 & & \textbf{SORTING} \arrow[dl] \arrow[dr] & \\
 & \textbf{INTERNAL} \arrow[d] & & \textbf{EXTERNAL} \\
 & {} \arrow[dl] \arrow[d] \arrow[dr] & & \textbf{*merge} \\
\textbf{INSERCTION} & \textbf{SELECTION} & \textbf{EXCHANGE} &
\end{tikzcd}$$

INSERCTION	**SELECTION**	**EXCHANGE**
***straight insertion**	***selection**	***bubble**
***shell**	***heap**	

✓ **Internal operations performing:**

We perform

- **Insertion**
- **Selection**
- **Exchange**

✓ **Exchange operations performing:**

Taking two arrays and merging it called as external operations

Ex: **array1 + array2**

{ } + { }

✓ **Basic terminology for sorting**

- **Sort stability:**

  Sort stability is an attribute of a sort indicating that data with equal keys maintains their relative input order in the output.

  {The sort order identifies the sequence of sorted data, ascending or descending}

- **Sort efficiency :**

  Sort efficiency is the measure of relative efficiency of a sort.

Unsorted data			**Stable sort**	
365	blue		119	purple
212	**green**	→	**212**	**green**
876	white	↗	**212**	**yellow**
212	**yellow**		365	blue
119	purple		876	white

- Here we are arrainging two elements having same number

  212 green

  212 yellow
- So we write its input order as its output order

✓ **Sort Pass**

- During the sort pass the data traversed many time.
- Each traversed of the data is referred s a sort pass
- While the traversing process is

a) 3	7	6	P1
b) 3	6	7	p2

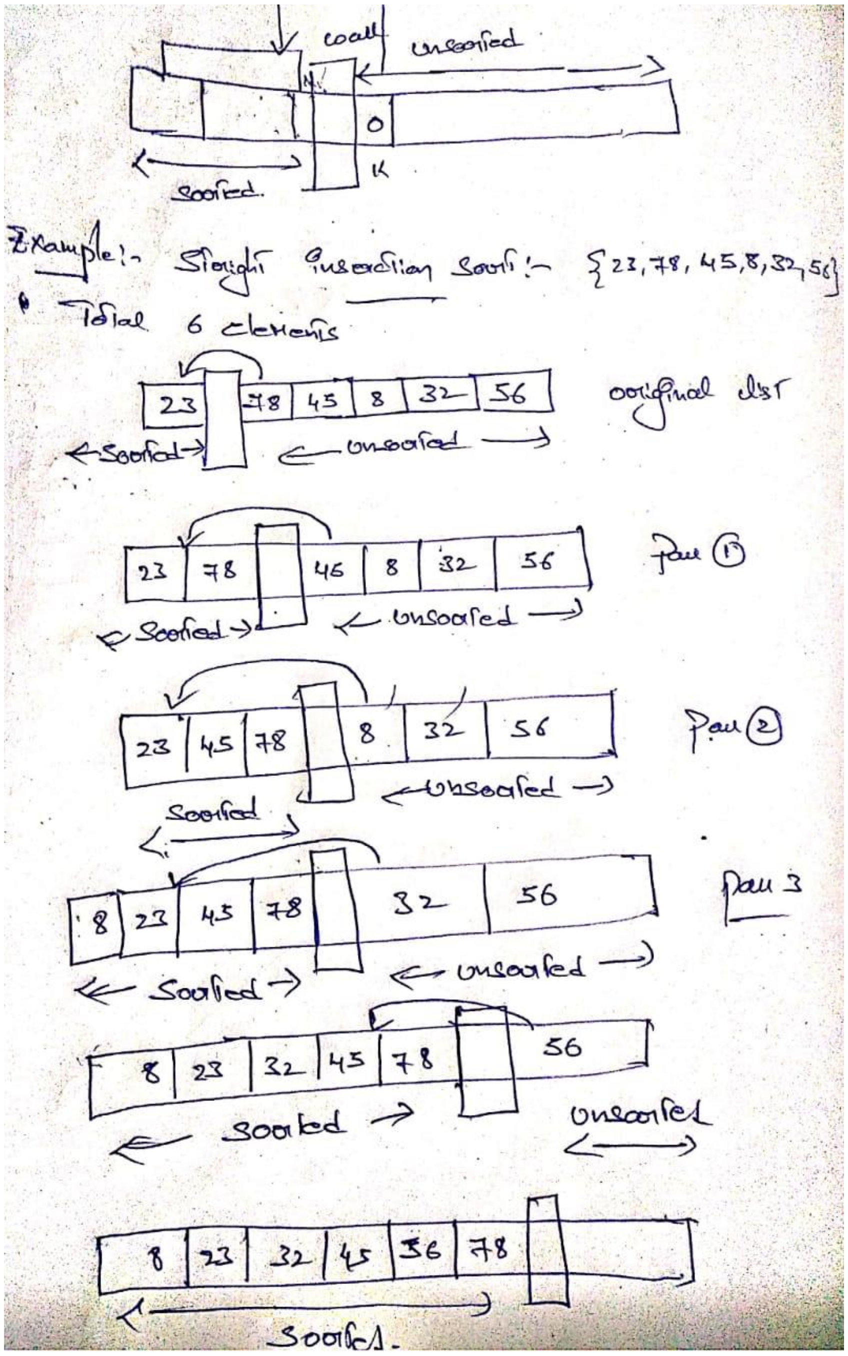
INSERCTION SORT:

In each pass of insertion sort one or more pieces of data are inserted into their correct location in an ordered list.

- Let us take any one of the list which is unsorted
- Let us see two inserction sorts
  1. Straight inserction sort
  2. Shell sort

- What we are going to do in this……………

- The list at any moment can be divided into sorted and unsorted sublist
- What ever the list we are taking is divided into **sorted and unsorted**
- First element of the unsorted sublist is inserted into sorted sublist
- Based on greater than or less than checking the value we have to insert
- After pass 1 the element is placed inbetween **23 & 78**



Algorthim (insertion sort)

- pre list contain at least one element
- last is an index to last element in the list
- post list has been rearrainged

```
1. step current to 1
2. loop/until last element sorted
       a. more current elements to hold
       b. set walker to current-1
       c. loop(walker>=0 and hold key < walker key
              1.  move walker element right one element
              2. decrement walker
3. 3nd loop
4. Move hold to walker+1 element
```

Shell Sort

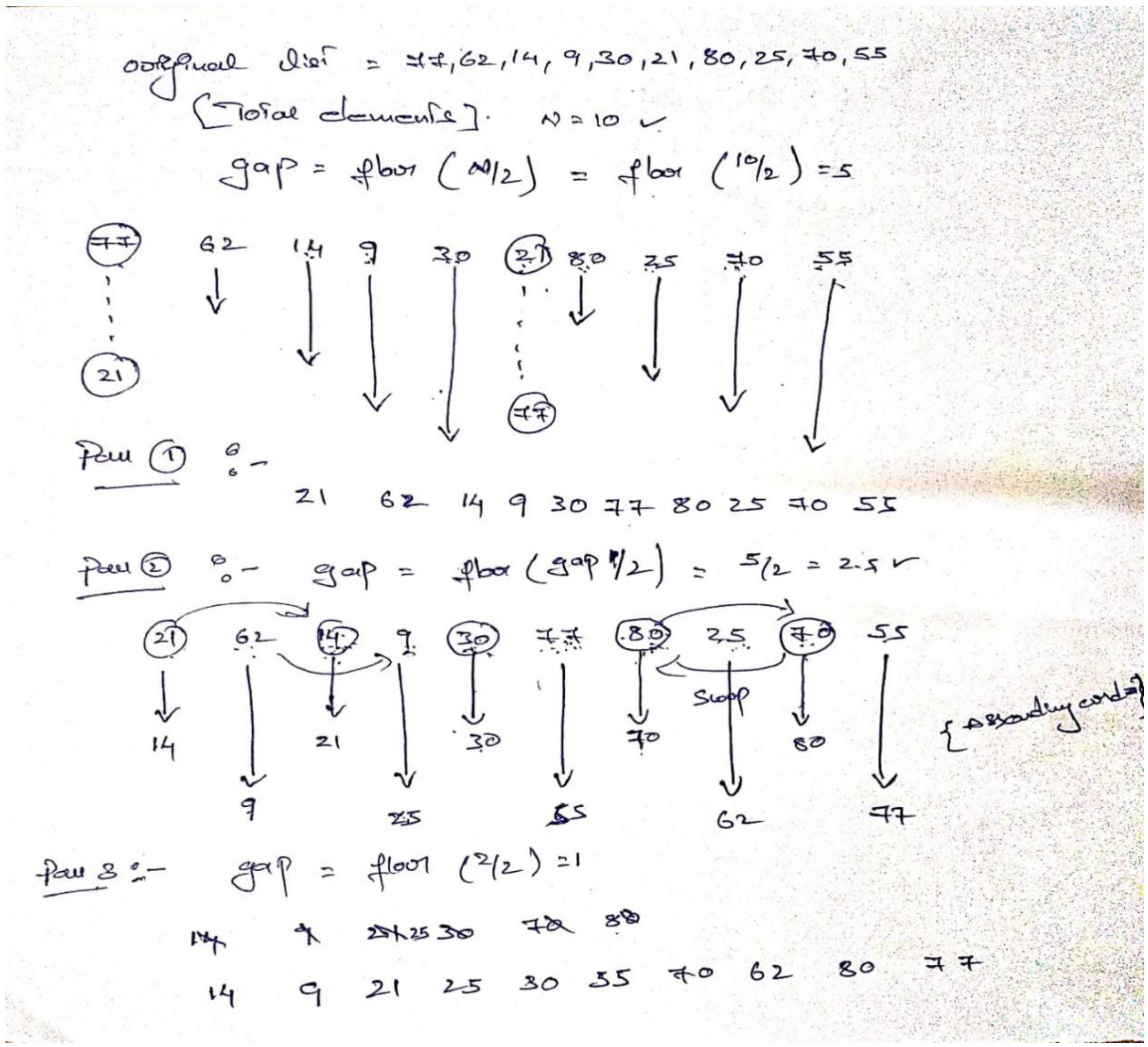
- Shell sort is an improved version of the straight insertion sort in which dinishing partision are used to store the data.
- Compare the elements that are distant apart rather than adjacent
- We have to compare the elements distant a part of adjacent elements
- We have to check the elements distant apart
- If there are "N"elements then we start a value GAP<N
- **FORMULA : GAP=FLOR(N/2)**
- **WHERE N= no.of elements in an array**

Shell sort example:

- In each pass we keep reducing the value of gap till we reach the last pass when gap is "1"

- In the last pass shell short is like inserction short
- **GAP=FLOR(N/2)**
- **GAP1=FLOR(GAP/2)**
- **GAP2=FLOR(GAP1/2)**

Original list= 77 62 14 9 30 21 80 25 70 55 [Total elements "N"=10]



Only for reference please follow textbook for more explanation

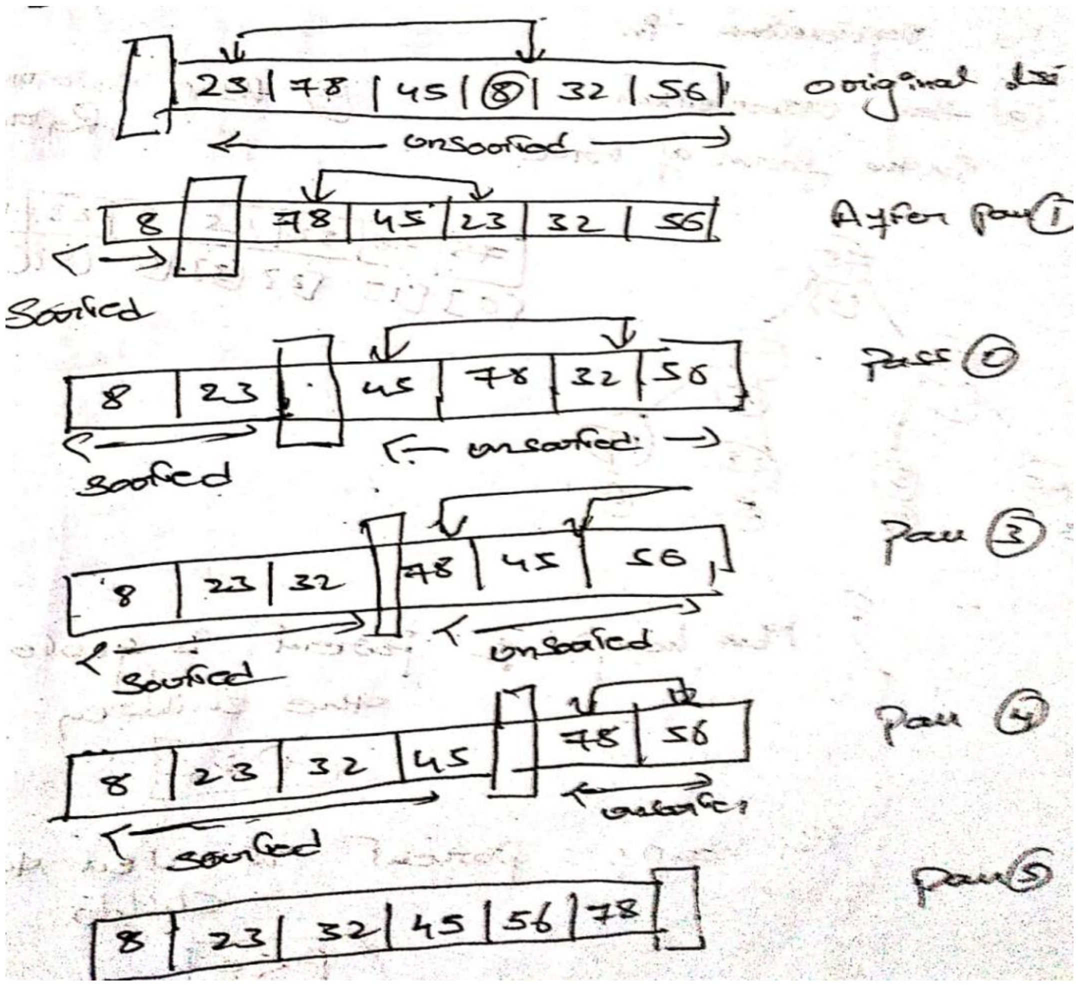
Algorithm:

SELECTION SORT

- In each pass of the selection sort the smallest element is selected from the unsorted sublist.
- Exchanging the elements at the beginning of the unsorted list
- Two types of classic selection sorts

  *straight selection sort

  *heap sort

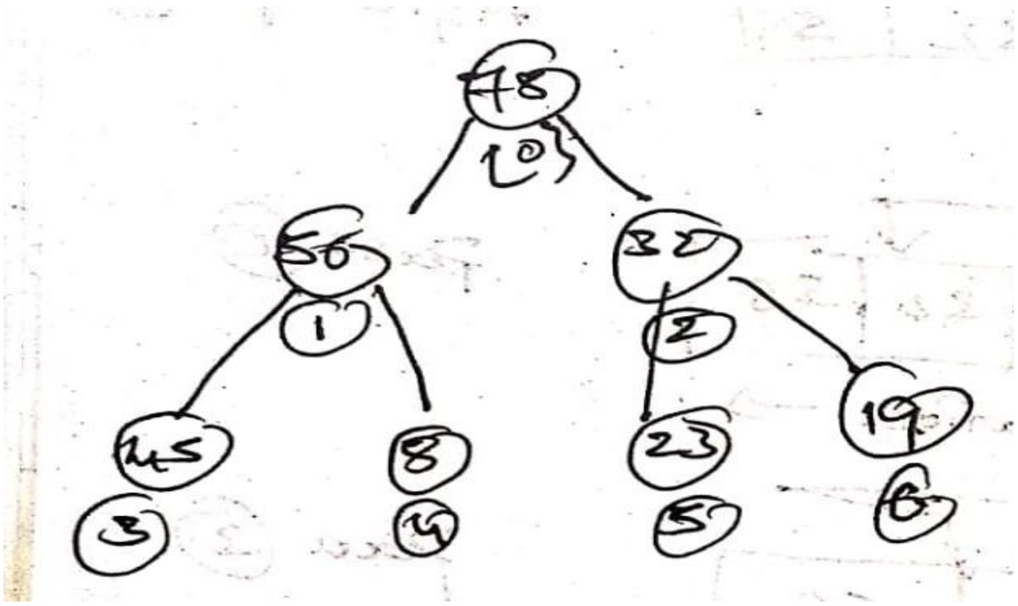


Now it is in sorted order..

Heap Sort

- **The heap sort is an improved version of the straight selection sort in which the largest element (the root) is selected and exchanged with the last element in the unsorted list.**

Heap representation in the form of tree



Heap in array representation

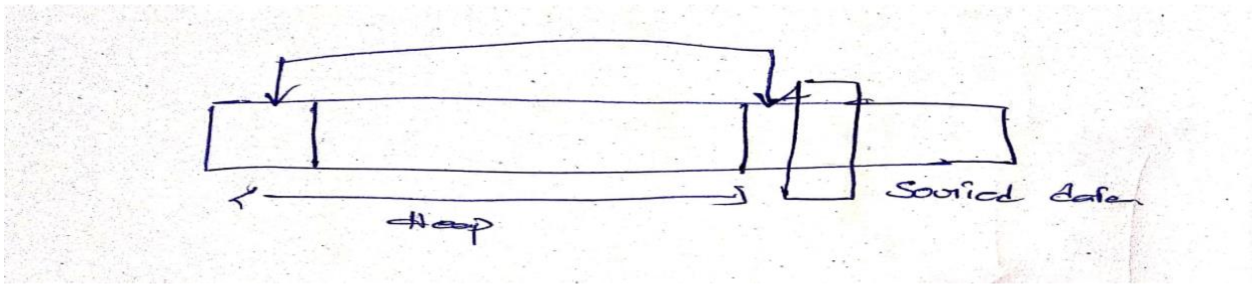
78	**56**	**32**	**45**	**8**	**23**	**9**

[0] **[1]** **[2]** **[3]** **[4]** **[5]** **[6]**

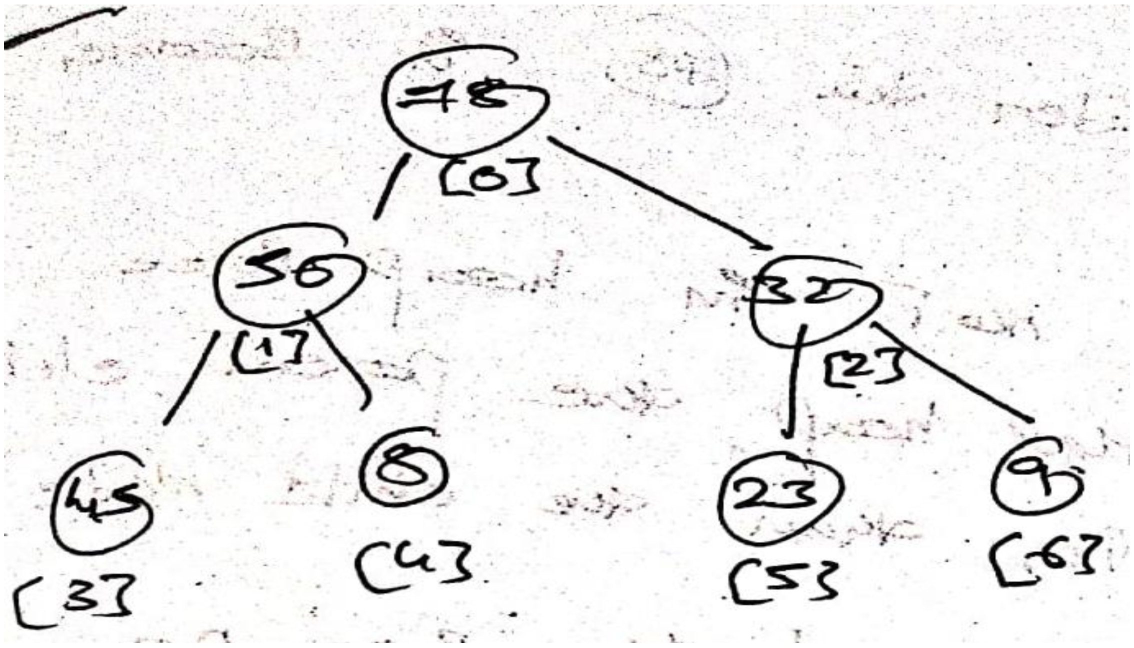
MAX HEAP = PARENT NODE IS GRATER THAN THE CHILD NODE(P>C)

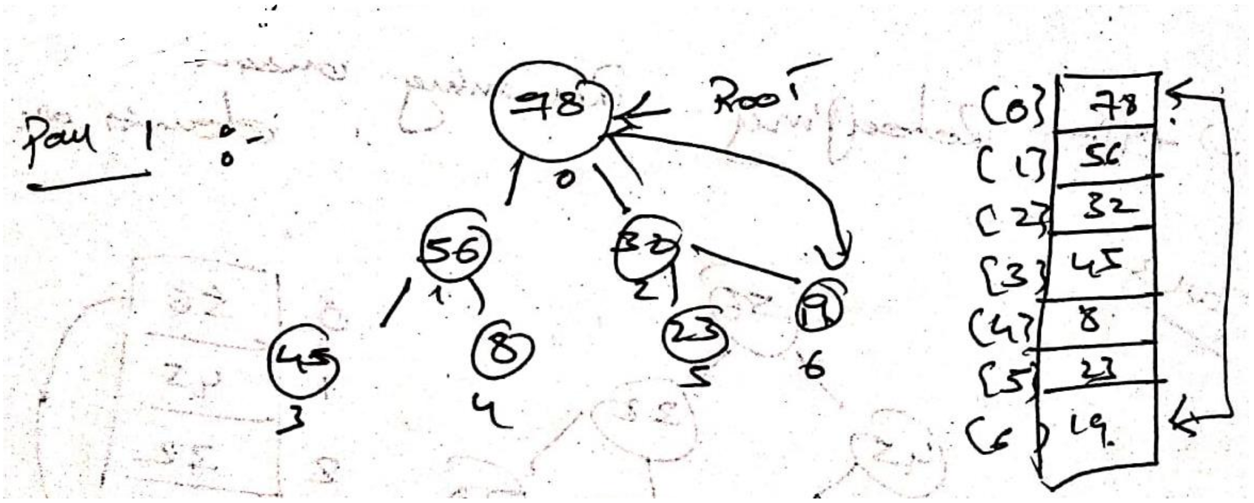
MIN HEALP= PARENT NODE IS LESS THAN THE CHILD NODE (P<C)

HEAP SORT 11EXCHANGE PROCESS:

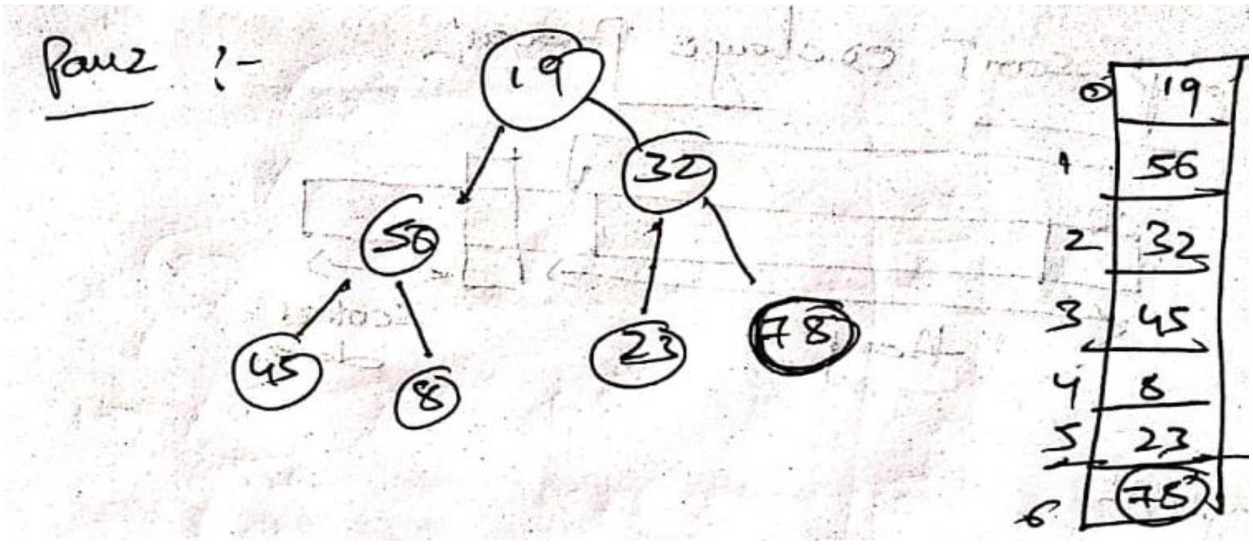


EXAMPLE:

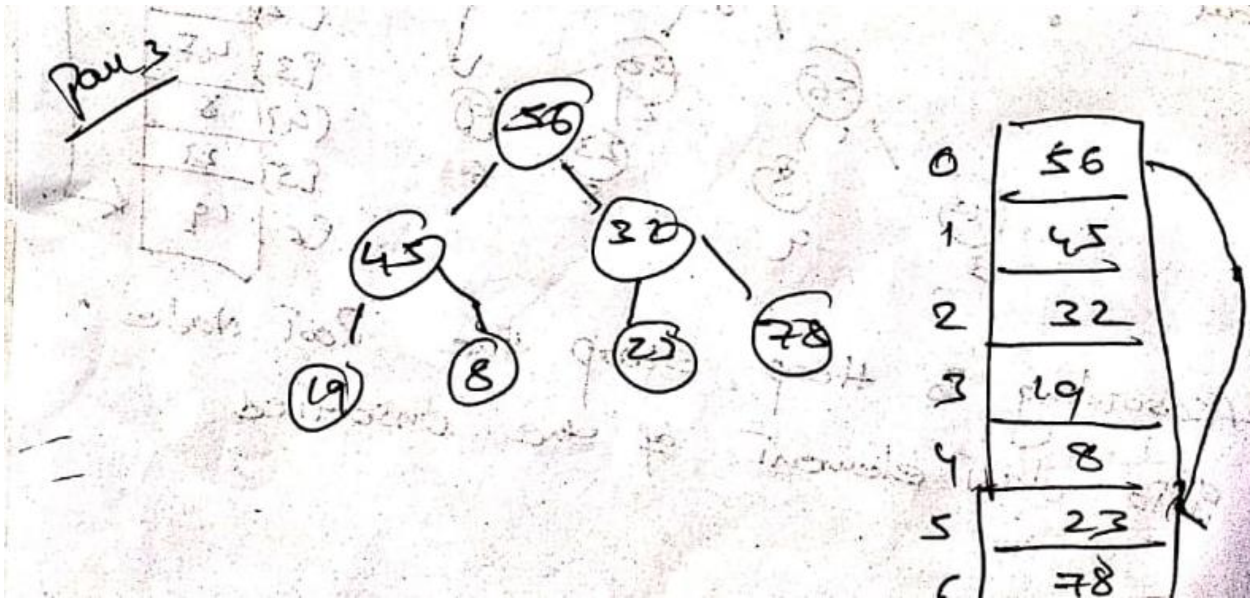




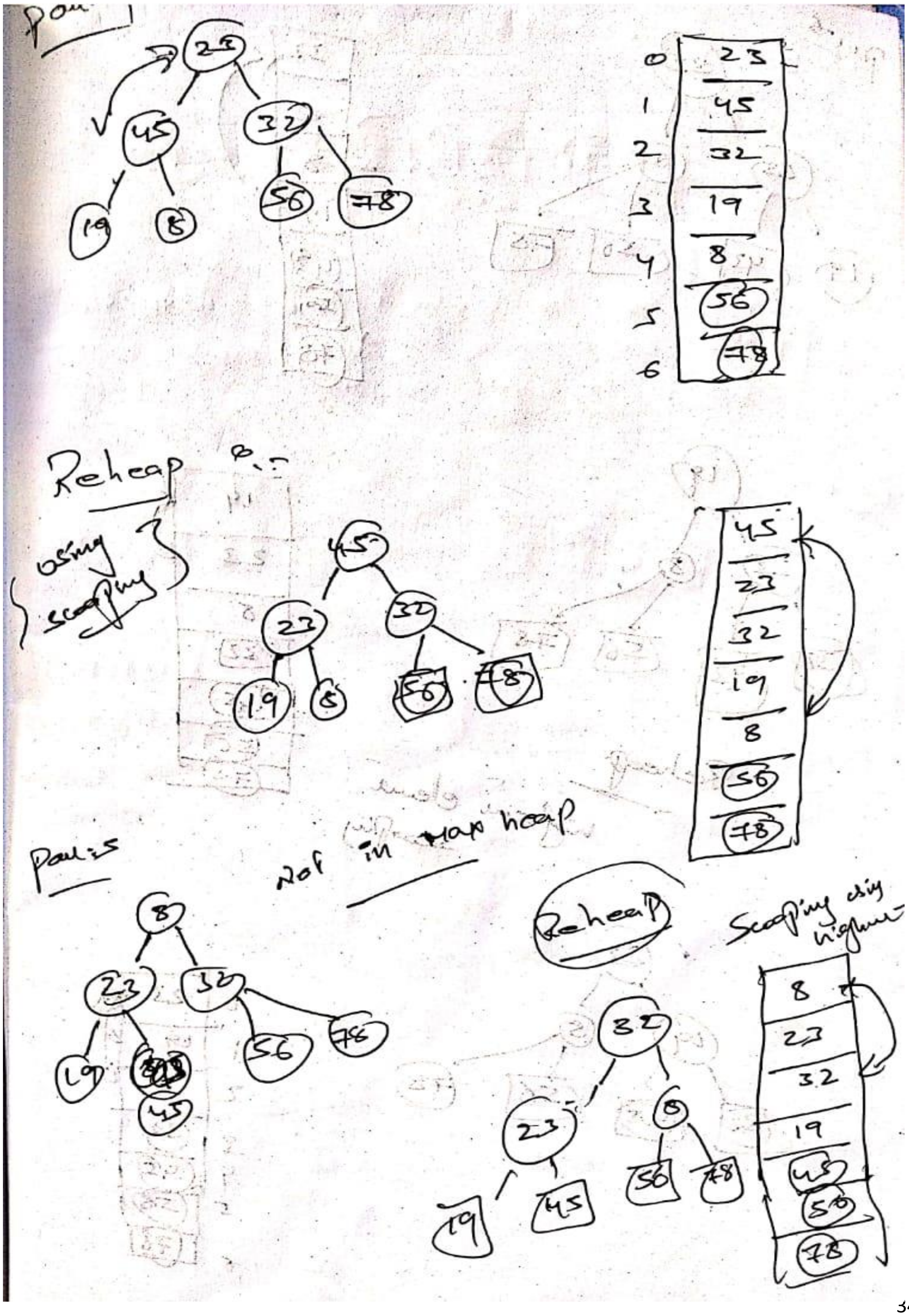
- **ACCORDING to heap sort swap the root node into last element of the unsorted.**

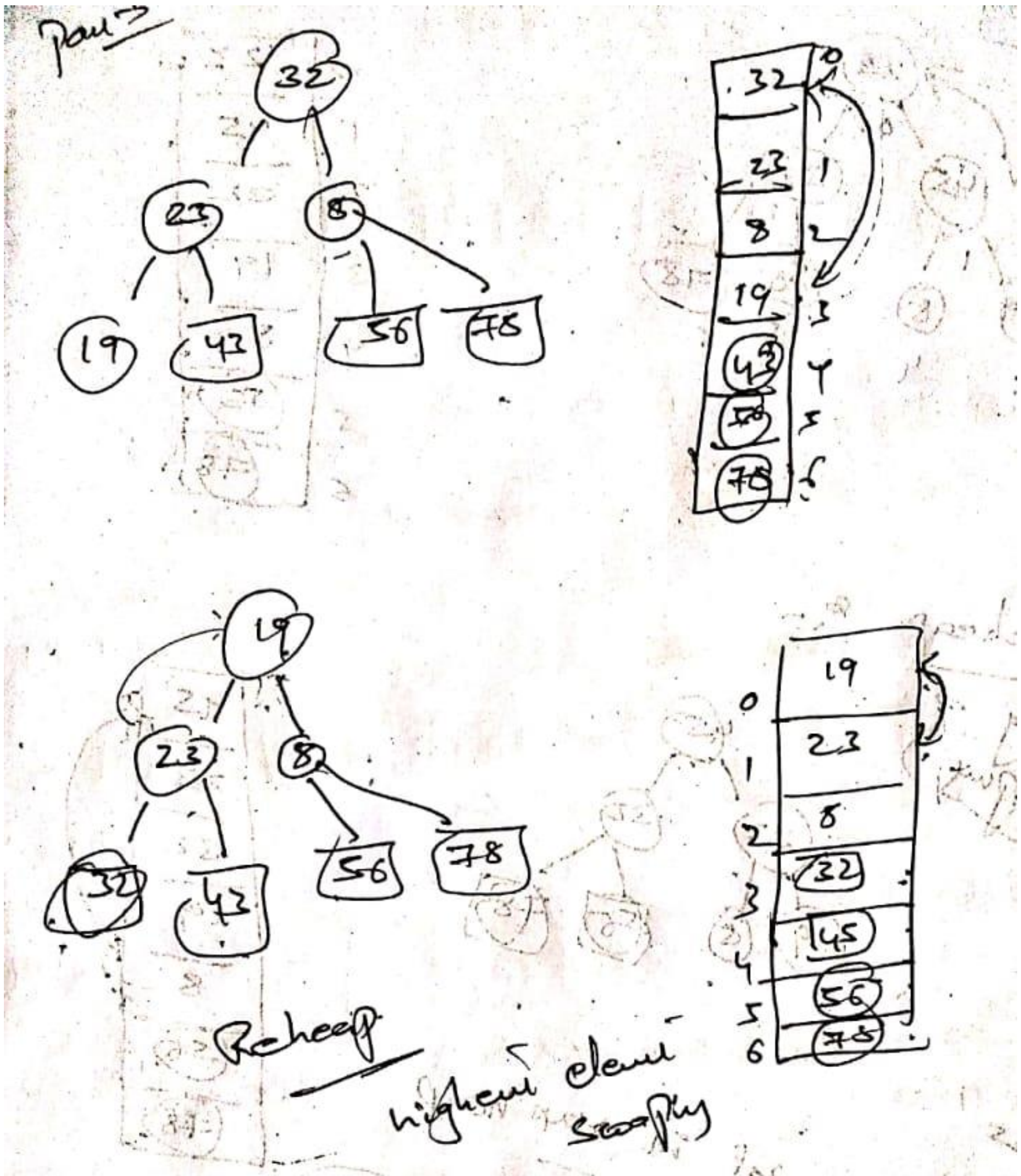


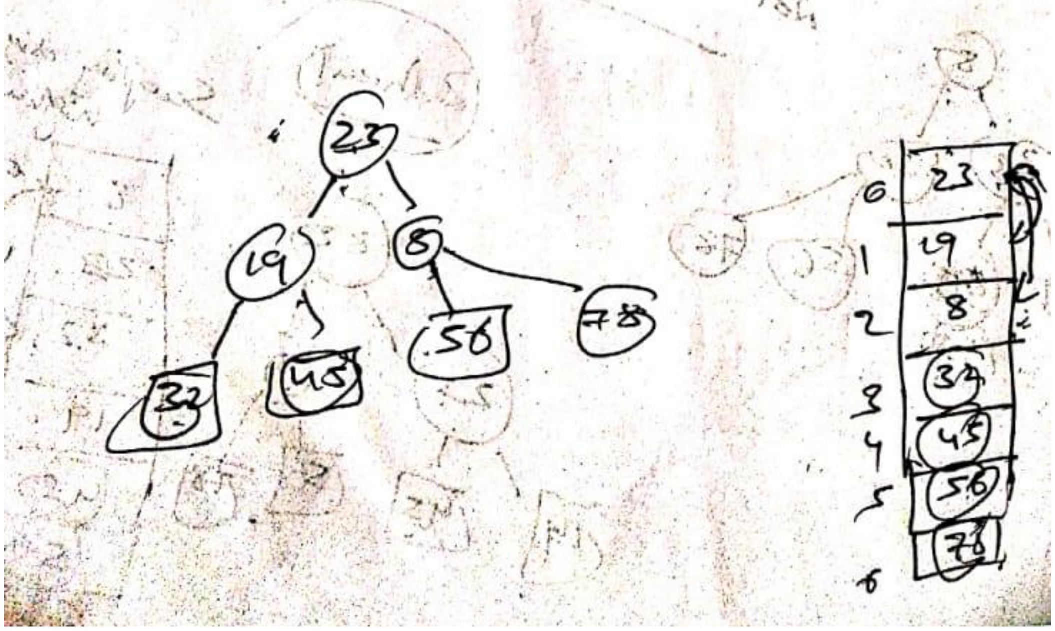
- Here no need to consider this **78 it becomes sorted**
- **Pass 2 is not in heap tree**
- **According to max heap the parent element should be grater than the child node**
- **So reheaping shold be done**
- **After reheaping the remaning unsorted elements are**



Only for reference please follow textbook for more explanation



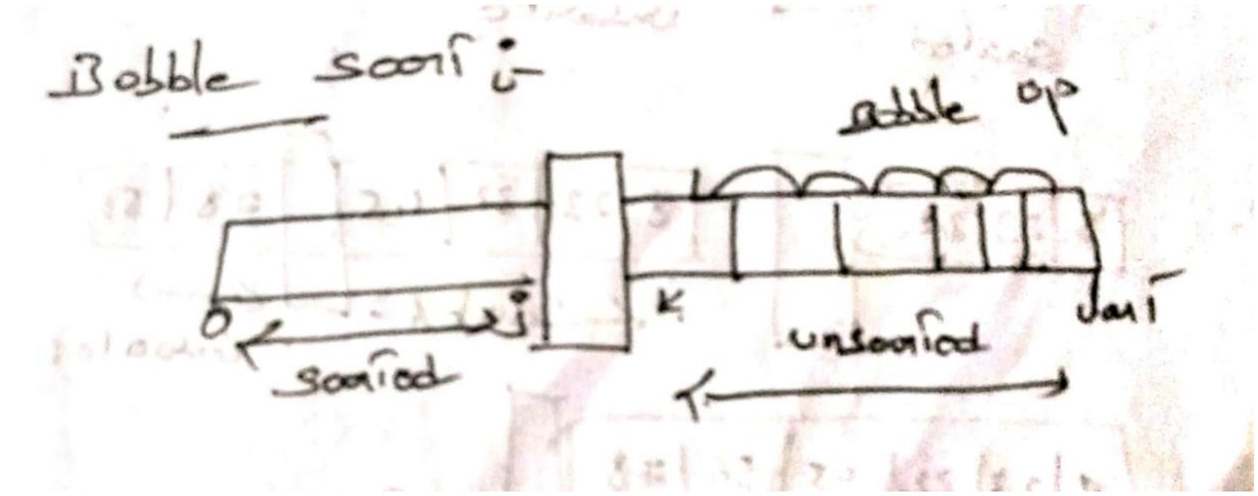

pass 3
32
23
8
19
43
56
78
0
1
2
3
4
5
6
19
23
8
32
45
56
78
Reheap
highest element swapping

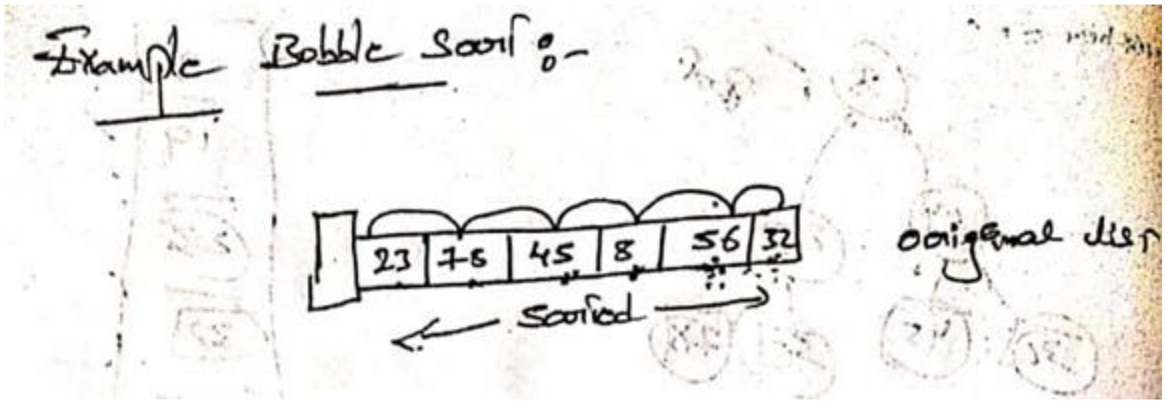

23
19
8
32
45
56
78
0
1
2
3
4
5
6

Bubble Sort

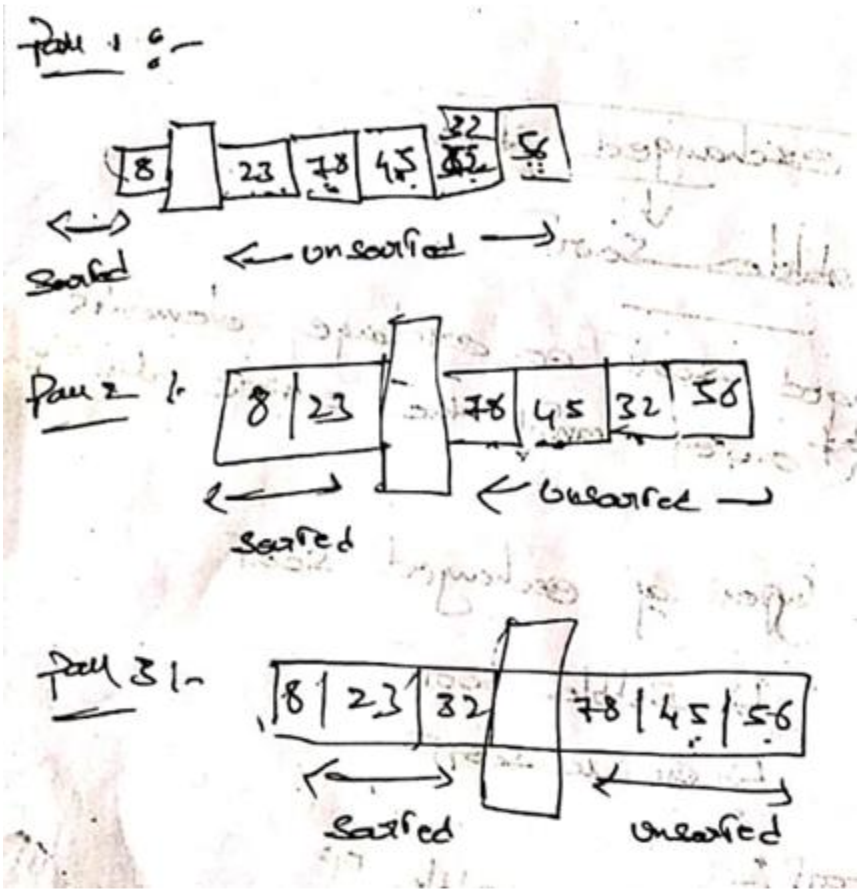
- **In bubblesort we exchange elements that are out of order until the entire list is sorted.**
- **Take the last element check with the next element and small element will comes to sorted part**

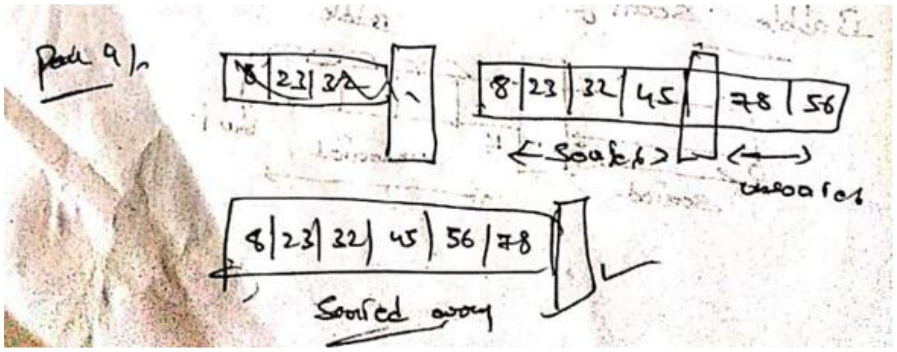
Example:





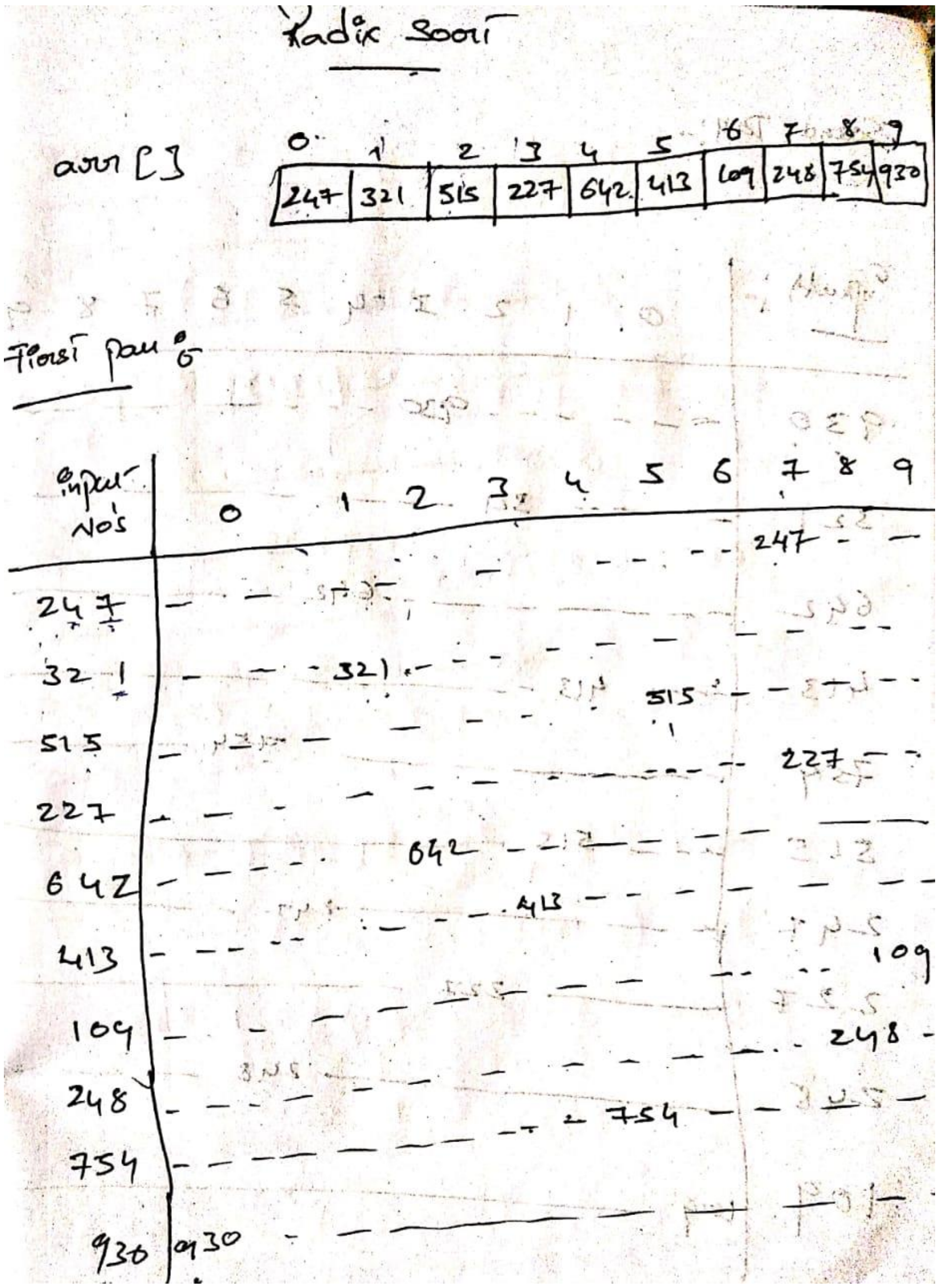
Only for reference please follow textbook for more explanation



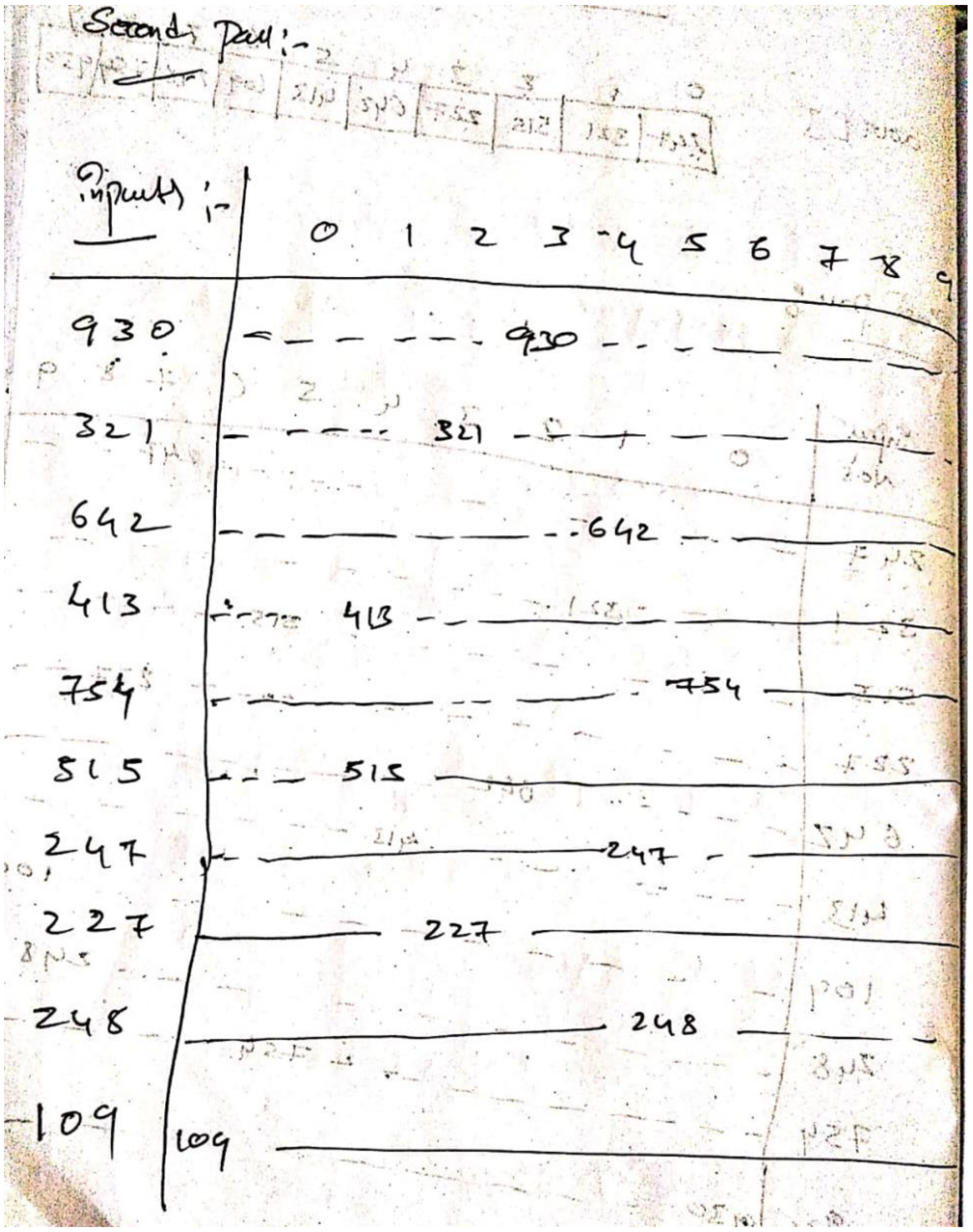


Radix sort

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.



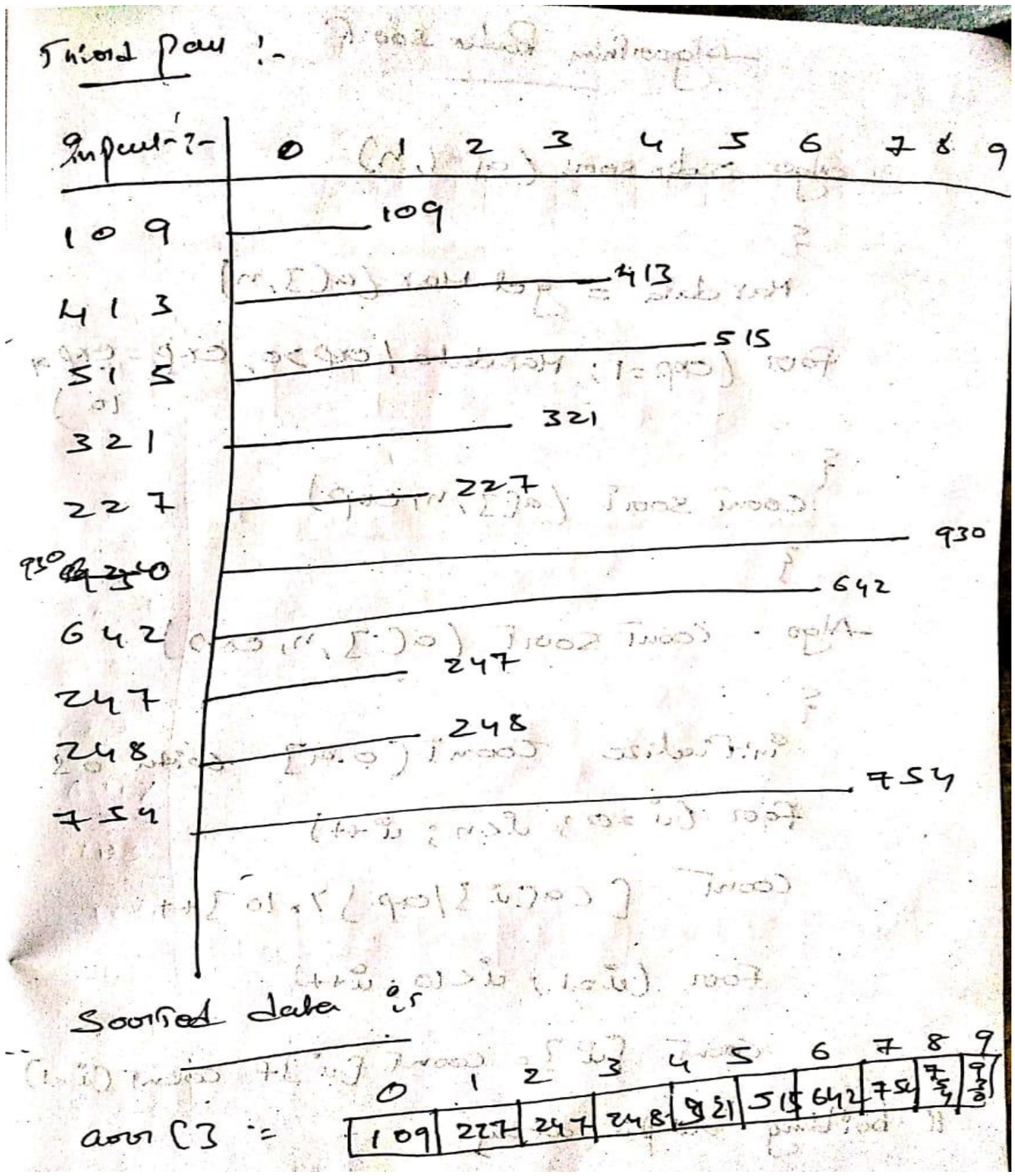
Only for reference please follow textbook for more explanation



Second Pass :-

Inputs :-	0	1	2	3	4	5	6	7	8	9
930				930						
321			321							
642					642					
413		413								
754						754				
515		515								
247					247					
227			227							
248					248					
109	109									

Only for reference please follow textbook for more explanation



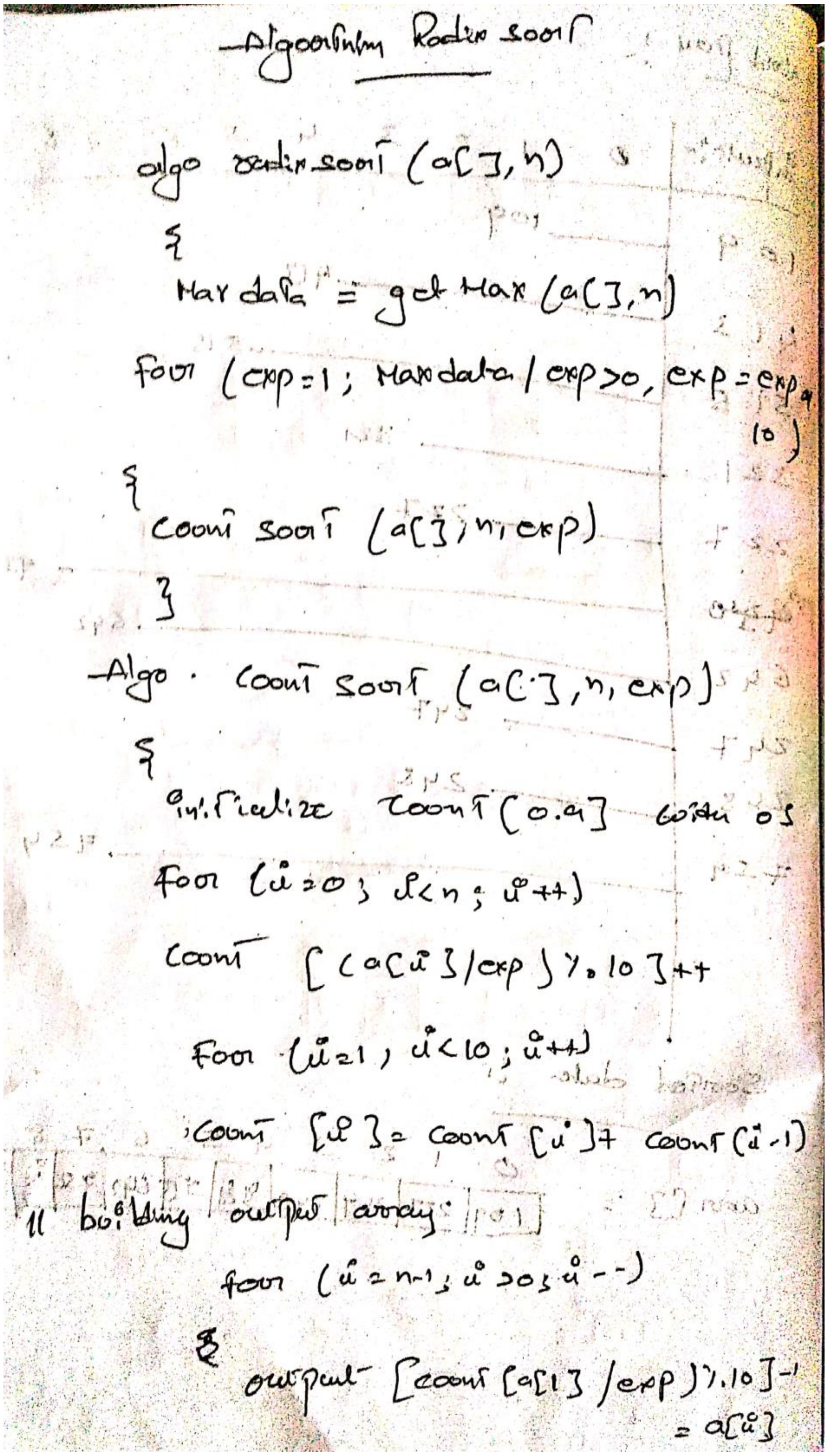
Third pass :-

Sorted data is

0	1	2	3	4	5	6	7	8	9
109	227	247	248	321	515	642	754	754	930

arr[] =

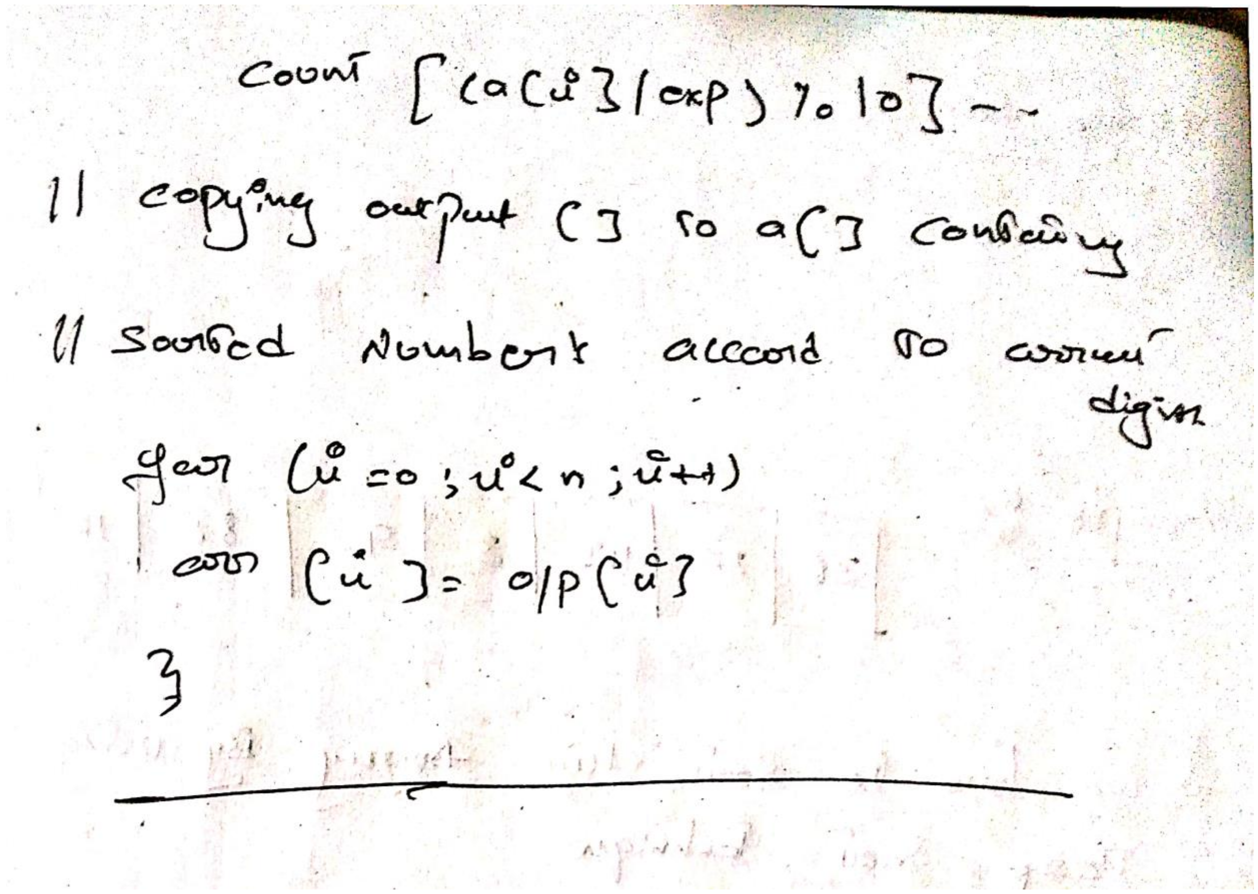
Only for reference please follow textbook for more explanation



Algorithm Radix sort

```
algo radix sort (a[], n)
{
  Max data = get Max (a[], n)
  for (exp=1; Max data / exp > 0, exp = exp *
                                       10 )
  {
    count sort (a[], n, exp)
  }
}

Algo . count sort (a[], n, exp)
{
  initialize count[0..9] with 0s
  for (i=0; i<n; i++)
    count [(a[i]/exp) % 10 ]++
    for (i=1, i<10; i++)
      count [i] = count [i]+ count (i-1)
// building output array
    for (i = n-1; i >= 0; i --)
    {
      output [count [a[i] /exp ) % 10 ]-1
                                   = a[i]
```

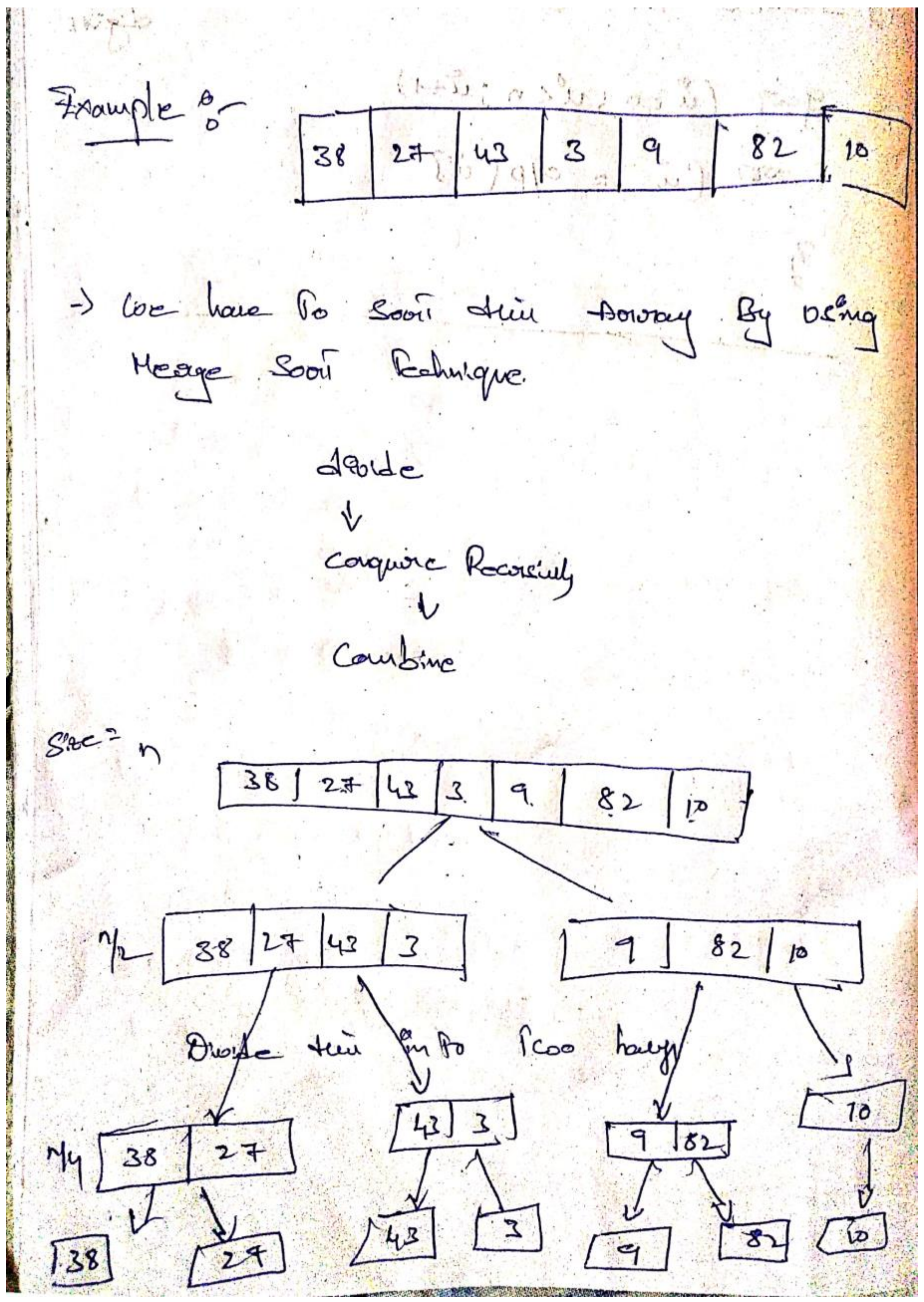


```
      count [(a[i]/exp) % 10] --
// copying output [] to a[] containing
// sorted numbers accord to current
                                  digit
   for (i=0 ; i<n ; i++)
    arr [i ]= o/p [i]
   }
```

Merge Sort

We have to sort this array by using merge sort technique

- Divide
- Conquer recursively
- Combine



Example :-

→ We have to sort this Array By using Merge Sort technique.

divide

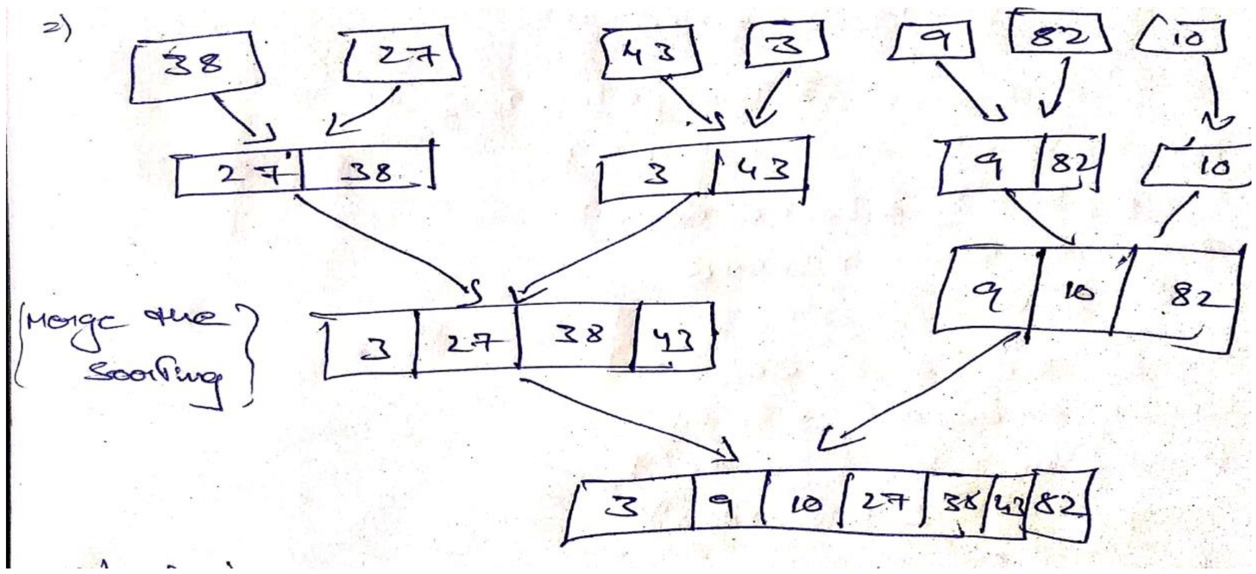
↓

conquer Recursively

↓

Combine

- **Conquer recerversly and compare the elemens come to one**
- **Now combine the array by sorting**
- **Compare the elements**
- **Divide the array until you reach size one**



Advantages :

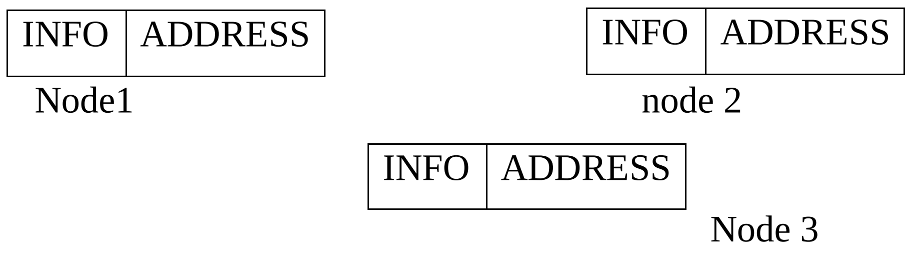
Guranted to run in O/nlogn

Disadvantages:

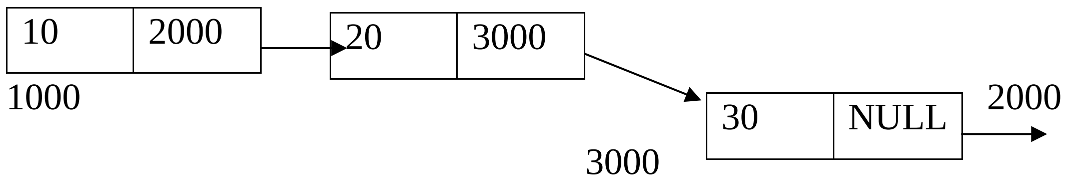
It requires extra space aproxmately N.

Self Referential Structures

- a structure which contains a pointer itself known as self referential structure
- p→ int
- P→P
- Pointer stores the address of the pointer variable
- A structure contains the pointers
- By using the pointers we are using to store the same pointer
- Best example for self referential structure is linked list.
- Linked list: linked list is a collection of nodes which are not necessary to be in adjacent memory location.



- Each node contains two fields
    1. Data field
    2. Address field



- First node contains the address of the second node
- When it stores the address a link will be formed.