

Unit I

Algorithm - An algorithm is a series of steps required to solve the problem. These steps are performed on a sample data representing an instance of a problem.

An algorithm must have following properties

Input - Input data, supplied externally

Output - Result of the program.

Finiteness - Termination of an algorithm should be finite

Definiteness - The steps should be clear and unambiguous

Effectiveness - Algorithm should be feasible enough to convert into a program.

Relation between data structure and algorithm
The structuring of data structure and synthesis of algorithm are related

To understand this concept consider example of a list of 'n' pairs of names and phone numbers $(w_1, n_1), (w_2, n_2), \dots, (w_n, n_n)$:-

If we have to write a program to print the phone number of a person whose name is entered as an input

i) If the list is sequenced on name then binary search algorithm will be able to search the algorithm.

ii) If the list is not sorted then binary search algorithm will not work

In computer Sci. we study different types of data object. For each object there are different operations. The objects should be represented in way that makes efficient implementation operation.

Algorithm Analysis (Goals of writing Hye)
- Algorithm should be easy to understand

write and debug

- Algorithm should make efficient use of computer resources like CPU, memory etc.

When a program to be used few times goal (1) is important

- when the program is to be used many times goal (2) important.

Resources of algorithm analysis

- Memory space complexity
- Computing time time complexity

Space requirement: space required to store variables

The running time of a program (Time Complexity)

Can be judged on the basis of factors such as

- Input to the program
- Size of a program
- Machine language instruction set
- The machine we are executing on
- Time required to execute each machine instruction
- The time complexity of the algorithm of the program

→ Measurement of growth Rate
(Asymptotic Growth Rate)

Asymptotic Consideration

When considering time complexities $f(n)$ and $f_2(n)$ of two different algorithms for a given problem of size n we need to consider and compare the behaviour of the two functions only for large n . If the relative behaviour of two functions for smaller values conflict with the relative behaviour for larger values - then we ignore the conflicting behaviour for smaller values

eg Consider $f_1(n) = 100n^2$
 $f_2(n) = 5n^3$

Representing the time complexities of solutions of two problems

n	$f_1(n)$	$f_2(n)$
1	100	5
5	2500	625
10	10000	5000
20	40000	40000

$f_1(n) \geq f_2(n)$ for $n \geq 20$

We still prefer $f_1(n)$ as time complexity because

$f_1(n) \leq f_2(n)$ for all $n \leq 20$

The constant factor in complexity measure

Let us consider an algorithm having a timing complexity given by the function $f(n) = 100n^2$
[100 :- is a constant n :- size of a problem]

The time required to solve the problem depends not only on the size of the problem but also on h/w and s/w used to execute

The effect of hardware and software on the time required may closely be approximated by a constant.

The time requirement for execution of a solution changes by a constant factor on change in h/w or s/w.

An important conclusion is that if the time taken by one machine in executing a solution is of the order of n^2 (say) the time taken by machine is of order of n^2

Thus the function differ from each other by constant factor when executed as time complexities

Following functions have same time complexities

$$f_1(n) = 5n^2$$

$$f_2(n) = 100n^2$$

$$f_3(n) = 1000n^2$$

$$f_4(n) = n^2$$

The time complexity of $f_1(n) = f_2(n)$

The Notation ~~OTO~~ " O "

Pronounced as big-oh ($O(n)$ is pronounced as big-oh of n) provides Asymptotic upper bound for given functions

[Asymptotic - tendency of an independent variable to increase or decrease without bound]

If there exist two functions $f(x)$ & $g(x)$ with positive integer/real number constants such that

$$f(x) \leq c g(x) \text{ for all } x \geq k$$

The constant is due to asymptotic consideration of constant C for h/w, s/w environment

Example $f(x) = x^2 + 5x$

$g(x) = x^2$ let $c = 2$

Solⁿ

x	$x^2 + 5x$	$2x^2$
1	5	2
2	14	8
3	50	50

$$f(x) \leq c x^2$$

for $x \geq 5$

A symmetric behaviour

for all $x \geq k$

$$f(x) = x^2 + 5x \leq c g(x)$$

where $c = 2$ and $k = 5$

Ex 2] For the functions defined by

$f(x) = 5x^3 + 6x^2 + 1$ show that $f(x) = O(x^3)$

$$f(x) = 5x^3 + 6x^2 + 1 \leq 5x^3 + 6x^3 + 1x^3$$

for all $x \geq 1$

$$\text{or } f(x) \leq 12x^3 \text{ for } x \geq 1$$

there is $c = 12$ $k = 1$

$$f(x) \leq c x^3 \text{ for all } x \geq k$$

Hence $f(x)$ is $O(n^3)$

Time complexity of polynomial is same as the time complexity of the highest degree term.

Q-7 A program fragment without a loop

$$x = x + 1;$$

The timing complexity is of constant order and is represented by $O(1)$

Example for ($i = 1; i \leq n; i++$)

$$x = x + 1;$$

Step 1 will be executed $n+1$ times
 where as Step will be executed n times
 with assumption that each step requires
 a constant time. The time required to execute
 the above program will be the order
 of $2n+1 = O(n)$

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

$$f(n) \leq c n \quad \text{for } c = 3$$

$$\text{and } n \geq 1$$

A loop without ~~nesting~~ complexity is $O(n)$

* Best case, Worst case and the Average case behaviour

Many programs do not produce same time complexity in every case

Consider the problem of searching an element in an array

5	1	4	6	2	11	13	6	7	16
---	---	---	---	---	----	----	---	---	----

Element 5 will be found in 1 attempt

Element 16 require 10 comparisons

Best case behaviour = 1

Worst case = n comparisons

Average case = no. of comparisons will be $n/2$ from probability

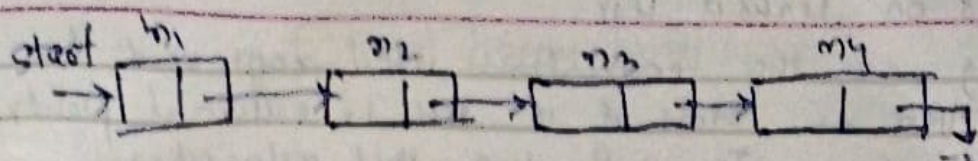
Linked organization

A list is defined as the ordered set of elements which may vary in number

The linked list is a dynamic sequence of items. The elements of the list are connected or pointed to other elements by means of a link.

To manipulate the elements from the list the following operations are required

- Determine whether the list is empty
- Add new elements in the list



A linked can be diagrammatically represented as a collection of elements or a node each containing data as well as the link to the next node in the list. An external pointer is needed to the first node in the list.

For a list of nodes the node should have two fields

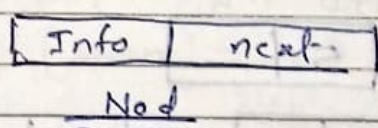
- A data field
- A link/pointer

The data field may be integer, char or may be a record or a struct.

We will use the term "INFO" to refer data field.

The second field is a link to the next node. Depending on type of info the link will be pointer to that type.

NEXT can be used as a label to this field.



Assume info has two fields name & roll no. of the student

Node can be defined as

struct node

{

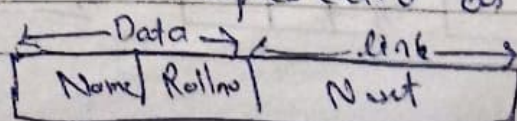
char name[20];

int roll no;

struct node *next;

}

And it can be represented as



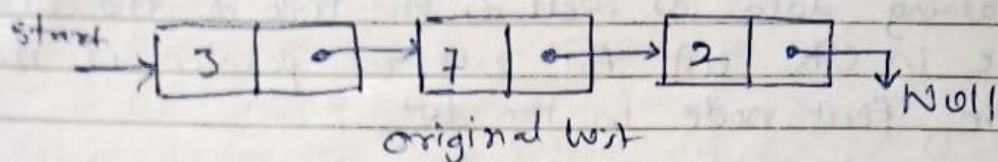
next is a pointer to the next node

Operations on linked list

Following are the conventions used to represent list
START $\Rightarrow$ start of a list is external pointer

Getnode $\Rightarrow$ Is a function that allocates space to node

INFO $\Rightarrow$ info part of a node

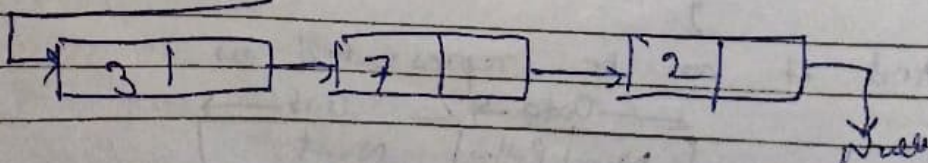
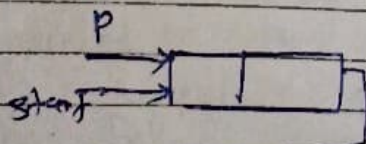
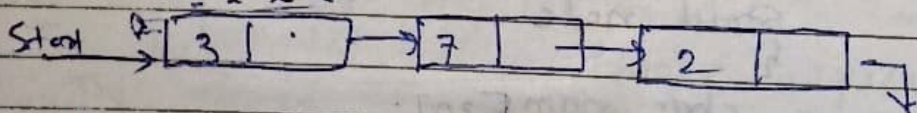
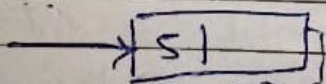
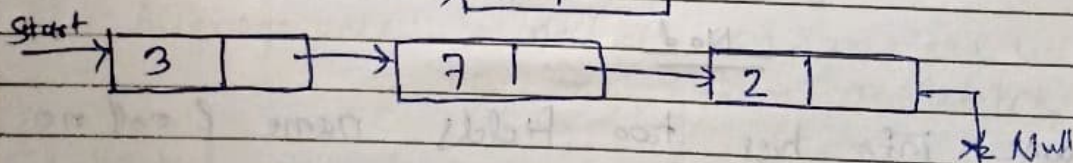
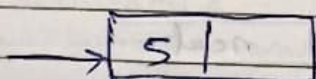
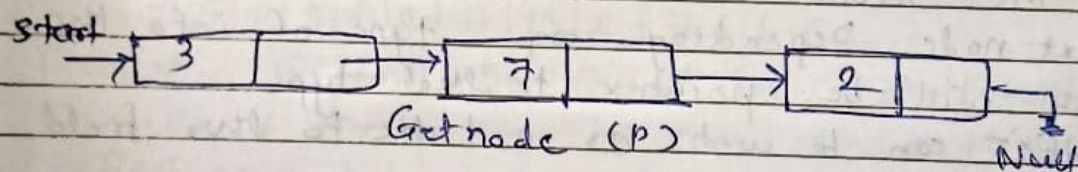


To add a new node at the beginning

First create a space for a node

using GetNode function. 'P' will point to empty node.

- Put data '5' in 'P's information field and connect P in front of list by $P \rightarrow \text{NEXT} = \text{LIST}$



Polynomial

Coeff	Exponent	next
-------	----------	------

Algorithm to traverse a linked list

Step 1: Initialize $ptr = start$

Step 2: Repeat steps 3 & 4 while $ptr \neq NULL$

Step 3: Apply required opⁿ on $ptr \rightarrow info$

Step 4: $ptr = ptr \rightarrow next$

End of loop

Step 5: exit

* Link list operations

struct node

{ int data;

struct node * next

}

struct node * start = NULL

struct node * create (struct node * start)

struct node * new_node, * ptr

int num

printf ("Enter data: ")

scanf ("%d", &num)

while (num != -1)

new_node = ~~malloc~~

struct node * malloc

(sizeof (struct node));

new_node $\rightarrow$ data = num

if (start == NULL)

{ new_node $\rightarrow$ ~~NULL~~ $\rightarrow$ next = NULL

start = new_node;

} else

{ ptr = start;

while (ptr $\rightarrow$ next $\neq$ NULL

ptr = ptr $\rightarrow$ next;


```

    ptr → next = next-node;
    new_node → next = NULL;
}
printf ("Enter the data");
scanf ("%d", & num);
return start;
}

```

* Abstract Data Type

Abstract data type is a type (or class) for objects whose behavior is defined by a set of values and set of operations, so user only needs to know what a data type can do but not how it will do.

ADT can be imagined as a black box which hides inner structure or design type.
 eg. list type of a data can be implemented as Array
 linked list

Stack can be implemented Array / linked list.
 ADT does not specify how the data type is implemented.

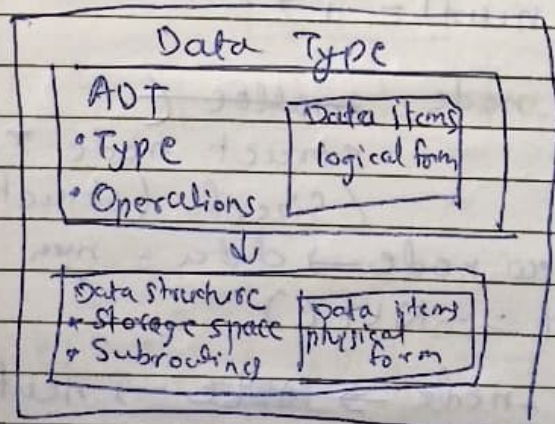


Fig- Relationship between data item, abstract data types and data structures

ADT defines the logical form of the data type. The data structure implements

the physical form of the data type.
 Users of ADT are programmers.
 These ADTs can be used by the programmer as a component in another program. A particular programming environment may provide list of library classes.

Polynomial Representation

$A(x) = a_m x^m + \dots + a_1 x^1$ here a_i are non-zero coefficients with the exponents e_i such that $e_m > e_{m-1} \dots > e_2 > e_1 \geq 0$
 each term will be represented as a node with three fields

Coef	Exp	next
------	-----	------

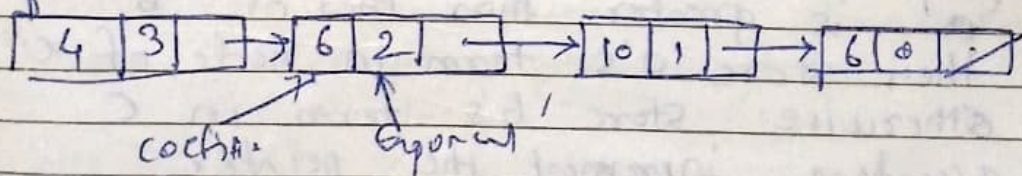
Each polynomial consists of two parts

- coefficient
- Exponent

$$10x^2 + 26x$$

here 10 & 26 are coefficients and 2 & 1 are exponents.

eg $4x^3 + 6x^2 + 10x + 6$



to represent
 struct polyd
 int coeff;
 int pow-val;
 poly * next;
 }


```
class add {
```

```
    poly * poly1, *poly2, *poly3
```

```
public:
```

```
    add() { poly1 = poly2 = poly3 = NULL;
```

```
    void addpoly();
```

```
    void display();
```

```
};
```

```
void add::addpoly
```

```
{ int i, p;
```

```
    poly *next = NULL
```

```
    *end = NULL
```

Algorithm for addition of polynomials

step 1 Read the two polynomials say
a and b (ie create two linked
lists one for a & one for b);

step 2: Compare components of 'a' and 'b'
of each term one by one

If the ^{expo}ponents are equal then

i) add coefficients of both terms of a
and b and store the result in
list 'c' advance to next terms
in a and b

step 3 i) If the exponents of the term of
'a' is greater than that of 'b'
then store 'a's' term in node of 'c'
otherwise store b's term in c
according increment the pointer

step iii) If 'a' is exhausted then copy
containing elements of 'b' (if any)
in 'c' or if b is now empty
copy the remaining elements of
a (if any) in c

step 3 Display the list 'c'

step 4 Stop

Sparse Matrix

A Matrix $m \times n$ m : colⁿ & n : rows
there may be a situation in which a matrix contains more number of zero values than non zero

when a sparse matrix is represented with a two dimensional array we waste lot of space to represent that matrix

eg. Consider a array
to represent only '0' values

$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$	$\Rightarrow$	<table><tr><th>Row</th><th>Colⁿ</th><th>Value</th></tr><tr><td>0</td><td>4</td><td>9</td></tr><tr><td>0</td><td>1</td><td>8</td></tr><tr><td>2</td><td>0</td><td>4</td></tr><tr><td>2</td><td>3</td><td>2</td></tr><tr><td>3</td><td>5</td><td>5</td></tr><tr><td>4</td><td>2</td><td>2</td></tr></table>	Row	Col ⁿ	Value	0	4	9	0	1	8	2	0	4	2	3	2	3	5	5	4	2	2
Row	Col ⁿ	Value																					
0	4	9																					
0	1	8																					
2	0	4																					
2	3	2																					
3	5	5																					
4	2	2																					

Traversing a linked list

- Step 1: - [initialize] set $ptr = \text{Start}$;
Step 2: - Repeat steps 3 and 4 while $ptr \neq \text{NULL}$
Step 3: - Apply process to $ptr \rightarrow \text{data}$;
Step 4: set $ptr = ptr \rightarrow \text{next}$
END of loop
Step 5: Exit

Algorithm to Print the number of nodes in a linked list

- Step 1: (initialize) set $count = 0$;
Step 2: if [initialize] set $ptr = \text{Start}$;
Step 3: Repeat steps 4 and 5 while $ptr \neq \text{NULL}$;

step 4 set count = count + 1;
 step 5: set ptr = ptr → next;
 end of loop
 step 6 Write count;
 step 7 Exit;

Algorithm to search a linked list

step 1: set ptr = start;
 step 2: Repeat step 3 while ptr != NULL
 step 3: If Val = ptr → Data
 set pos = ptr (address of node)
 Go to step 5
 step Else
 set ptr = ptr → next
 END IF
 step 4 END LOOP
 step 4 pos = NULL

Inserting a new node

step 1: if Avail = NULL
 write overflow
 Go to step 7
 end if
 step 2: set new node = Avail;
 step 3: set avail = avail → next
 step 4: set new node → data = Val
 step 5: set new node → next = start
 step 6: set start = new node
 step 7: exit

Circular linked list

In a circular linked list the last node contains a pointer to the first node of the list. We can have a circular singly linked list. We can start from any node and traverse a list.

Step 1 : if avail = NULL
 write overflow
 Go to step 11
 Step 2 : ^{End if} set new node = avail
 Step 3 : set avail = avail → next
 Step 4 : set new node → data = Val
 5 : ~~stop~~ set ptr = start
 6 : Repeat step 7 while ptr → next ≠ start
 7 : ptr = ptr → next
 end of loop
 8 : set new node → next = start
 9 : set ptr next = new node
 10 : set start ~~start~~ = new node
 11 : Exit

Algorithm - To insert new node in the linked list.

Algorithm - Insert a new node at the end

Step 1 : - If Avail = Null
 write overflow
 Go to step 10
 [End of If]
 Step 2 : set new node = Avail
 Step 3 : set avail = avail → next
 Step 4 : set new node → data = val
 Step 5 : set new node → next = start
 Step 6 : set ptr = start
 Step 7 : Repeat step 8 while ptr → next = start
 Step 8 : set ptr = ptr → next
 end of loop
 Step 9 : set ptr → next = new node
 Step 10 : Exit

Algorithm to delete first node

Step 1: if start = Null
write overflow

Go to step 8

2 set ptr = start;

3 Repeat Step 4 while ptr $\rightarrow$ next $\neq$ start

4: Set ptr = ptr $\rightarrow$ next

end of loop

5: set ptr $\rightarrow$ next = start $\rightarrow$ next;

6 free start

7 set start = ptr $\rightarrow$ next

8 Exit

The concept of pointers

Concept - The variables which are used to hold memory addresses are called pointer variables.

A pointer variable is therefore a variable which holds an address of some other variable and the value of a pointer variable gets stored in another variable another memory location.

They point to the (or hold the address) of the variable of same type.

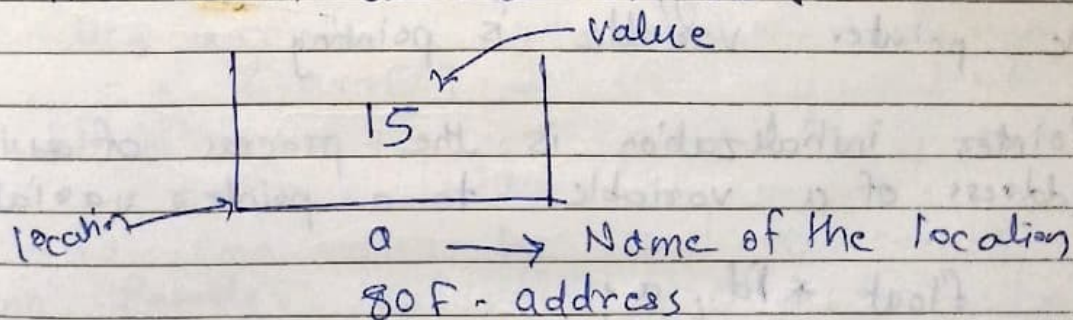
Address :

Whenever a variable is defined in 'C' a memory location is assigned to it, in which it's value will be stored.

This address can be checked using '&' symbol.

If 'var' is a name of variable then $\&\text{var}$ is an address of the variable.

Whenever a variable is declared in a program system allocates a location i.e. an address to that variable in the memory to hold the assigned value. This location has its own address number.



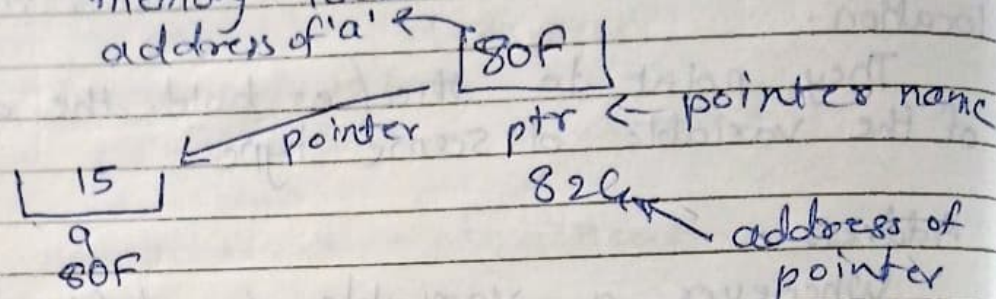
We can access the value 'a' either by using the variable name as 'a' or by using address '80F'.

Now how to access this address.

The variables which are used to hold

memory addresses are called pointer variables.

A pointer variable is therefore a variable which holds an address of some other variable and the value of a pointer variable gets stored in another memory location.



Benefits of using pointers

1. Pointers are more efficient in handling arrays and structures
2. Pointers allow references to function and helps in passing functions as arguments in other functions
3. It reduces length of the program and its execution time
4. Support dynamic memory allocation

Declaration of pointer variable

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing

Pointer initialization is the process of assigning address of a variable to a pointer variable

```
float *fp, a;
```

```
int *ip;
```

```
a = 10;
```

```
p = &a;
```

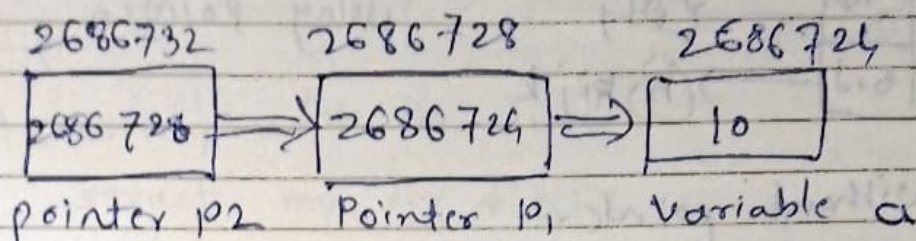
```
printf("%d", *p) // prints value of a
```

```
printf("%d", &a) // prints address of a
```

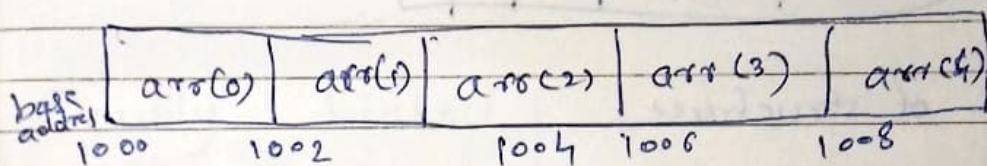


```
printf("%u", &a);    print address of a
printf("%u", &p);    print address of p
printf("%d", p);     print address of p
```

Pointer to pointer



Pointers and Arrays



int a[5]

Variable arr will give same base address, pointing to first element of an array.

int *p

p = arr;

p = &arr[0];

} both the statements are equivalent.
Here arr++ or p++ is possible

Pointers can also be used to create a string. Pointer variables of char type are used to create strings.

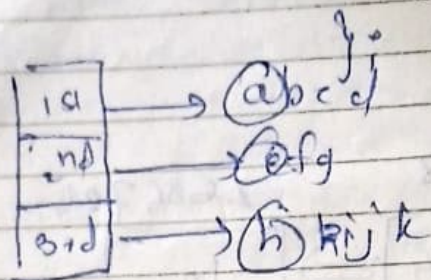
char *str;

str = "Hello"

printf("%c", str);

Array of pointers

```
char * name[3]
= { abcd,
    efg,
    hijk }
```



using pointer

without pointer

a	b	c	d			
e	f	g				
h	i	j	k			

without pointer

Array of structures

```
struct book
{ char nam[10];
  int price;
}
```

```
int main()
```

```
{ struct book a;
```

```
  struct book * ptr;  // pointer of structure type
```

```
  ptr = &a; struct book b[10];
```

```
  struct = book b[10];
```

```
  struct book * p
```

```
  p = &b
```


$b[0]$ ^{most variable} array of
name price ... $b[17] b[1]$ ^{structures}
^{variables}
 points to 1st variable
 $[&b[0]] \leftarrow$ struct base
 p ^{type}
^{pointer p}

Accessing with structure

struct my_str * ptr

ptr -> va

ptr -> data

Doubly Linked List

It is a data structure with three fields
One information field and two link fields

llink	info	rlink
-------	------	-------

llink - to attach to previous node

rlink - to the next node

Here in case of DLL both the llink and rlink need to be attached

struct node

```

{
    int info;
    struct node *llink, *rlink;
};

```

Let start be a pointer pointing to the start of DLL. Initially when no elements are present, it points to NULL. Now get the node say 'p' i.e.

$p = \text{malloc}(\text{size of } (\text{struct node}))$

$10 \rightarrow \text{info} = x$; $1+x$ is any number read

$p \rightarrow \text{link} = \text{NULL}$

$p \rightarrow \text{info} = x$; $1+x$ is any number read

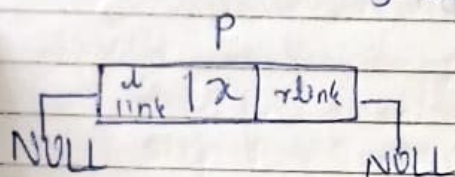
$p \rightarrow \text{link} = \text{NULL}$

$p \rightarrow \text{link} = \text{NULL}$

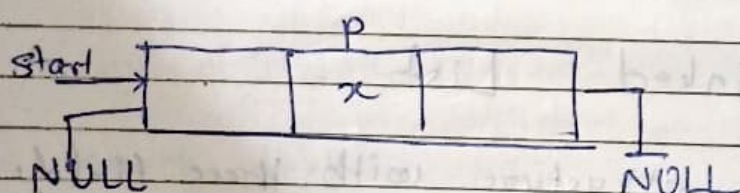
Now if start is NULL then list is currently empty

$\text{start} = p$ means 'p' itself is a beginning of list

$\text{start} = \text{NULL}$



a) New node 'p' is created



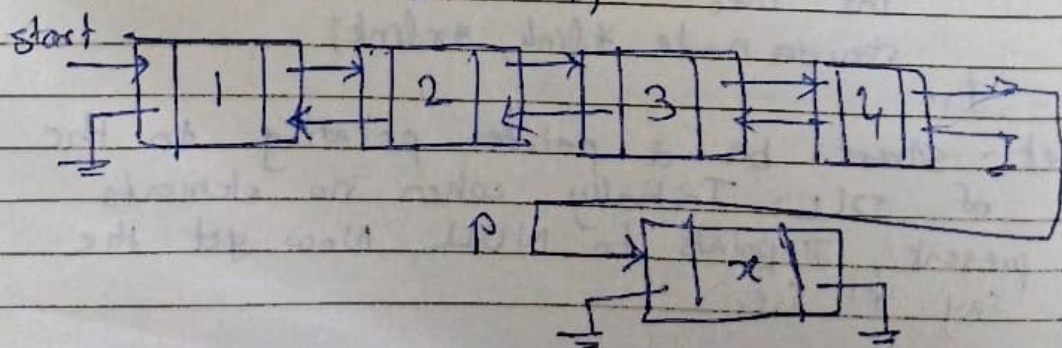
b) When list is empty, START points to pointer 'p'

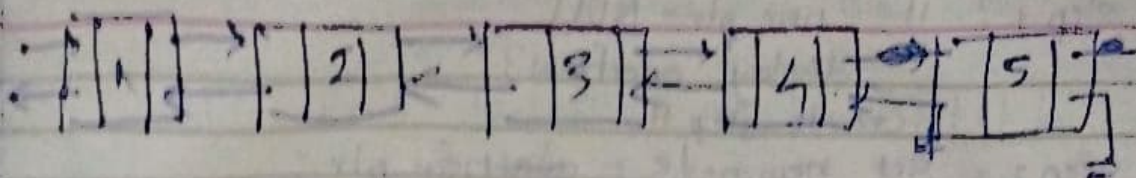
Otherwise, if already some elements are present then travel along the list till end
Let 'q' point to the last node

Then attach 'p' to 'q'

$q \rightarrow \text{link} = p$

$p \rightarrow \text{link} = q$

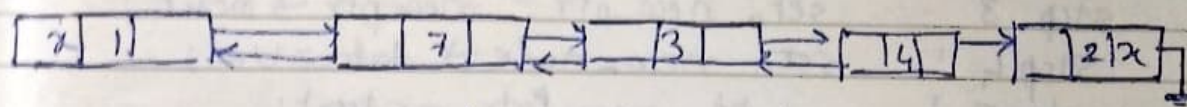




Algorithm to insert new node at the beginning

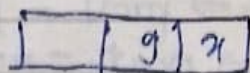
- Step 1: if $\text{new_ptr} = \text{NULL}$;
display overflow;
Go to step 9;
- Step 2: set $\text{new_node} = \text{new_ptr}$;
- Step 3: set $\text{new_ptr} = \text{new_ptr} \rightarrow \text{next}$;
- Step 4: set $\text{new_node} \rightarrow \text{data} = \text{Var}$;
- Step 5: set $\text{new_node} \rightarrow \text{prev} = \text{NULL}$;
- Step 6: set $\text{new_node} \rightarrow \text{next} = \text{start}$;
- Step 7: set $\text{start} \rightarrow \text{prev} = \text{new_node}$;
- Step 8: set $\text{start} = \text{new_node}$;
- Step 9: Exit;

Inserting a new node at the end of the Doubly Linked list.



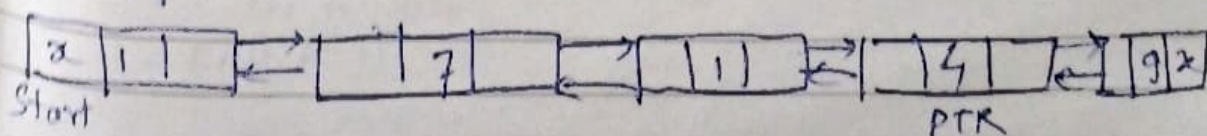
add a new node with val 9

Allocate the memory for the new node and initialize its data part as 9 and its next-field to NULL



Take a pointer variable ptr and make it point to the first node of the list.

Start_ptr



step 1: if $\text{new_ptr} = \text{NULL}$
display overflow;
Go to step 11

step 2: set $\text{new_node} = \text{new_ptr}$;

step 3: set $\text{new_ptr} = \text{new_ptr} \rightarrow \text{next}$;

step 4: set $\text{new_node} \rightarrow \text{data} = \text{Val}$;

step 5: set $\text{new_node} \rightarrow \text{next} = \text{NULL}$;

step 6: set $\text{ptr} = \text{start}$;

step 7: Repeat step 8 while $\text{ptr} \rightarrow \text{next} \neq \text{NULL}$;

step 8: set $\text{ptr} = \text{ptr} \rightarrow \text{next}$
end of loop

step 9: set $\text{ptr} \rightarrow \text{next} = \text{new_node}$;

step 10: set $\text{new_node} \rightarrow \text{prev} = \text{ptr}$;

step 11: ~~ptr~~ exit

Algorithm to insert a node after a given node:

Step 1: If $\text{new_ptr} = \text{NULL}$;
write overflow.
Go to step 12
end of if

step 2: set $\text{new_node} = \text{new_ptr}$;

step 3: set $\text{new_ptr} = \text{new_ptr} \rightarrow \text{next}$;

step 4: set $\text{new_node} \rightarrow \text{data} = \text{Val}$;

step 5: set $\text{new_ptr} = \text{start}$;

step 6: Repeat step 7 while $\text{ptr} \rightarrow \text{data} \neq \text{num}$;

step 7: set $\text{ptr} = \text{ptr} \rightarrow \text{next}$;
end of loop

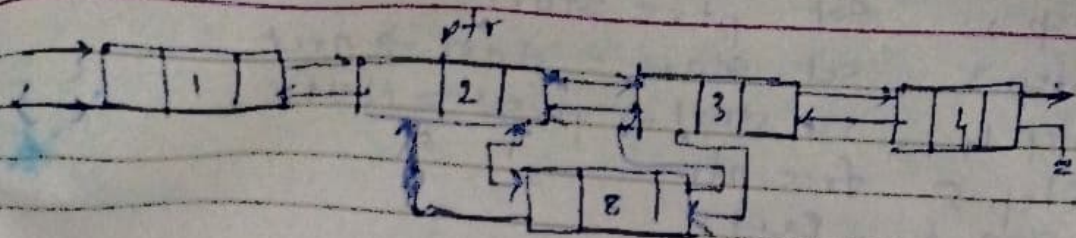
step 8: set $\text{new_node} \rightarrow \text{next} = \text{ptr} \rightarrow \text{next}$;

step 9: set $\text{new_node} \rightarrow \text{prev} = \text{ptr}$;

step 10: set $\text{ptr} \rightarrow \text{next} = \text{new_node}$;

step 11: set $\text{ptr} \rightarrow \text{next} \rightarrow \text{prev} = \text{new_node}$;

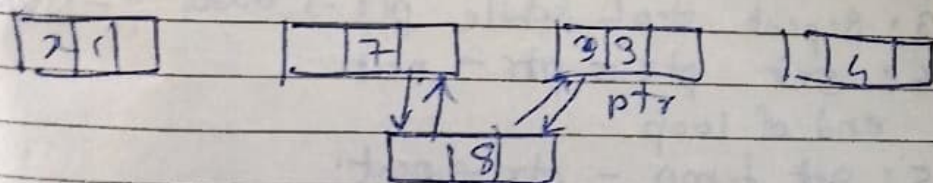
step 12: exit;



Algorithm

Insert a new node before a node.

- step 1: if $\text{new_ptr} = \text{NULL}$
 display overflow
 Go to step 12
- step 2 set $\text{new_node} = \text{new_ptr}$;
- step 3 set $\text{new_ptr} = \text{new_ptr} \rightarrow \text{next}$;
- step 4 set $\text{new_node} \rightarrow \text{data} = \text{Val}$
- step 5 set $\text{ptr} = \text{start}$
- step 6 Repeat Step 7 while $\text{ptr} \rightarrow \text{data} \neq \text{New Node}$
- step 7 set $\text{ptr} = \text{ptr} \rightarrow \text{next}$;
 end of loop
- step 8 set $\text{new_node} \rightarrow \text{next} = \text{ptr}$
- step 9 set $\text{new_node} \rightarrow \text{prev} = \text{ptr} \rightarrow \text{prev}$
- step 10 set $\text{ptr} \rightarrow \text{prev} = \text{new_node}$;
- step 11 set $\text{ptr} \rightarrow \text{prev} \rightarrow \text{next} = \text{new_node}$
- step 12 Exit



Operation of deletion on a linked list

Algorithm to delete the first node

- step 1: if $\text{start} = \text{NULL}$
 display overflow;
 Go to step 6
 end of if

step 2: set $ptr = start$;
 step 3: set $start = start \rightarrow next$.
 step 4: set $start \rightarrow prev = NULL$
 step 5: free ptr ;
 step 6: Exit;

Algorithm to delete last node

step 1: if $start = NULL$
 display overflow
 Go to step 7
 end of if
 step 2: set $ptr = start$;
 step 3: Repeat step 4 while $ptr \rightarrow next \neq NULL$
 step 4: set $ptr = ptr \rightarrow next$;
 end of loop
 step 5: set $ptr \rightarrow prev \rightarrow next = NULL$;
 step 6: free ptr
 step 7: Exit

Algorithm to delete a node after a given node

step 1: if $start = NULL$
 write overflow
 Go to step 9
 End of if
 step 2: set $ptr = start$
 step 3: Repeat step 4 while $ptr \rightarrow data \neq NUM$
 step 4: set $ptr = ptr \rightarrow next$
 end of loop:
 step 5: set $temp = ptr \rightarrow next$;
 step 6: set $ptr \rightarrow next = temp \rightarrow next$;
 step 7: set $temp \rightarrow next \rightarrow prev = ptr$;
 step 8: free $temp$
 step 9: Exit;

Algorithm to delete a node before a given node

Step 1 if start = NULL

display overflow

Go to step 9

end of if

Step 2 set ptr = start;

Step 3 Repeat step 4 while ptr → data != Num

Step 4 set ptr = ptr → next;

end of loop

Step 5 Set temp = ptr → prev;

Step 6 set temp → prev → next = ptr;

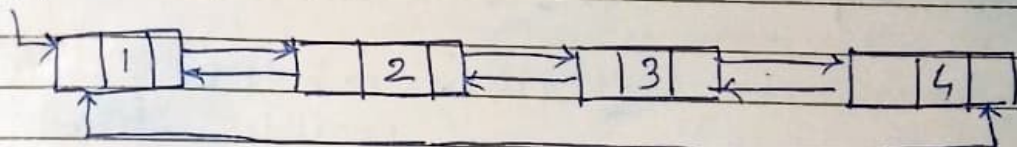
Step 7 Set ptr → prev = temp → prev

Step 8; free temp

Step - 9 : exit

Circular doubly linked list

Start



Memory representation

	Data	prev	Next
1	H	9	3
2	///	///	///
3	E	1	6
4	///	///	///
5	///	///	///
6	L	3	7
7	L	6	9
8	///	///	///
9	0	7	1

two Cases

1. A new node to be added at the beginning
2. A new node is inserted at the end.

Algorithm to insert a new node at the beginning

Step 1 if new_ptr = NULL
write overflow

Go to step 13

Step end of if

Step 2 set new_node = new_ptr;

Step 3: set new_ptr = new_ptr -> next;

Step 4: set new_node -> data = val;

Step 5: Set ptr = start;

Step 6: Repeat step 7 while ptr -> next ¹² start

Step 7 set ptr = ptr -> next;
end of loop

Step 8 set ptr -> next = new_node

Step 9 set new_node -> prev = ptr

Step 10 set new_node -> next = start

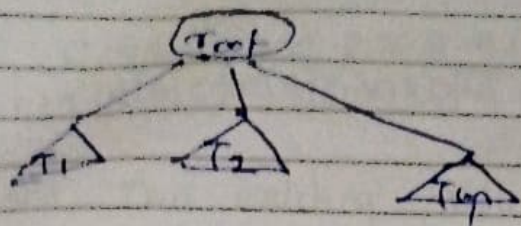
Step 11 set start -> prev = new_node

Step 12 set start = new_node

Step 13 exit;

Non-Linear Data Structures

Trees



Root - It is a special node in a tree structure and the entire tree is referenced through it. This node does not have a parent.

Parent - It is an immediate predecessor of a node.

Child - All immediate successors of a node are its children.

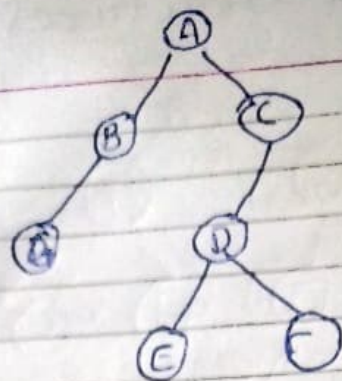
Siblings : Nodes with the same parent are called siblings.

Path - It is a number of successive edges from source node to destination node.

Degree of a node - The degree of a node is a number of children of a node.

Binary tree

A tree is a binary if each node of the tree can have maximum of two children. Moreover children of a node of binary tree are ordered. One child is called left child and other is called right child.



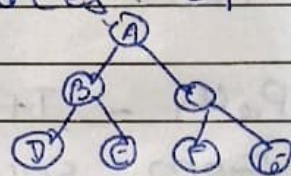
Node 'A' has two children B and C. Similarly nodes 'B' and 'C' each have one child 'G' and 'D' respectively.

Representation of a binary tree using array

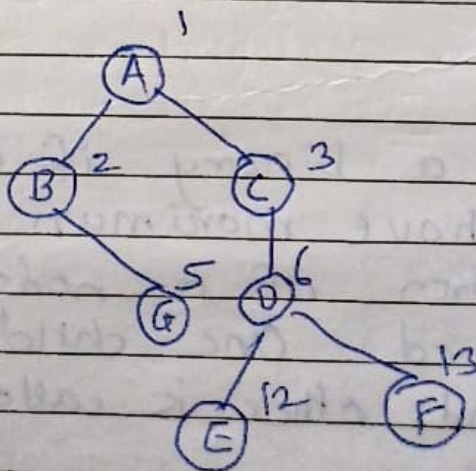
In order to represent a tree in a single one dimensional array, the nodes are numbered sequentially level by level left to right. - Even empty nodes are numbered.

When the data of a tree is stored in an array then the number appearing against the node will work as indices of the node in an array.

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G



The location number '0' can be used to store the size of the 'tree' in terms of total number of nodes (existing or not existing)



Numbering of nodes with some

Array representation

0	1	2	3	4	5	6	7	8	9	10	11	12	13
13	A	B	C	D	E	F	G	H	I	J	K	L	M

All non-existing children are shown by "10" in array.

Index of the left child of a node $i = 2 \times i$

Index of the right child of a node $i = 2 \times i + 1$

Index of the parent of a node $i = i/2$

Sibling of a node i will be found at location $i \pm 1$ if i is a left child

Linked Representation of a binary tree

Linked representation of a binary tree is more efficient than array representation.

A node of a binary tree consists of three fields as shown below:

- Data
- Address of the left child
- Address of the right child

left	Data	Right
------	------	-------

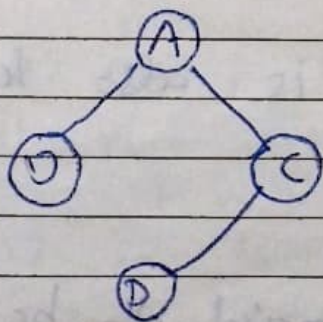
struct node

{ int data;

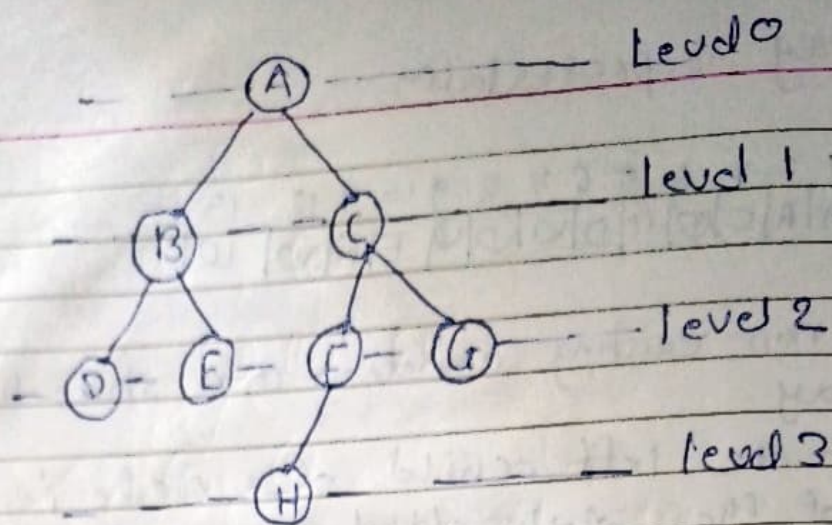
struct node *left;

struct node *right;

} node



A sample binary tree



The level of root node is always '0'.
 The immediate children of the root are at level 1 and their immediate children at level 2 and so on.

Height of a node

Height of a node is the distance of the node from its farthest descendant.

Height of the node C is 2 as its distance from the descendant 'H' is 2.

Nodes	Height
A	3
C	2
B, F	1
D, E, H, G	0

Height of a tree - Height of tree is a height of a root node.

Height of a root node 'A' is 3.

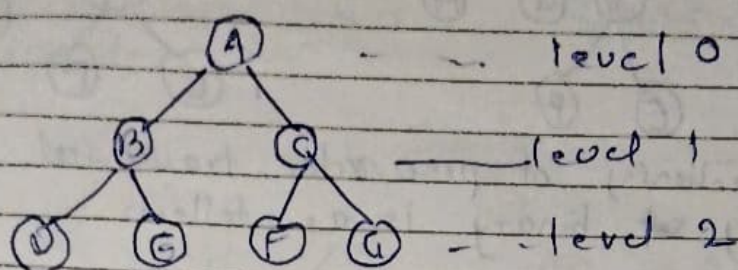
Binary Search Tree

Binary search tree is also known as ordered binary tree.

Complete Binary Tree

A binary tree is said to be a complete binary tree if each of its node has either

two children or no child at all. The number of nodes at any level 'i' in a complete binary tree is given by 2^i . A complete binary tree is shown in fig.



The height of a tree is '2'. The number of nodes is given by $2^{h+1} - 1$.

$$\therefore \text{No. of nodes} = 2^{h+1} - 1$$

Taking log both sides

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

$$\therefore h = (\log_2(n+1)) - 1$$

Binary tree traversal

To perform operations on tree it is necessary to traverse a tree in a particular order. Traversing a tree is a process of visiting every node of the tree exactly once.

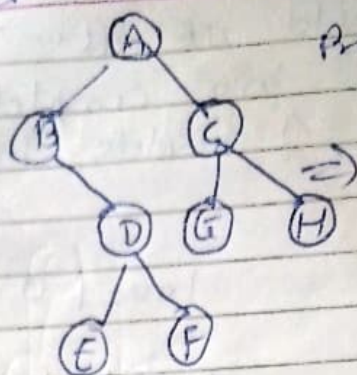
Since a binary tree is defined in a recursive manner, tree traversal too could be defined recursively.

1. Visit root, left, right (Subtree)
2. Visit left, root, right (Subtree)
3. Visit left, right, root

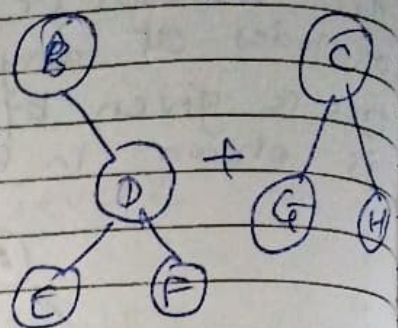
These three techniques are known as preorder, inorder, and postorder traversal.

2.7-1 Preorder Traversal

Pre-order on



Preorder on



The functionality of preorder traversal of a non-empty set binary is as follows

1. Firstly visit the root
2. Next traverse the left subtree in preorder
2. At last traverse the right subtree in preorder

$$\Rightarrow A + B + \text{Preorder on } \begin{array}{c} D \\ \swarrow \searrow \\ E \quad F \end{array} + C + \text{Preorder on } \begin{array}{c} G \\ \swarrow \searrow \\ H \end{array}$$

$$\Rightarrow A + (B + (D + \text{Preorder on } (E) + \text{Preorder on } (F))) + C + H$$

$$\Rightarrow A + (B + (D + (E + F))) + (C + H)$$

$$\Rightarrow ABDEFCH$$

C. function

void preorder (node *T) // Address of the root node is passed in T

{ if (T != NULL)

{ printf("\n %d", T->data);

preorder (T->left);

preorder (T->right);

}

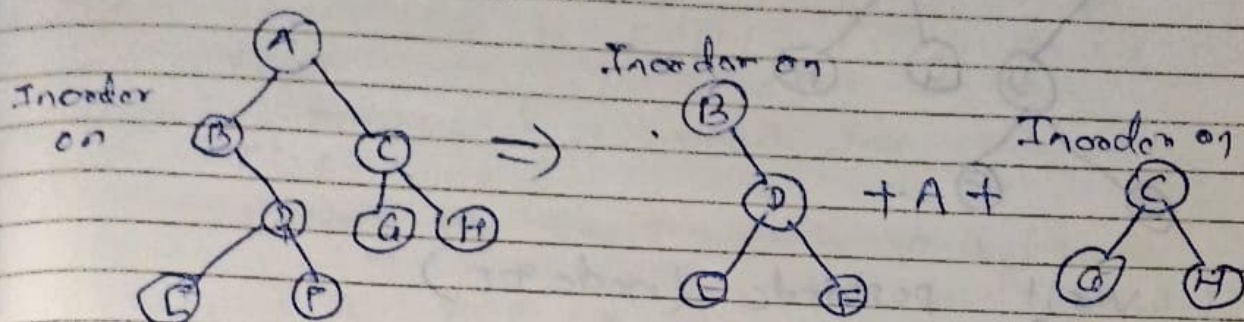
}

// Visit the right

Inorder traversal

The functioning of inorder traversal of a non-empty binary tree is as follows

1. traverse the left subtree in order
2. Next visit the root node
3. At last traverse the right subtree in order



$$= B + \text{inorder on } (D) + A + (\text{inorder on } (G)) + C + (\text{inorder on } (H))$$

$$= (B + \text{inorder on } (E)) + D + \text{inorder on } (F) + A + G C H$$

← B E D F A G H

```
void inorder (node *T)
```

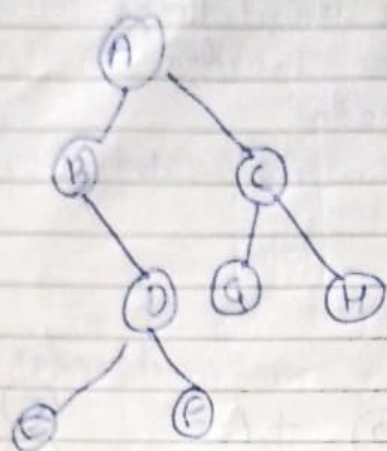
```
{
    if (T != NULL)
    {
        inorder (T->left);
        printf ("\n%.d", T->data);
        inorder (T->right);
    }
}
```

Post order traversal

The functioning of postorder traversal of a non-empty binary tree is as follows

1. traverse the left subtree in postorder

2. traverse the right subtree in post order
3. At last visit the root node



$\Rightarrow$ EFDBGHC A

```

visit postorder (node & r)
{
  if (T != NULL)
  {
    postorder (T->left)
    postorder (T->right)
    printf("\n %d", T->data);
  }
}

```

Non-recursive postorder traversal
Recursion can be removed using stack.

Step 1: start traversing from the root (say T);
traverse left and continue traversing left

All nodes traversed are pushed into stack (say S)
(Nodes are saved in stack for the purpose of traversing back i.e. backtracking)

i.e. traversal of right subtrees of all nodes visited so far)


```
while (T != NULL)
```

```
{ visit (T);
```

```
  push (Ps, T);
```

```
  T = T → left;
```

```
}
```

If stack 'S' is empty
then

traversal is finished

else

[visit the right subtree of the node
T = pop (Ps); ^{popped from the stack}

T = T → right;

Start traversing from T, traverse left
and continue traversing left
all nodes traversed are pushed into
stack 'S'

```
while (T != NULL)
```

```
{
```

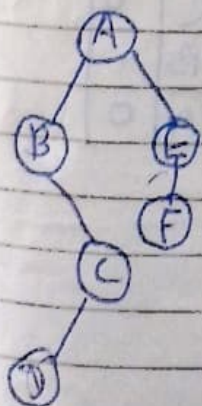
```
  visit (T)
```

```
  push (Ps, T);
```

```
  T = T → left;
```

```
}
```

Simulate the working of non recursive
postorder traversal on the following tree.



Initially stack will be
empty

Start traversing from
the root.

step 1: Start traversing from the root.

Every node encountered is pushed onto the stack with flag set to 0

goto node A

A	0

node flag

goto B

B	0
A	0

node flag

step 2: Pop an element from stack

Node B will be popped. Since flag is equal to '0'. It is pushed back with flag = '1'. Now we will traverse the right subtree of 'B'.

B	1
A	0

step 3 Start traversing from the root of the subtree rooted at 'C'.
Now traverse right subtree of 'B'.

B	1
A	0

goto C

C	0
B	1
A	0

goto D

D	0
C	0
B	1
A	0

D	1
C	0
B	1
A	0

Start node node 1 is 0

C	0
B	1
A	0

Go to right of node C

C	1
B	1
A	0

Visit C

B	1
A	0

Visit B

A	0
---	---

Go to the right of node B

F	0
E	0
A	1

Go to the right of E

Visit A

F	1
E	0
A	1

C	0
A	1

Go to right of C

Visit C

E	1
A	1

A	1
---	---

Visit A

--	--

stack is empty indicates end of the process

Binary search tree

A binary search tree is a binary tree which is either empty or in which each node contains a key that satisfies the following conditions

1. All keys are distinct.
2. For every node X in the tree, the values of all the keys in its left subtree are smaller than the key values in X .
3. For every node X , in the tree the values of all the keys in its right subtree are larger than the key value in X .

Create a binary search tree

A binary search tree can be created by making repeated calls to insert operation

Insert (Tree, Val).

step 1: if Tree = NULL

Allocate memory for tree

set $T \rightarrow \text{data} = \text{Val}$

set $T \rightarrow \text{left} = \text{Tree} \rightarrow \text{right} = \text{NULL}$

else

if $\text{Val} < \text{tree} \rightarrow \text{data}$

insert ($T \rightarrow \text{left}$, Val);

else

insert ($\text{tree} \rightarrow \text{right}$, Val);

end of if

step 2: end

Created :

step 1 : Create a new node

root = new node

root $\rightarrow$ left = NULL

root $\rightarrow$ right = NULL

Algorithm to search for a give value

search element (tree, val)

step 1: IF tree $\rightarrow$ data = Val or tree = NULL:

Return tree

else

if val < tree $\rightarrow$ data

Return search element (tree $\rightarrow$ left, val)

else

Return search element (tree $\rightarrow$ right, val)

end of if

end of if

step 2 end.

Insert a given value

IF tree = NULL

Allocate memory for tree

Set tree $\rightarrow$ data = Val

Set tree $\rightarrow$ left = tree $\rightarrow$ right = NULL

else

if val < tree $\rightarrow$ data

insert (tree $\rightarrow$ left, val)

else

insert (tree $\rightarrow$ right, val)

end of if

end of if

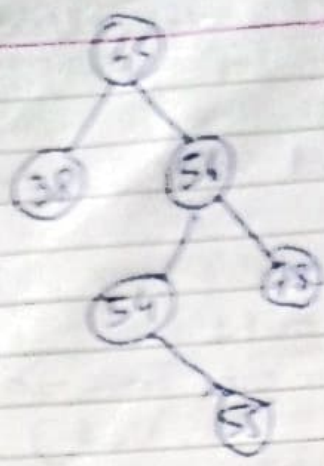
step 2: end

Delete

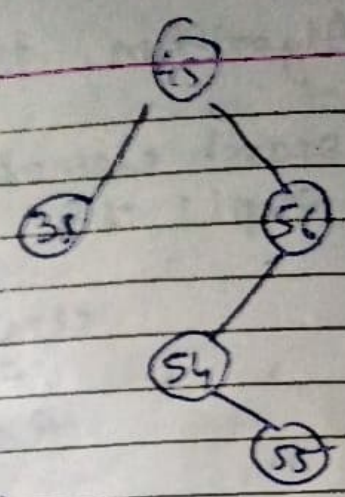
- Deleting a node with no child

Deleting a node with one child

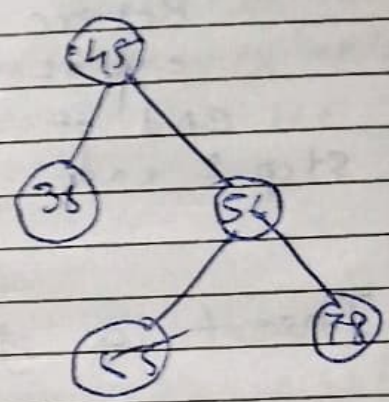
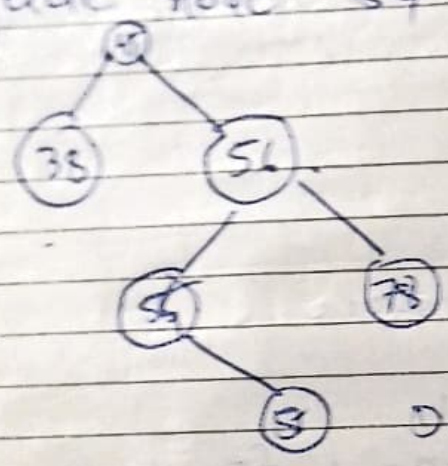
Deleting a node with two children



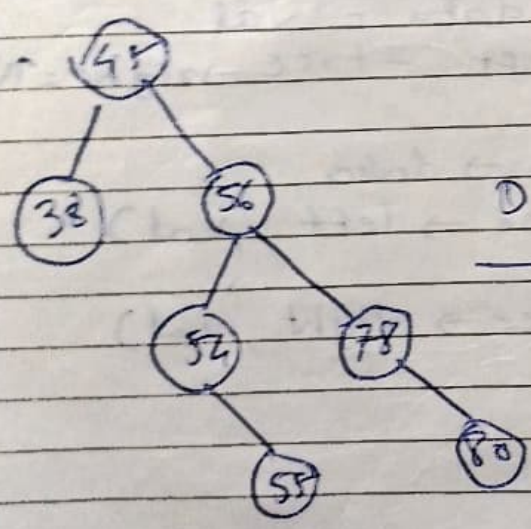
Delete node 78



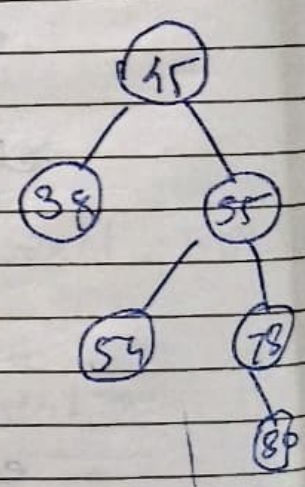
Delete node 54



Delete 54

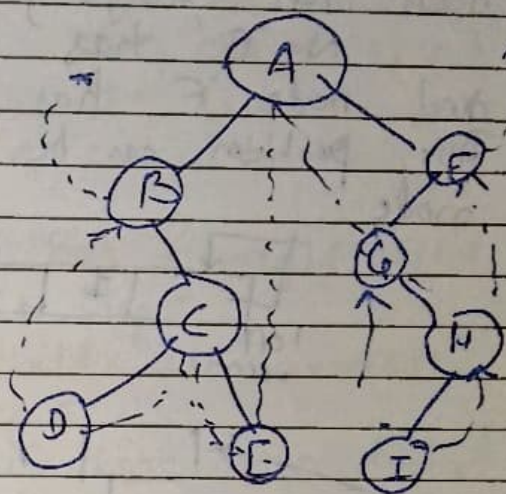
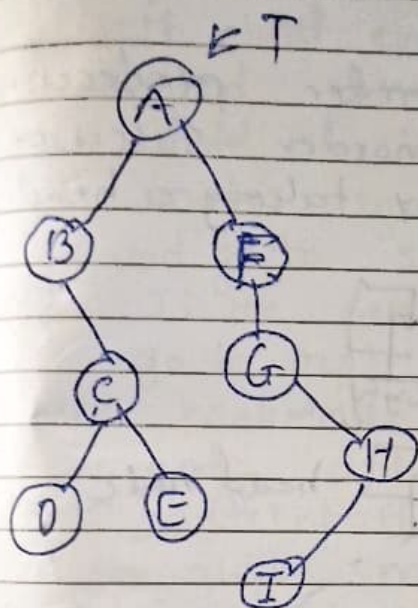


Delete node 56



Threaded binary tree

In a linked representation of a binary tree there are more null links than actual pointers. These null links can be replaced by pointers called threads to other nodes. The left null link of a node is replaced with address of its in-order predecessor. Similarly a right null link is replaced with address of in-order successor.



Simple tree

The above tree has '9' nodes and '10' NULL links, which have been replaced by threaded links.

If we traverse a tree in-order the nodes will be visited as:

B D C E A G H I F

Consider a node 'D'. The left null link of 'D' is replaced with link pointing to its in-order predecessor. The right NULL link will be replaced with thread link pointing to its in-order successor.

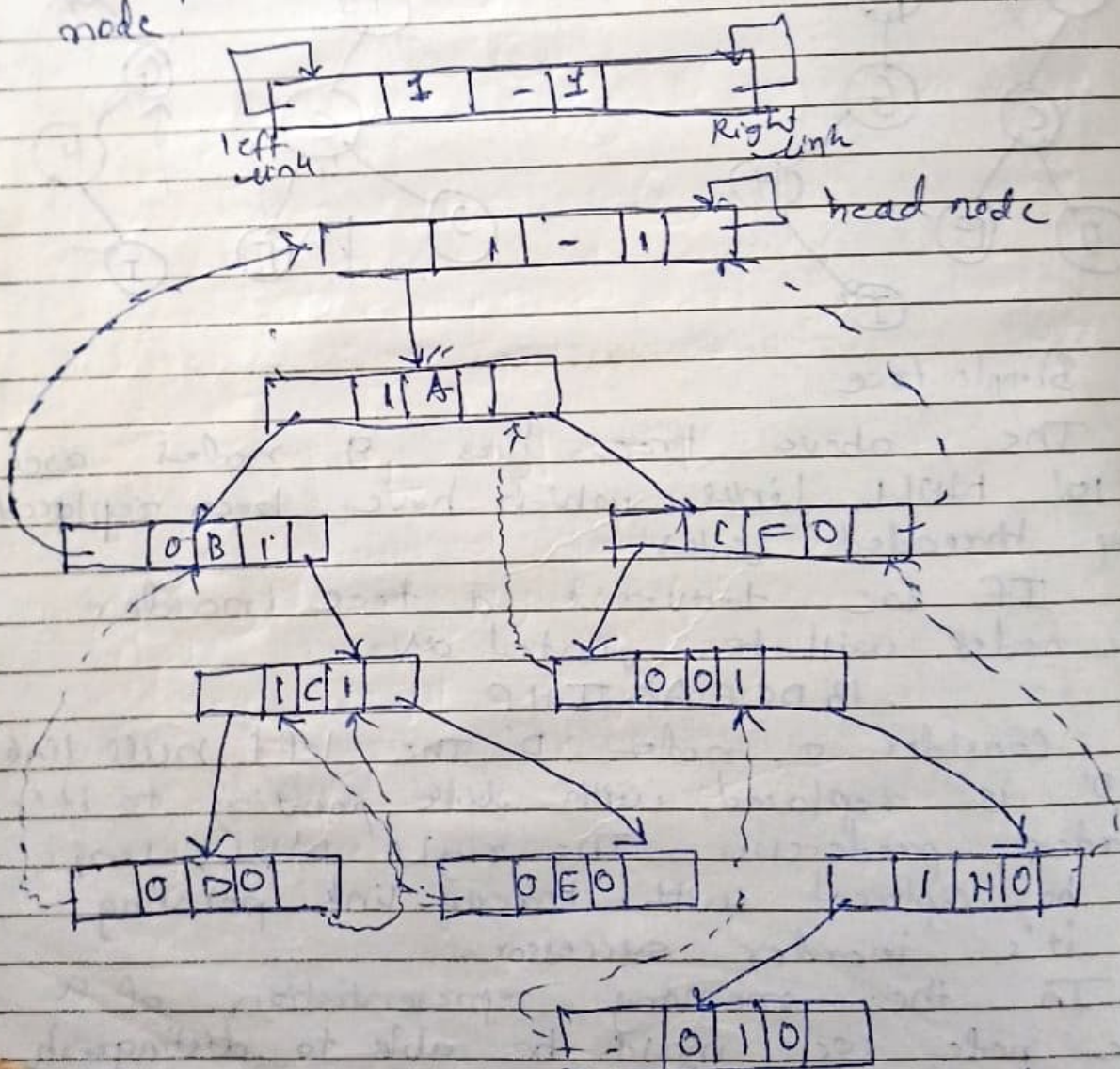
In the memory representation of a tree node we must be able to distinguish between threads and normal pointers. Thus

can be done by adding two extra fields
 1 bit and 1 bit

1 bit of a node = 1 left child is normal
 1 bit of a node = 0 left link is replaced with thread

1 bit of a node = 1 right child is normal
 1 bit of a node = 0 right link is replaced with thread

In the above fig. two threads have been left dangling
 No 'B' has no inorder predecessor and node 'F' has no inorder successor.
 This problem can be solved by taking a head node.



Memory organization of a threaded binary tree.

struct node

{

char data;

struct TBTnode *left, *right

int lbit, rbit;

}

Algorithm for inorder traversal of TBT

- step 1: Check if the current node has a left child that has not been visited. If a left child exists that has not been visited go to step 2 else go to step 3
- step 2: Add the left child in the list of visited nodes make it as a current node and go to step 6.
- step 3: If the current node has a right child go to step 4 else go to step 5
- step 4: make the right child as current node and go to step 6
- step 5: Print the node if there is a threaded node make it the current node
- step 6: If all the nodes have visited then end also go to Step 1.

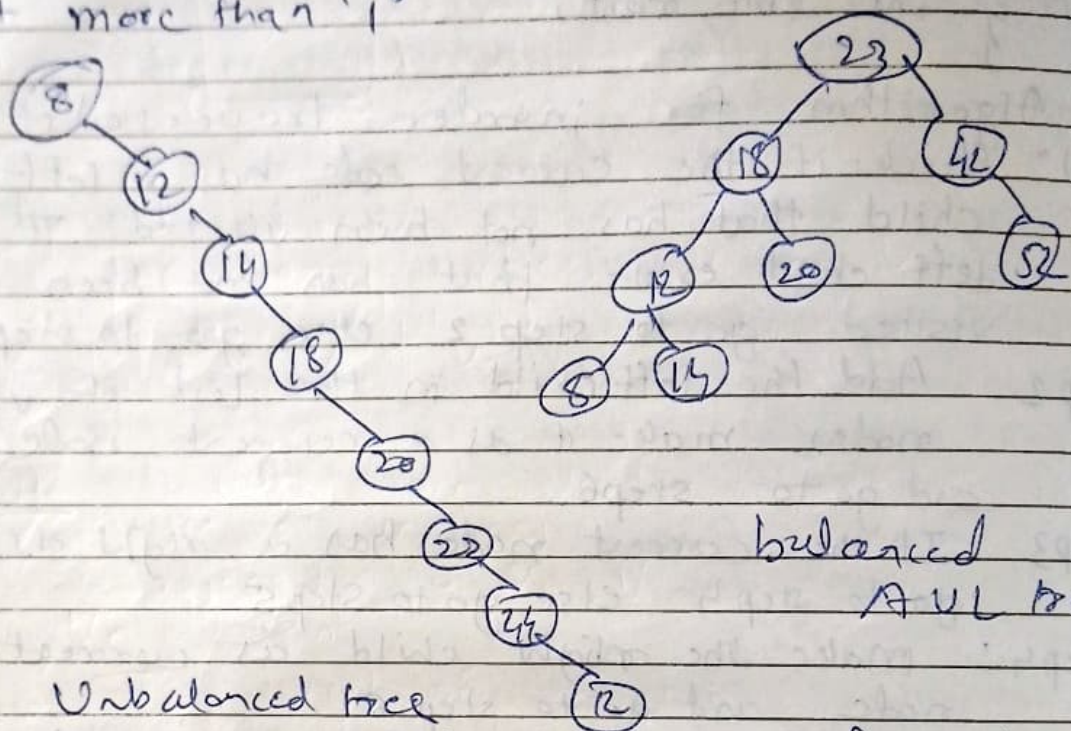
Advantages

- It enables linear traversal of elements in the tree.
- Linear traversal eliminates the use of stacks which consumed memory space and computer time.
- It enables to find the parent of a given node without explicit use of parent pointers.
- Since nodes contain pointers to inorder predecessor and successor the threaded tree enables forward & backward traversal of the nodes as given by inorder fashion.

AVL trees

It is a self balancing binary search tree

AVL tree is a balanced binary structure. In this tree height of the subtree differ not more than '1'.



Unbalanced tree

balanced
AVL tree

An AVL tree is a binary tree that either is empty or consists of two AVL subtrees T_L & T_R whose heights differ by no more than '1'.

$$|H_L - H_R| \leq 1$$

H_L is height of left subtree and H_R is height of right subtree.

AVL trees are balanced by working with their height. They are known as height balanced trees.

An AVL tree is a height balanced binary search tree.

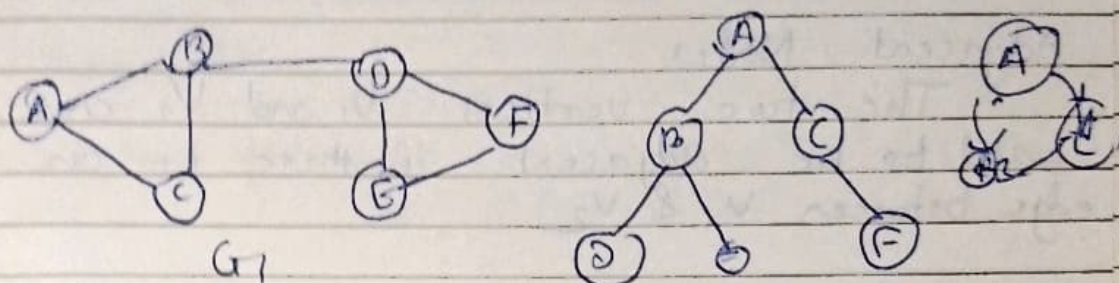
Graphs

A Graph 'G' is a set of vertices 'V' and set of edges 'E'. The set 'V' is a finite non empty set of vertices. The set 'E' is set of pair of vertices representing edges.

$$G = (V, E)$$

$V(G)$ = vertices of Graph G

$E(G)$ = Edges of graph 'G'



$$V(G_1) = \{A, B, C, D, E, F\}$$

$$E(G_1) = \{(A, B), (A, C), (B, C), (C, D), (D, E), (D, F), (E, F)\}$$

A Graph containing unordered pairs of vertices is called an undirected graph.

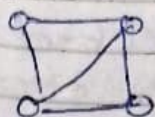
In an undirected graph pair of vertices (A, B) and (B, A) represent the same edge.

A Graph with ordered pairs of vertices is called as directed graph.

A complete Graph

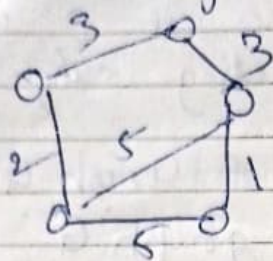
An undirected graph, in which every vertex has an edge to all other vertices is called a complete graph.

For a complete graph with N vertices has $\frac{N(N-1)}{2}$ edges



Weighted Graph

A weighted graph is a graph in which edges are assigned some value.



Adjacent Nodes

The two vertices V_1 and V_2 are said to be adjacent if there is an edge between V_1 & V_2 .

Path - A path from vertex V_0 to V_n is a sequence of vertices $V_0 - V_1 - \dots - V_n$. V_0 is adjacent to V_1 , V_1 is adjacent to V_2 , ..., V_{n-1} to V_n .

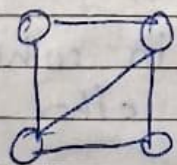
The length of the path is number of edges on the path.

A path with n vertices has length $n-1$.

The path is simple if the first and last vertices are distinct.

Cycle -

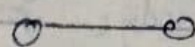
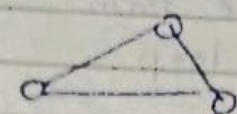
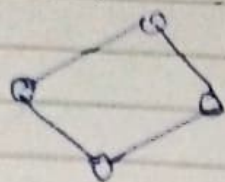
A cycle is a simple path that begins and ends at the same vertex.



Connected Graph

A Graph is said to be connected if there exists a path between every

of vertices v_i and v_j



A Graph containing 3 connected components

The graph shown in figure is a connected graph

Subgraph

A subgraph of 'G' is a graph G_1 such that $V(G_1)$ is a subset of $V(G)$ and $E(G_1)$ is a subset of $E(G)$

Component

A component 'H' of an undirected graph is a maximal connected subgraph.

Degree of a Vertex

The total number of edges linked to a vertex is called its degree.

The total no. of edges coming to that node is called indegree of a node.

Total number of edges going out from a node is called outdegree of a node.

A vertex which has only outgoing edge and no incoming edge is called as source.

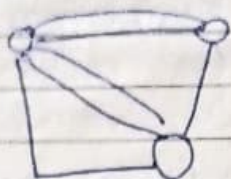
A vertex having only incoming edge and no outgoing edge is called as sink. When indegree is one and outdegree is '0' the vertex is called as pendant node.

Self edge or self loop.

An edge of the form $(v_i v_i)$ is called self edge or self loop.

Multigraph

A graph with multiple occurrences of the same edge is known as multigraph.



Tree - Tree is a connected graph without any cycle.

Tree is a special case of graph. Tree can be treated as a graph. It need not to have special vertex as root.

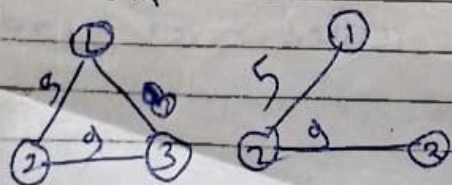
Spanning tree

A spanning tree is a graph $G=(V, E)$ is a subgraph of G having all vertices of G and no cycles in it.

If Graph ' G ' is not connected then there is ~~no~~ spanning tree of the graph.

Minimal Spanning tree

The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a Graph $G=(V, E)$ is called a minimal cost spanning tree or simply the minimal spanning tree of ' G ' if its cost is minimum.



$G \rightarrow$ Sample Weighted Graph
 $T \rightarrow$ Spanning tree of G with Cost
 $5+9 = 14$

Draw possible trees and mention weight.

Applications of spanning tree

Routing of a packet in a network

A node can represent a router located in a city and a link between routers can be represented using an edge.

A spanning tree can represent a network with the minimum number of links.

- Electrical network - A spanning tree can be used to obtain an independent set of circuit equations for an electric network.

Representation of Graphs

The adjacency matrix

A two dimensional matrix. A graph $G = (V, E)$ where $V = \{0, 1, 2, \dots, n-1\}$ can be represented using a two-dimensional integer array of size $n \times n$.

`int adj[20][20];` can be used to store a graph with 20 vertices.

$adj[i][j] = 1$ indicates presence of an edge between two vertices i and j
 $= 0$ indicates absence of an edge between i and j

A graph is represented using square matrix.

Adjacency matrix of an undirected graph is always a symmetric matrix.

Adjacency matrix of a directed graph is never symmetric.

for a weighted graph ad
 instead of 1, the weights are represented
 at (i, j) location

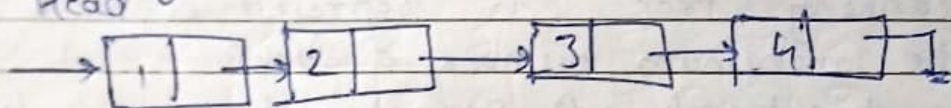
Adjacency List

A graph can be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list.

It creates a separate linked list for each vertex V_i in the Graph $G = (V, E)$

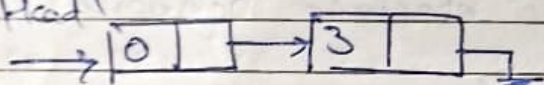
List of adjacent to vertex 0

Head 0



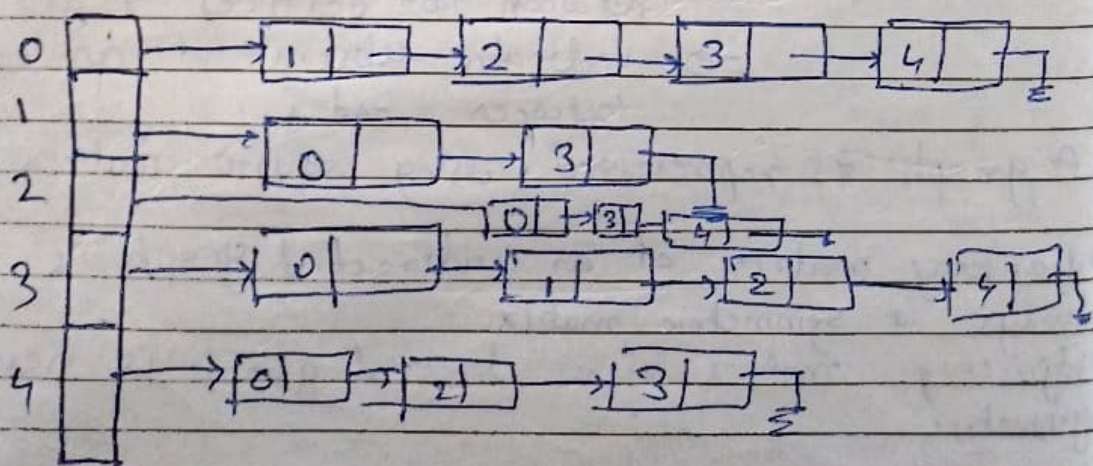
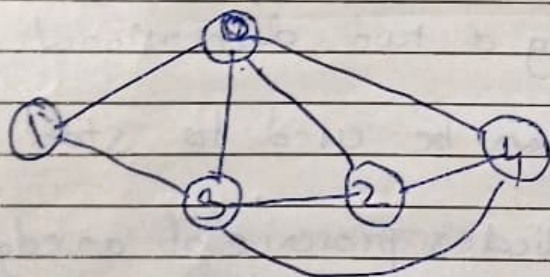
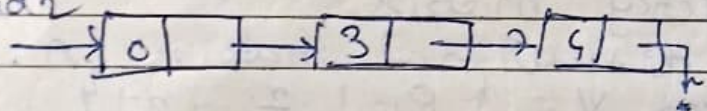
List of adjacent vertices to vertex 1

Head 1



List of adjacent vertices to vertex 2

Head 2



Traversal of Graphs

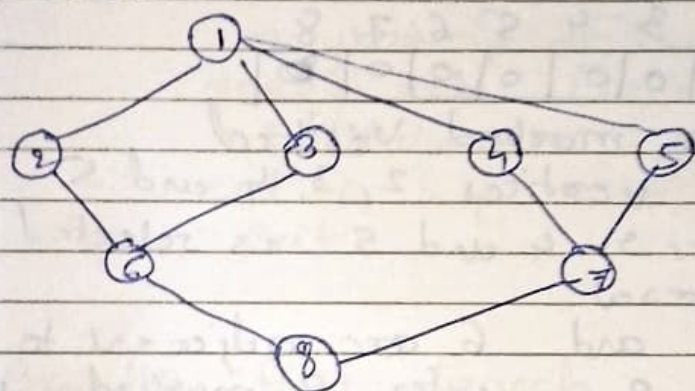
Traversing a graph means visiting each node exactly once. Two commonly used techniques are

i) DFS - Depth first Search

ii) Breadth first search

DFS

It is like preorder traversal of a tree. Traversal can start from any vertex say V_i . V_i is visited then all the vertices adjacent to V_i are visited recursively using DFS.



at three goes
back to 6
(DFS) predecessor
of 3

visit (1)

DFS (G, 2)

DFS (G, 3)

DFS (G, 4)

DFS (G, 5)

} all adjacent to 1

Since a graph can have cycles avoid revisiting a node.

When a vertex V_i is visited mark it visited.

A node that has already been marked as visited should not be selected for traversal.

Marking of visited vertices can be done with the help of a global array `visited[]`. initialise it visited ~~false~~ as false i.e. '0'.

Algorithm DFS

$n \leftarrow$ Number of nodes

i) Initialize visited [] to false

for $(i = 0; i < n; i++)$

visited [i] = 0;

void DFS (vertex i)

{

visited[i] = 1;

for each w adjacent to i

if (!visited [w])

DFS (w);

}

1 2 3 4 5 6 7 8

visited

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

vertex 1 is marked visited

out of adjacent vertices 2, 3, 4 and 5

vertex number 2, 3, 4 and 5 is selected for further traversal

vertices 1 and 6 are adjacent to

vertex 2 vertex 2 marked

1 2 3 4 5 6 7 8

1	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---

visited

then 6 is selected for further traversal

for 6, vertex 3, 2, and 8 are adjacent

2 is already used

or visited 3 is used for

further expansion

1 2 3 4 5 6 7 8

1	1	1	0	0	1	0	0
---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8

1	1	1	0	0	1	0	1
---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8

1	1	1	0	0	1	1	1
---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

visited

traversal sequence

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

Step 1: Set status = 1 (for each node in G)

Step 2: Push the source node in A on the stack and set its status = 2 (waiting state)

Step 3: Repeat step 4 and 5 until stack is empty

Step 4: Pop the top Node 'N' process it and set its status = 3 (processed state)

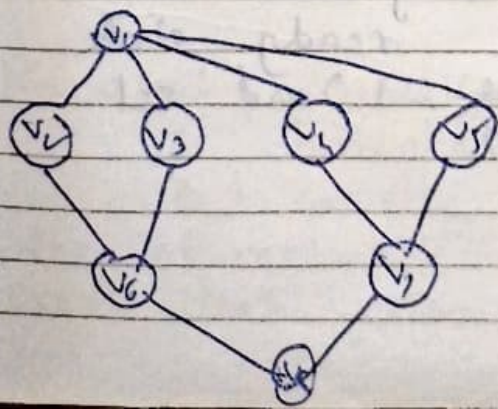
Step 5: Push on the stack all the neighbours of 'N' that are in the ready state (whose status = 1) and set their status = 2 (waiting state)
end of loop

Step 6: Exit

Breadth First Search

Begins at source node and explores all neighbouring nodes then for each of these neighbouring nodes explore unexplored neighbour nodes until it finds goal

- Initially a queue contains the start vertex
- In every iteration a vertex is removed & its queue of adjacent vertices which are not visited are added to queue
- The algo terminates when queue becomes empty



* Array visited[] initialize to 0 *

void BFS(int v)

{ q: a queue type variable

Initialize q;

visited[v] = 1; /Mark v as visited

add the vertex v to queue q;

while (q is not empty)

{

v ← delete an element from queue
for all vertices w adjacent to v

{ if (!visited[w])

{

visited[w] = 1;

add vertex w to q;

}

}

}

Step 1: set status = 1 (ready state)

for each node in G

Step 2: Enqueue the starting node 'A' and
set its status = 2 (wait state)

Step 3: Repeat steps 4 and 5 until queue
is empty

Step 4: Dequeue a node N. Process it and
set its status = 3 (processed state)

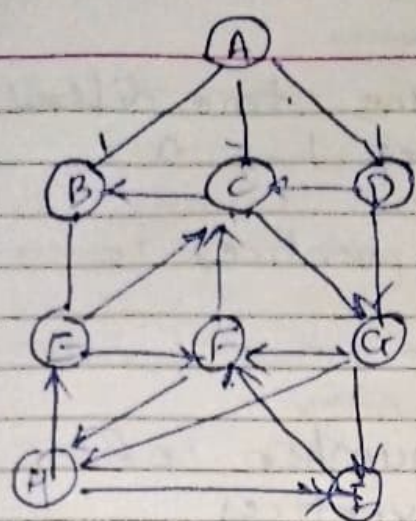
Step 5: Enqueue all the neighbours of N
that are in the ready state

(whose status = 1) and set
their status = 2

(waiting state)

END OF LOOP

Step 6: Exit



Adjacency list-

A, B, C, D
 B, E
 C, B, D, F, G
 D, C, G
 E, C, F
 F, C, G, H, I
 G, F, H, I
 H, E, I
 I, F

Minimum cost Spanning tree

Prim's algorithm

Let $G = (V, E)$ be a graph with n vertices
 step 1: Choose a vertex u_1 of G

Let $V_T = \{u_1\}$ and $E_T = \{\}$

Step 2: Choose nearest neighbour v_j of u_1 that is adjacent to u_1 $v_j \in V_T$ for which edge (u_1, v_j) does not form a cycle with members of E_T

Add u_i to V_T and add (u_i, v_j) to E_T

step 3: Repeat step 2 until $|E_T| = n - 1$
 then V_T contains all n vertices of G and E_T contains the edges of the minimum cost spanning tree of G

Kruskal's algorithm

In Kruskal's algorithm edges are added to the spanning tree in increasing order of cost. If the edge forms a cycle then that edge is discarded

Time complexity discussion

Prim's algorithm contains two different nested loops. Each of these loops has a complexity of $O(n)$.

∴ The complexity with n vertices is $O(n^2)$.

Kruskal's algorithm

A graph with ' e ' number of edges and ' n ' number of vertices.

Kruskal's algorithm sorts number of edges.

Complexity of sorting = $O(e \log e)$

In Kruskal's algorithm an edge is added to the spanning tree in each iteration.

This involves merging of two components.

Time complexity of merge two components = $O(\log n)$

Overall complexity = $O(e \log e) + O(e \log n)$

For a dense graph $O(e \log n)$ may be worse than $O(n^2)$.

∴ For dense graph Kruskal's algo should be avoided.

Sorting

Bubble Sort

Sorting is the process of arranging the elements of a list in some order either ascending or descending.

Internal sorting - Sorting the data stored in computer's memory.

External sorting - Deals with sorting the data stored in files.

Bubble sort

Bubble sort is a simple method of sorting

Step 1: Repeat step 1 for $i = 0$ to $N-1$

Step 2: Repeat for $J = 0$ to $N-1$

Step 3: IF $A[J] > A[J+1]$
SWAP $A[J]$ and $A[J+1]$

end of inner loop

end of outer loop

Step 4: exit

Complexity of any sorting algorithm depends on complexity of number of comparisons.

In bubble sort there are $N-1$ passes in total.

In first pass $N-1$ comparisons are made to place the highest element in correct position.

In pass 2 again $N-2$ comparisons and the 2nd highest element is placed in its position.

$\therefore$ total no. of comparisons

$$f(n) = (n-1) + (n-2) + (n-3) \dots 3+2+1$$

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

$$f(n) = \frac{n^2}{2} + O(n)$$

$$= O(n^2)$$

Insertion Sort

It is a very simple sorting algorithm in which sorted array or list is built one element at a time.

Insertion Sort working

- The list is divided into two sets one that stores sorted values and other that stores unsorted values.
- The sorting will proceed until there are no elements in the unsorted list.
- Suppose there are 'n' elements in a array.

Initially the element with index 0 is in the sorted set. Rest of the elements

① are in sorted set.

39 9 45 63, 18 81 108 54 71 36

part 1

39

9	45	63	18	81	108	54	71	36
---	----	----	----	----	-----	----	----	----

63 is the only element in sorted list.

② The first element of the unsorted portion has array index 1.

- During each iteration of the algorithm the first element in the unsorted set is picked up and inserted into correct position in the sorted set.

pass 2 9 39 45 63 18 81 108 54 71 36

pass 3 9 39 45 63 18 81 108 54 71 36

pass 4 9 18 39 45 63 81 108 54 71 36

pass 5 9 18 39 45 63 81 108 54 71 36

pass 6 9 18 39 45 63 81 108 54 71 36

pass 7 9 18 39 45 54 63 81 108 12 36

pass 8 9 18 39 45 54 63 71 81 108 36

pass 9 9 18 36 39 45 54 63 71 81 108

step 1 : Repeat steps 2 to 5 for $k=1$ to $N-1$

step 2 : Set $temp = ARR[k]$

step 3 : Set $j = k-1$;

step 4 : Repeat while $temp <= ARR[j]$
 Set $ARR[j+1] = ARR[j]$
 Set $j = j-1$
 end of loop

step 5 : Set $ARR[j+1] = temp$;
 end of loop

step 6 exit

Complexity

In best case the list is already sorted the algo has linear time i.e $O(n)$

because the first element from unsorted list is compared only with last element of a sorted list.

The worst case is when the list is sorted in reverse order.

The first element of unsorted set has to be compared with every element in sorted set.

Every iteration of the inner loop will have to be shifted the elements of the sorted set

So in worst case quadratic time complexity $O(n^2)$

Even in average case it has to make $(k-1)/2$ comparisons

Advantages

- Easy to implement
- Efficient for small set of data
- Perform better than bubble sort
- Require less memory space

* Selection Sort

Technique

Consider an array ARR with 'n' elements

First find the smallest value in the array and place it in the first position. Then find the next smallest value and place it in the 2nd position.

Repeat this procedure until the entire array is sorted.

In pass 1 find the position (pos) of the

- Smallest value in the array and then swap $ARR[pos]$ and $ARR[0]$. Thus $ARR[0]$ is sorted.

- In pass 2 find the position pos of the smallest value in sub-array of $N-1$ elements.

swap $ARR[pos]$ with $ARR[1]$

now $ARR[0]$ and $ARR[1]$ is sorted.

- In pass $N-1$ find the position pos of the smallest of the elements $ARR[N-1]$ and $ARR[N-2]$ and $ARR[N-1]$.

Swap $ARR[pos]$ and $ARR[N-2]$

So that $ARR[0], [1] \dots [N-1]$ is sorted.

Smallest

Step 1: initialize set $small = ARR[K]$

Step 2: initialize set $POS = K$

Step 3 Repeat for $J = K+1$ to $N-1$

if $small > ARR[J]$

set $small = arr[J]$

set $POS = J$

end of if

end of loop

Step 4 Return POS

Selection sort(Arr, N)

Step 1: Repeat steps 2 and 3 for $k=1$ to $N-1$

Step 2: call $smallest(Arr, K, N, POS)$

Step 3: swap $A[K]$ with $ARR[POS]$
end of loop

Step 4: exit

Complexity

In first pass $N-1$ comparisons are required to find the smallest element.

Thus the smallest element is swapped with 1st element. In pass 2 selecting 2nd smallest element scanning $n-1$ elements

$$\therefore (n-1) + (n-2) \dots 2 + 1$$

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

Simple way to implement

Suitable for small data set

60% more efficient than bubble sort

Heaps

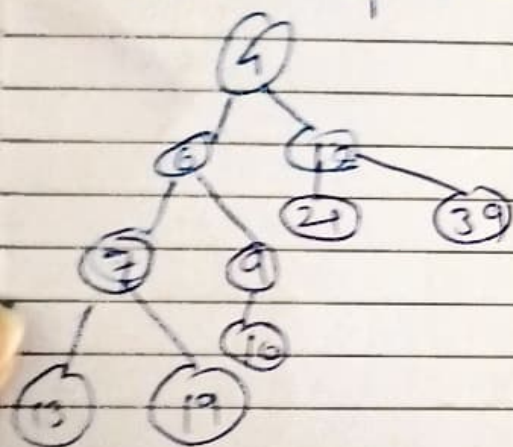
A binary heap is a complete binary tree in which every node satisfies the heap property which states that

If 'B' is a child of A

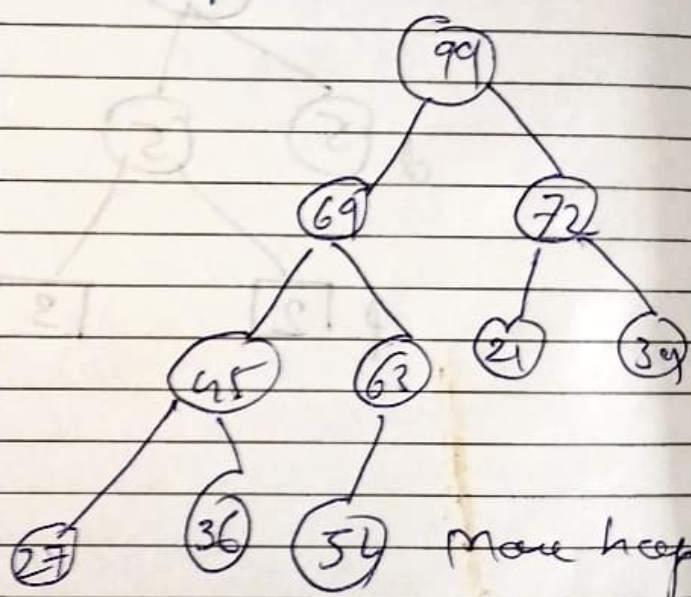
then $\text{key}(A) \geq \text{key}(B)$

i.e. the elements of every node will be either greater than or equal to the element at its left and right child. Thus the root node has the highest key value in the heap.

Such heap is max heap



Min Heap



max heap

Heap is a complete binary tree

- If in sequential representation if the element is stored at i^{th} location then its left child is stored at $2i$ and right child at $2i+1$ i.e. the element at position i its parent is stored at location $i/2$

- All the levels except the last level are completely filled

- Height of tree is $\log_2 n$ where n is no. of elements

Algorithm to insert an element in
max heap

Step 1: - Add the new value and set its
POS

Set $N = N + 1$ $POS = N$

Step 2 = Set $Heap[N] = val$

Step 3 = Find the appropriate location
of val

Repeat Step 4 and 5 while $POS > 1$

Step 4 = Set $PAR = POS / 2$

Step 5 = If $Heap[POS] < Heap[PAR]$

Then Go to step 6

Else

swap ($Heap[POS]$, $Heap[PAR]$);

$POS = PAR$

end of if

end of loop

Step 6 = Return

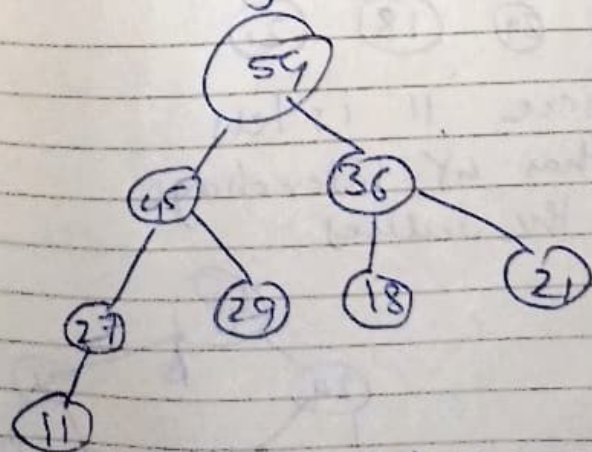
Memory representation

like levelwise in an array

The location of VAL is POS as it
rises in the heap

PAR is the location of parent

* Deleting a node from Heap



Perform deleting on root

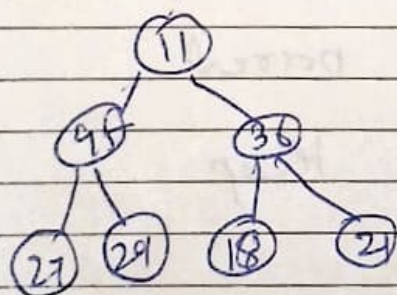
Consider a max heap having 'n' elements. An element is always deleted from root of the heap so following are the steps

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap
2. Delete the last node
3. Sink down the new root node's value so that 'H' satisfies the heap property.

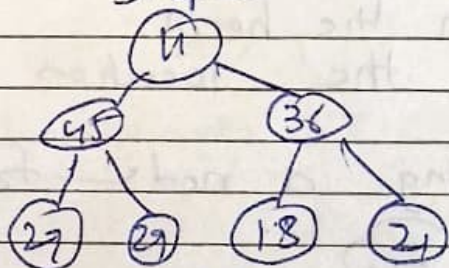
In this case interchange the root node's value with the child node's value (whichever is largest amongst children)

Here the value of root node is 54 and value of last node is 11. So replace 54 with 11 and delete last node

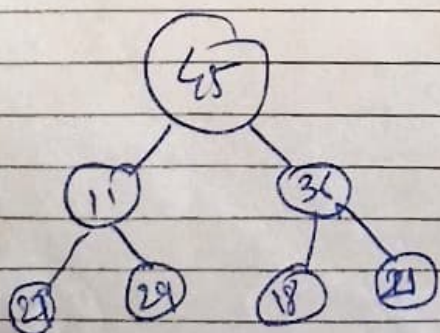
Step 1



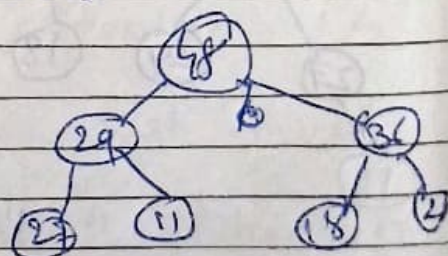
Step 2



Since '11' is less than 45 interchange the values



Since 11 is less than 29 swap



Algorithm

LAST is last element

PTR left and right denote position of its left and its right children

Step 1: Remove the last node from the heap

Set $\text{Last} = \text{HEAP}[N]$, Set $N = N - 1$

Step 2: Initialization

Set $\text{PTR} = 1$ Left = 2 right = 3

Step 3: Set $\text{HEAP}[\text{PTR}] = \text{LAST}$

Step 4: - Repeat step 5 to 7 while $\text{left} \leq N$;

Step 5 = $\text{Heap}(\text{PTR}) \geq \text{Heap}(\text{Left})$
AND

$\text{HEAP}(\text{PTR}) \geq \text{Heap}(\text{right})$

Go to step 8

Step 6: - if $\text{Heap}(\text{right}) < \text{Heap}(\text{left})$
swap ($\text{Heap}(\text{PTR}), \text{Heap}(\text{left})$)

Set $\text{PTR} = \text{left}$;

else

swap ($\text{Heap}(\text{PTR}), \text{Heap}(\text{right})$)

Set $\text{PTR} = \text{right}$;

end of if

Step 7: Set $\text{left} = 2 * \text{PTR}$ and $\text{right} = \text{left} + 1$

end of loop

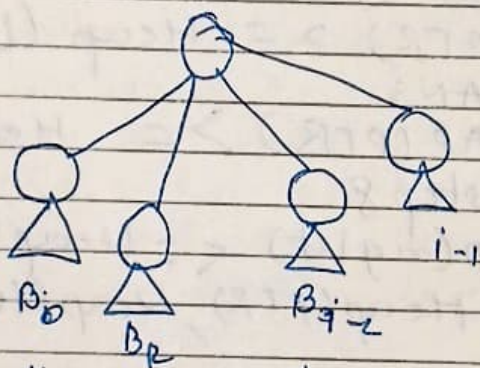
Step 8: Return

Binomial Heaps

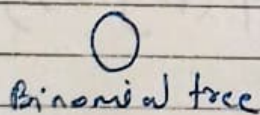
Binomial Heap 'H' is a set of binomial trees that satisfy the binomial heap properties.

Binomial Tree

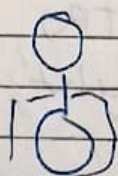
- Ordered tree that can be recursively defined as follows.
 - Binomial tree of order 0 has a single node.
 - A binomial tree of order i has a root node whose children are root nodes of binomial trees of order $i-1, i-2, \dots, 2, 1$ and 0.
 - A binomial tree B_i has 2^i nodes.
 - The height of B_i is i .



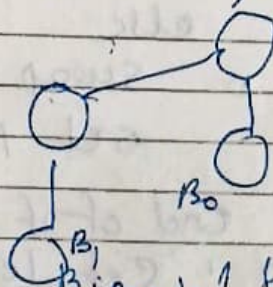
Collection of trees of order $0, 1, 2, \dots, i-1, i-2, \dots, 2, 1, 0$.



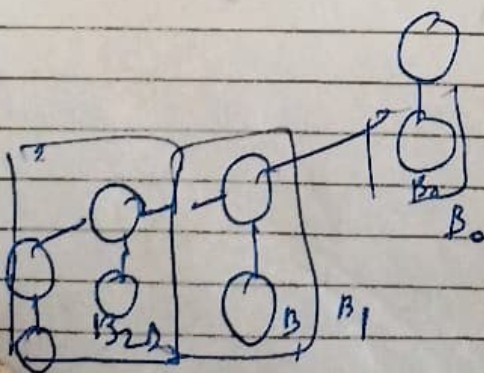
B_0



Binomial tree B_1



Binomial tree B_2



A binomial heap is a collection of binomial trees that satisfy the following properties

- Every binomial tree in H satisfies the min heap property.
- There can be one or more binomial trees for each order including zero order.

* Fibonacci Heaps

Fibonacci heap contains following pointers

- A pointer to its parent
- A pointer to any one of its children.

→ The children of each node are linked together in a circular doubly linked list which is known as the child list of that node.

- Each child v in a child list contains pointers to its left and right siblings.

- If a node x is the only child of its parent then
 $left(x) = right(x) = x$

Heap Sort

Application of Heaps is to provide efficient algorithm for sorting complexity $O(n \log n)$

Two phases

- 1) build the heap using elements of array
- 2) Repeatedly delete the root element of the heap formed in phase 1

Heapsort (arr, N)

Repeat for $I = 0$ to $N - 1$

 call insert-Heap (arr, N, arr[I])

 end of loop

step 2: Repeatedly delete the root element)

Repeat while $N > 0$

 call delete-Heap (arr, N, Val)

 set $N = N - 1$

 end of loop

step 3: end.

* Shell Sort

- Generalization of insertion sort

- The insertion sort works well when input is already almost sorted.

- quite inefficient as insertion sort moves ^{value} only one place

- Shell sort is improvement over insertion sort.

- This enables the element to take bigger steps towards its expected position.

Technique

- Arrange the elements in the form of table and sort the column using bubble sort

- Repeat step 1 each time with smaller number of longer columns each time i.e. at one time there is one column of data to be sorted

63, 19, 7, 90, 83, 36, 54, 45, 72, 27, 22
9, 41, 59, 33

1) 48 63, 19, 7, 90, 83, 36, 54, 45
38 72, 27, 22, 9, 41, 59, 33

2) 63, 19, 7, 9, 41, 36, 33, 45
72, 27, 22, 90, 83, 54

63, 19, 7, 9, 41

3) 36, 33, 45, 72, 27

22, 90, 83, 39, 54

22, 19, 7, 9, 27

36, 33, 45, 72, 41

63, 90, 83, 72, 54

22, 19, 7, 9, 27, 36, 33, 45

31, 41, 63, 90, 83, 72, 54

22, 19, 7, 9

shell_sort (arr, n)

step 1: set flag = 1 gap_size = N;

step 2: Repeat step 3 to 6 while
flag = 1 or gap_size > 1

step 3: set flag = 0;

step 4: set gap_size = $(\text{gap_size} + 1) / 2$

step 5: Repeat step 6 for $I = 0$
to $I < (N - \text{gap_size})$

step 6: if $\text{arr}(I + \text{gap_size}) > \text{arr}(I)$
swap ($\text{arr}(I + \text{gap_size})$,
 $\text{arr}(I)$)
set flag = 1

Step 7: end

Algorithmic Strategy

Algorithm design is an important skill in creating a good algorithm. A knowledge of design helps the number of basic design strategies of algorithm is very small

Types of algorithm

1. Divide and Conquer
2. The Greedy method
3. Dynamic programming
4. Backtracking
5. Randomized algorithm

In divide and conquer strategy a problem is divided into subproblems. The subproblem type is same as that of the original one.

These problems are solved recursively. The recursion will terminate at the step where division of a problem into a subproblem is not possible further and they combine the subproblem to form a solution.

- Divide conquer has two steps
1. Divide - Smaller problems solved recursively
 2. The solution to the problem is formed from the solutions to subproblem

Binary Search
Consider the problem of determining whether an element 'x' is present in a list. List is sorted in non-decreasing order.

Division of problems into subproblems = Elements are stored in an array arr . 1 and j are indices of the first and last element respectively

Initially $i = 0$ and j is $n-1$ where n is number of elements

- Element be searched, x , is compared with centre element $a[c]$

$$c = (i+j)/2$$

Now one of three things could happen

1) $x > a[c]$ right half of the element
 2) $x < a[c]$ left half of the element
 3) $x = a[c]$ element selected as subproblem

are selected as subproblem
 i remains unchanged
 $j = c - 1$

!!!) $x = a[c]$ solution is found
 further search for subproblems
 is not required

Combining Subproblems

In this problem only one

subproblem is generated either left or

right half at each level of recursion

The solution of subproblem is solution of

original problem.

The function `bsearch()` has

four inputs `a[]`, `i`, `j`, `x`
`a[]` array of sorted elements
`i` = index of 1st element
`j` = index of last element
`x` = element to be searched

If x is not present -1 is

returned and index of centre element
 is returned if x is present

`bsearch (int a[], int i, int j, int x)`

int `c`;
 if (`i > j`)
 return `-1`;
`c = (i+j)/2`;

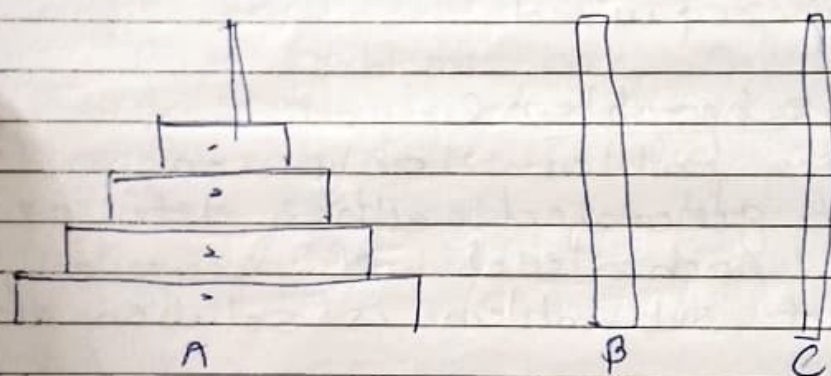

```

if (x == a[c])
    return(c);
if (x > a[c])
    return (binsearch(a, c+1, j, x));
else
    return (binsearch(a, i, c-1, x));

```

Examples

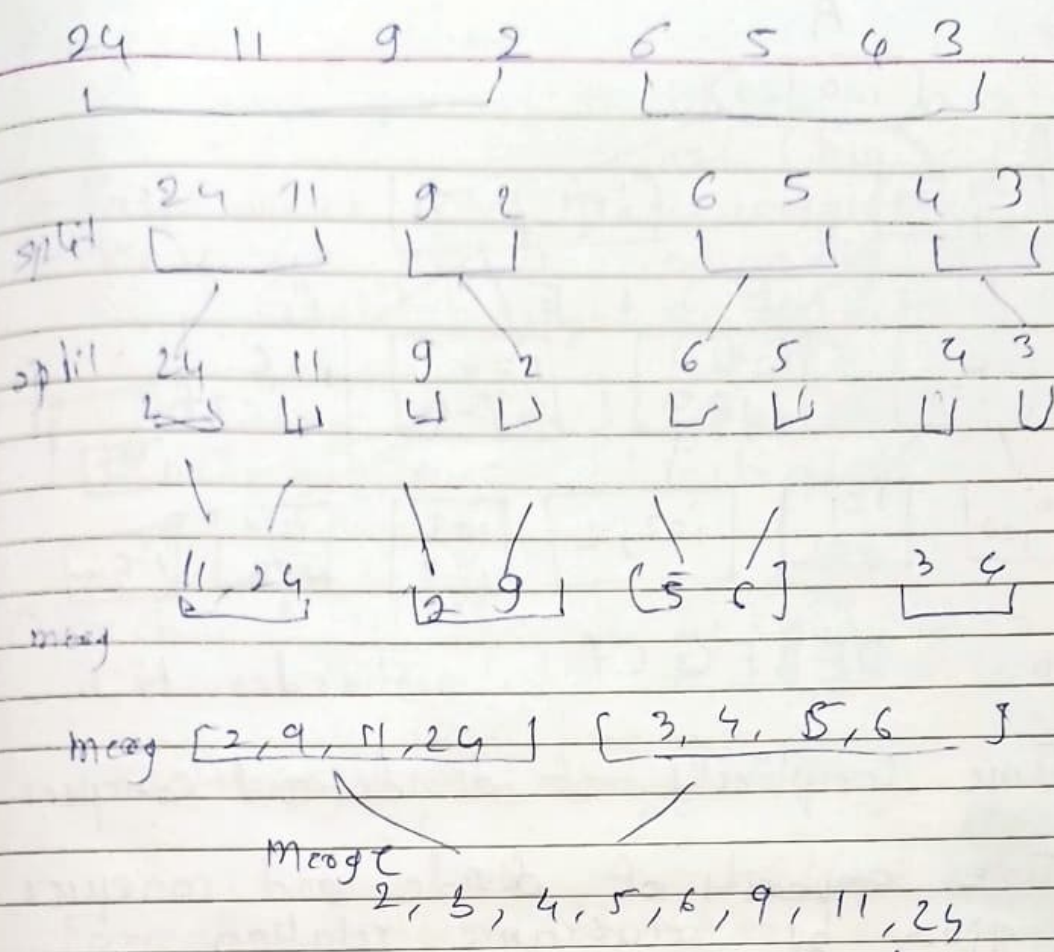
- Tower of Hanoi
- Merge Sort
- Quick Sort
- Finding of Maxima



Disks are stacked on peg A in decreasing order of size

Merge Sort

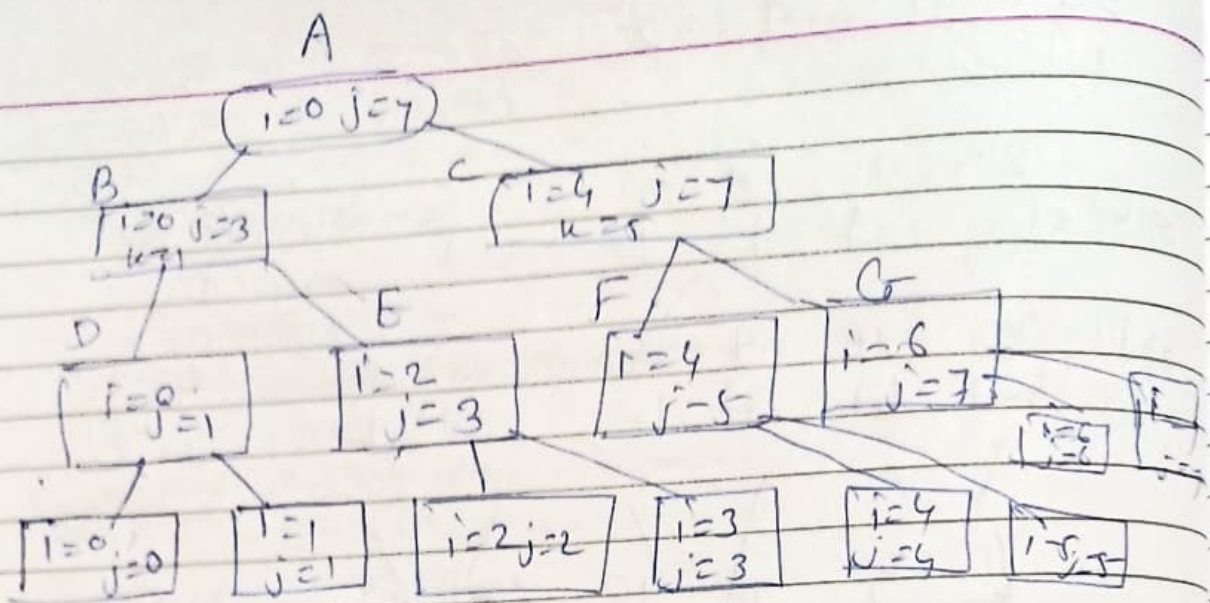
An array of 'n' elements is split around its centre producing two smaller arrays. After that two arrays are sorted independently. They can be merged to produce final sorted array. The process of splitting and merging can be carried out recursively till there is only one element in the array.



Given an array of 8 element
 array index of first element $i = 0$
 index of last element $j = 7$
 Index of centre element

$$k = \frac{i+j}{2} = \frac{0+7}{2} = 3$$

- Merge sort is applied recursively on 1st half of the list from $i = 0$ to $j = 3$
 - After sorting of the left half of the list the right half of the list is sorted from $i = 4$ to $j = 7$ using recursive merge sort.
 - After both the lists are sorted left list from $i = 0$ to $j = 3$ and right list from $i = 4$ to $j = 7$ are merged to produce a sorted array.
- This recursive merge sort can be seen as postorder traversal of a binary tree



DEBFGCA post order traversal

Time complexity of divide and conquer

Time complexity of divide and conquer is given by recurrence relation

$$T(n) = \begin{cases} T(1) & n = 1 \\ a \cdot T(n/b) + f(n) & n > 1 \end{cases}$$

where a and b are constants

$T(1)$ is known constant

and $n = b^k$

$$T(n) = a T(n/b) + f(n)$$

The Greedy Method

The greedy method is simple phase based decision making strategy

- At each phase an input is selected
- The input selected is added to the set of optimal solution
- Selection is made on the basis of some selection procedure
- Inclusion of next input into the partially constructed optimal solution should result in a feasible solution

Knapsack problem

- We are given n objects in a knapsack (bag)
- Each object is of fixed weight
 - Knapsack has a capacity m (total weight of objects used to fill it)
 - There are different types of objects
 - Each type of object gives certain profit

Objective - fill the knapsack with suitable object with maximum profit.

constraint total weight of objects is $\leq m$

Formally a problem can be

Formally

An object 'i' has weight w_i with profit p_i

A fraction x_i [$1 \geq x_i \geq 0$] of object 'i' is used to fill the knapsack

Objective is to maximize $\sum_{i=1}^n p_i(x_i)$

with constraint $\sum_{i=1}^n w_i x_i \leq m$

Greedy Method.

- 1) Select an object with maximum profit per unit weight
ie select the object with largest value of p_i/w_i

Use the object filling up the knapsack. Now one of the two things can happen.

- knapsack is filled to its complete capacity.
- object i is exhausted and some more objects can be added in knapsack.

In case we terminate the algorithm in case (2) we search for another object.

eg Consider 3 types of objects
 $n = 3$

Capacity of knapsack $= m = 20$
 Profit due to objects $= (25, 24, 15)$

$$P_1 = 25$$

$$P_2 = 24$$

$$P_3 = 15$$

Weight of objects $= [18, 15, 10]$

Calculating profit per unit weight

$$r_1 = \frac{25}{18}$$

$$r_2 = \frac{24}{15}$$

$$r_3 = \frac{15}{10}$$

$$\text{Here } \frac{24}{15} > \frac{15}{10} > \frac{25}{18}$$

Select the object 2 for filling up knapsack

$x_2 \geq 1$ (will be object of type 2 for filling knapsack)

~~Step 2~~ / ~~Select $\frac{1}{2}$ objects of type 3 to fill the knapsack~~

Capacity remaining after filling object of type 2 = $20 - 15 = 5$

Step 2: Select $\frac{1}{2}$ objects of type 3 to fill the knapsack

$$x_3 = \frac{1}{2}$$

Step 3 : Since the knapsack is filled to its complete capacity $x_1 = 0$

$$\begin{aligned} \text{Total profit} &= 0 \times 25 + 1 \times 24 \\ &\quad + \frac{1}{2} \times 15 \\ &= \underline{\underline{31.5}} \end{aligned}$$

* Job Sequencing

- We are given a set of n jobs
- With every job i a deadline d_i is attached
- A job i gives a profit of p_i if it is completed by its deadline
- To complete a job one has to process the job on the machine for one unit of time.

Objective - Select a subset of jobs to maximize the profit

Constraint

Selected jobs should be completed by their deadline

Greedy method for selecting a next job to be included in set of jobs selected for execution

- jobs are considered in decreasing order of their profits
- Next job to be considered is one that increases profit the most - subject to constraint of deadline.

Let $n = 4 = (p_1, p_2, p_3, p_4)$ -
with deadlines $= (100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Arrange jobs (p_i, d_i)

$(p_1=100, d_1=2)$ $(p_4=27, d_4=1)$ $(p_3=15, d_3=2)$
 $(p_2=10, d_2=1)$

Let J be set of jobs with optimal solution.

Initially $J = \emptyset$

step 1: Job 1 is selected

$J = \{1\}$ with profit 100

step job 4 is selected

$J = \{1, 4\}$ with profit 127

step 3

'Job 3' is selected but '3'

is not feasible solution

step 4 Job 2 is selected but

discarded as $\{1, 2, 4\}$ is not feasible solⁿ

Thus $J = (1, 4)$ with profit 127 and sequence $1, 1$

Dynamic Programming

Dynamic programming is based on non-recursive approach. In this approach answers to the subproblem are recorded.

eg Fibonacci numbers

The recursive program to compute Fibonacci numbers is inefficient.

Compute Fibonacci number f_n there is a call to fibonacci of f_{n-1} , f_{n-2} , f_{n-3} computed two times, f_{n-4} or three times.

The same computations can be avoided by recording the value.

ie. if f_{n-3} is computed once it is recorded & subsequent recursive calls are not made.

$$F_2 = f_1 + f_0$$

F_2 is recorded can be used for computation of F_3

$$F_3 = F_2 + f_1$$

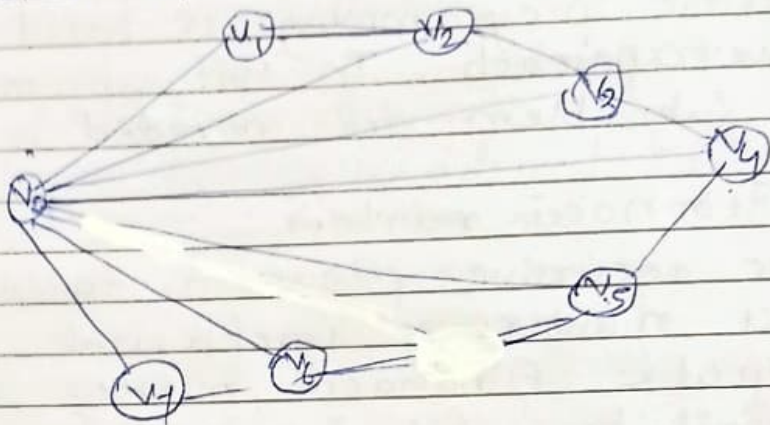
Triangulation Problem

In triangulation problem a polygon with 'n' vertices is given

$V_0, V_1, \dots, V_{n-1}$ Distance measure between each pair of vertices is provided. The problem is to select a set of chords (lines between non-adjacent vertices).

Such that

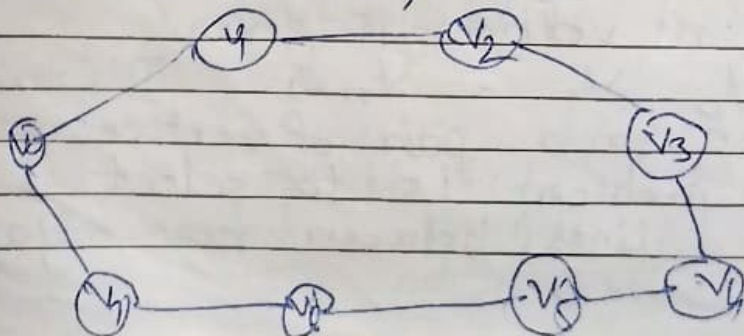
- 1) No two chords cross each other
- 2) The entire polygon is divided into triangles
- 3) The total length of the chords selected must be minimum



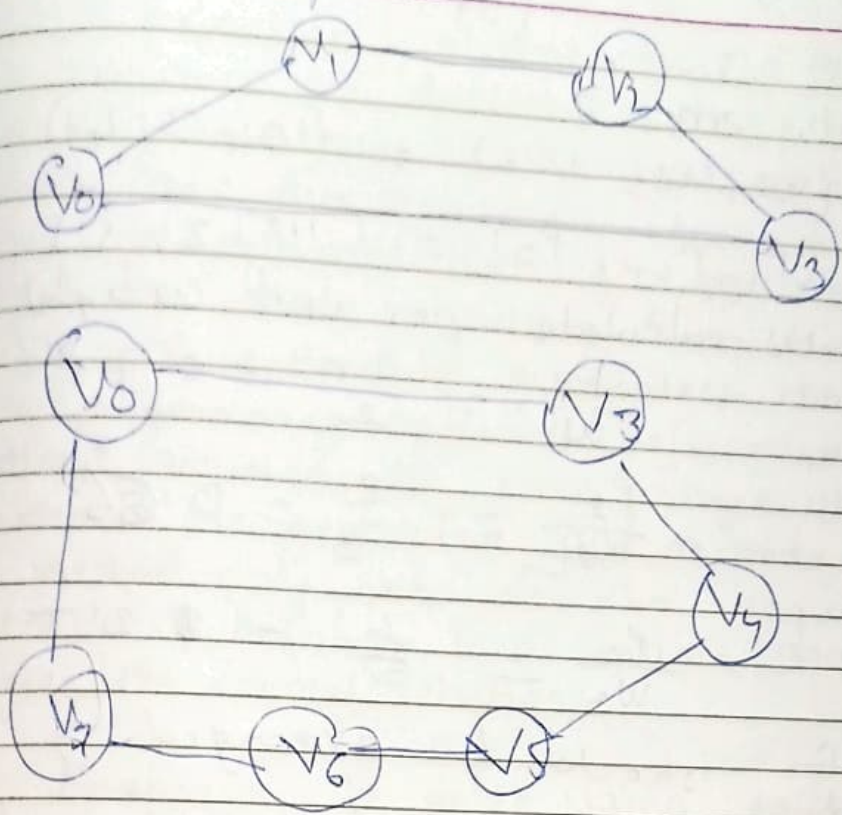
In order to find a minimum triangulation

- 1) Pick two adjacent vertices as V_i and V_{i+1}
- 2) Pick up a vertex V_k such that (V_i, V_k) and (V_k, V_{i+1}) are chords in triangulation
- 3) Find a good choice of 'k' by considering every value of k and analyzing triangulation
If the polygon has n vertices there are total $n-2$ choices to make

Each choice of 'k' leads to at most two subproblems



first sub problem



These are the two subproblems after selecting V_3

01. Knapsack problem

Given 'n' objects. Each object i has a weight w_i and if object i is placed then it gives profit p_i . Either an object is selected for filling or not selected. We cannot select fraction of object profit due to object is $i = p_i x_i$

$x_i = 0$ object is not selected
 $x_i = 1$ object is selected

The capacity of knapsack is m . The total weight of chosen objects should not be more than m .

$$\text{Maximize } = \sum_{i=1}^n p_i x_i$$

subject to $\sum_{i=1}^n w_i x_i \leq m$

$n = 3$

Capacity $m = 7$

$(w_1, w_2, w_3) = (2, 3, 4)$

$(p_1, p_2, p_3) = (1.5, 2, 5)$

Let us calculate per unit weight

$$\frac{p_1}{w_1} = \frac{1.5}{2} = 0.75$$

$$\frac{p_2}{w_2} = \frac{2}{3} = 0.66$$

$$\frac{p_3}{w_3} = \frac{5}{4} = 1.25$$

$\frac{p_i}{w_i}$ should be arranged in

~~increasing order~~ decreasing order of weight and objects should be in increasing order of weight

$$x_3 \left(\frac{p_3}{w_3} = 1.25 \right) \quad x_1 \left(\frac{p_1}{w_1} = 0.75 \right)$$

$$x_2 \left(\frac{p_2}{w_2} = 0.66 \right)$$

Step	object selected	weight	Profit
1	x_3	4	5
2	x_3, x_1	6	6.5

the objects selected are

$$(x_1, x_2, x_3) = (1, 0, 1)$$

with total profit 6.5

Principle of optimality

The principle states that the optimized sequence of decisions has the property that whatever the initial state & decisions are the remaining decisions must constitute an optimal decision sequence with respect to the state resulting from first decision.

Principle of optimality guarantees that the dynamic programming approach will always generate an optimal solution. At each stage all possible subproblems are generated and recorded. In greedy method only one decision sequence is generated it is based on local optima and may not give the optimal solution.

Q1/1 Knapsack to solve using dynamic programming

$$n = 3$$

$$(w_1, w_2, w_3) = (2, 3, 4)$$

$$(p_1, p_2, p_3) = (1, 2, 5)$$

Let us define a 3-tuple $t_i = (\text{set of objects}, \text{weight}, \text{profit})$

Object 1 is selected $t_1 = (1, 2, 1)$

If the object '1' is not selected $t_1 = (1, 0, 0)$

We will start with a set of above two tuples

$$\text{ie } S_1 = \{(1, 0, 0), (1, 2, 1)\}$$

Again in the next iteration we will create another set S_2 by adding (w_2, p_2) to every tuple of S_1 , S_2 will be obtained by taking union of S_1 and S_2

* Backtracking

8-queens problem

The problem is to place ~~8~~ 8-queens on 8×8 chessboard. No two should attack each other.

- a) No two should be on the same row.
- b) No two should be on the same column.
- c) No two should be on the same diagonal.

	0	1	2	3	4	5	6	7
0				Q				
1						Q		
2								Q
3		Q						
4							Q	
5	Q							
6			Q					
7					Q			

one solution to 8 queens problem

Let us assume that

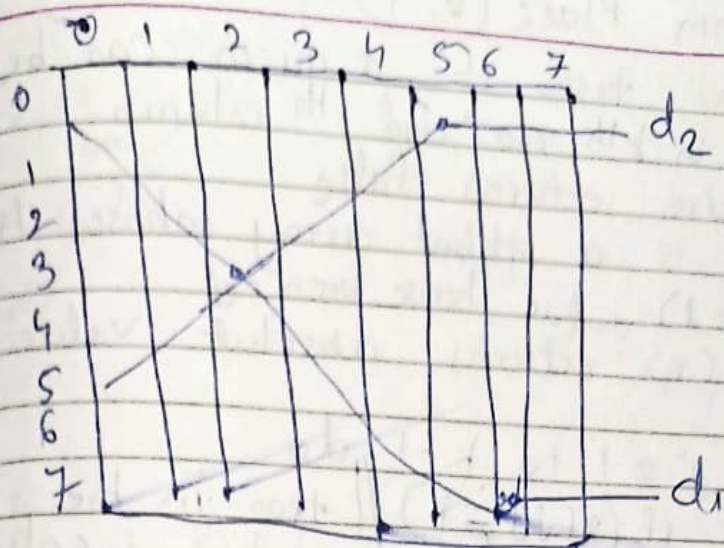
- a) Rows of the chessboard are numbered from 0 to 7.

Since each row can have exactly one queen we can assume that queen 'i' is placed in row 'i' of the chessboard.

A solution of the chessboard is given by 8 tuple $(x_0, x_1, \dots, x_7)$ where x_i is the column on which queen 'i' is placed.

i.e. $(3, 5, 7, 1, 6, 0, 2, 4)$

The relation between two columns on the same diagonal



Let us consider a cell with index $(3, 2)$

The cells on diagonal d_1 are given by $(1,0), (2,1), (4,3), (5,4), (5,5), (7,6)$

The cells on diagonal d_2 are given by $(5,0), (4,1), (2,3), (1,4), (0,5)$

Two elements (i, j) and (k, l) will have following relation

$$i - j = k - l \quad \text{--- (1)}$$

$$\text{or } j - l = i - k$$

Two elements on diagonal d_2 will have following relation or $j - k = l - i$

$$i + j = k + l \quad \text{--- (2)}$$

from eqⁿ (1) & (2) we can infer that queens lie on the same diagonal if and only if

$$|i - j| = |k - l|$$

The function `queens()` prints all possible placements of n queens on $n \times n$ chessboard so that they are non attacking

The function `queens` has two parameters

1) k - row in which current queen is to be placed

2) n - no. of queens
initial values of k & n are 0 & 8

Algorithm Place(k, i)

Returns true, if a queen can be placed in k th row and i th column otherwise returns false

$x[]$ is a global array whose first $(k-1)$ values have been set
 $Abs(x)$ returns absolute value of x .

```
for  $j = 1$  to  $k-1$  do
    if ( $x[j] == i$ ) // too in the same column
        or ( $Abs(x[j] - i) == Abs(j - k)$ ) // or in a diagonal
        then return false
return true
```

Algorithm NQueens(k, n)

// Using backtracking
// prints all possible placements of n queens on an $n \times n$ board such as non attacking

```
for  $i = 1$  to  $n$  do
```

```
    if Place( $k, i$ ) then
```

```
        {  $x[k] = i$ 
```

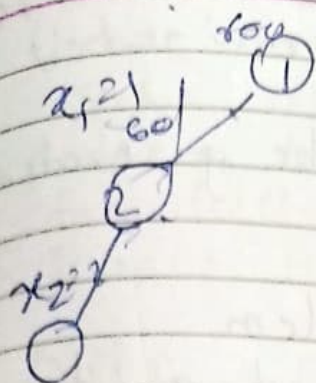
```
        if ( $k = n$ ) then write ( $x[1:n]$ )
```

```
        else
```

```
            NQueens( $k+1, n$ );
```

```
    }
```


state space tree



Branch & Bound

This strategy can be used to solve optimization problems without an exhaustive search in average case.

It is an algorithm design paradigm for discrete and combinatorial optimization problems as well as mathematical optimization.

The algorithm explores branches of this tree which represent subsets of a solution set.

This method searches a state space tree using any search mechanism in which all the children of E-node are generated before another node becomes E-node.

- Each answer node x has a cost $C(x)$. We have to find minimum cost answer node.

Use cost function $\bar{C}(x)$ such that $\bar{C} \leq C(x)$ provides the bounds on the solution obtainable from any node x .

Two Mechanisms

- A mechanism to generate branches of the solution space.
- A mechanism to generate a bound on the cost of any branch.

- Efficient in average case because many branches can be terminated very easily
- A very large tree may be generated in worst case

eg

Traveling salesman problem

Given a graph to find a tour starting from any vertex visiting every other vertex & return back with minimal cost.

- Initialization: Construct an initial feasible tour by some heuristic or cycle tour

$1 \rightarrow 2 \dots n$ and set

$$f_{min} = c_{12} + c_{23} + \dots + c_{n1}$$

- Solve the assignment relaxation if the solution is a Hamiltonian cycle stop it is optimum

- Choose a subtour with the smallest number of nodes in order $(1) \dots i(k) \dots k < n$
In this solⁿ

$$x_{(1)(2)} = x_{n(k)(3)} = \dots$$

$$x_{k-1,k} = \dots = x_{k,1} = 1$$

- Define k subproblems by deleting each edge of a subtour in turn by setting

$$x_{n(2)} = 0 \quad \text{--- 1st subproblem}$$

$$x_{(2)(3)} = 0 \quad \text{--- 2nd subproblem}$$

$$x_k(i) = 0$$

Choose a subproblem to be solved.
Solve associated assignment problem by setting $C_{ij} = \infty$ for deleted edge.
Denote the solution by x .

- i) if $f < f_{min}$ but subtours exists go to 2
- ii) bounding if $f < f_{min}$ and the solution has no subtours
 $x_{min} = x$ $f_{min} = f$ Go to 6
- iii) if $f \geq f_{min}$ go to 6

6. If unsolved subproblems exists go to step 4
 else stop, the optimum solⁿ is x_{min} , f_{min}

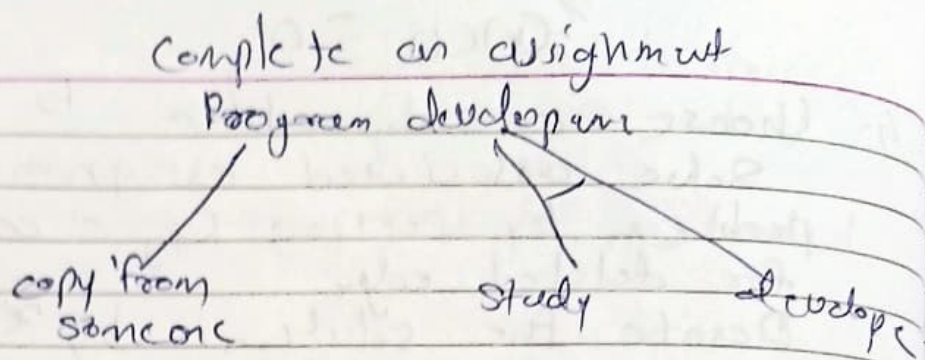
* Randomized algorithms

An algorithm that uses random numbers to decide what to do next anywhere in the logic is called randomized algorithm.
 eg. in randomized Quick Sort we use random number to pick the next pivot (or ~~sh~~ shuffle the array)

Some randomized algorithms have deterministic time complexity such algorithms are called Monte Carlo algorithm

* AND/OR Graphs

- Some problems are best represented as achieving subgoals. Some of which are achieved simultaneously and independently



Searching AND/OR tree

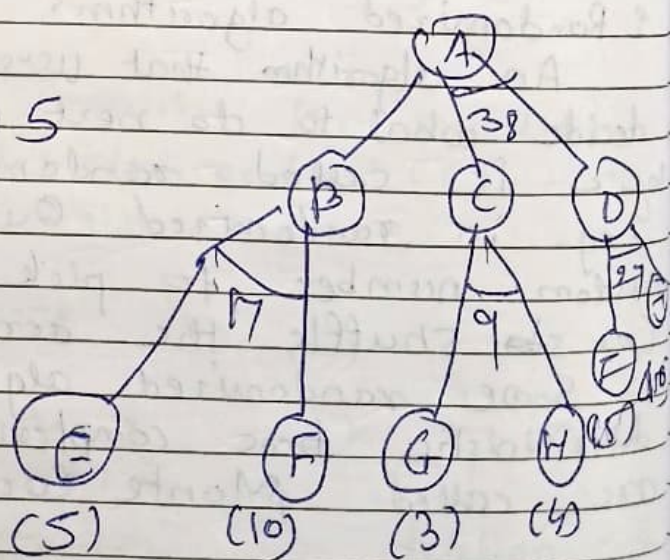
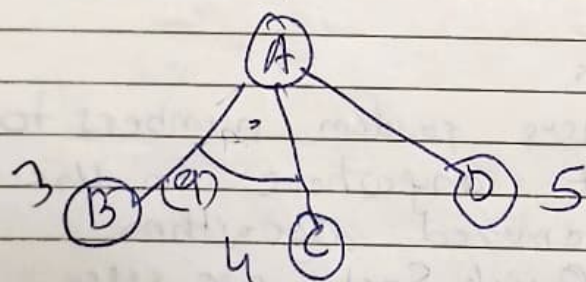
* A solution in AND/OR tree is a subtree whose leafs are included in a goal set

* cost function: sum of costs in AND node
 $f(n) = f(n_1) + f(n_2) + \dots + f(n_n)$

* How

AND/OR Search

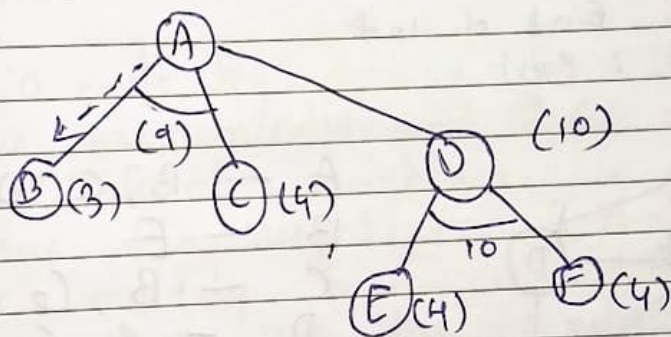
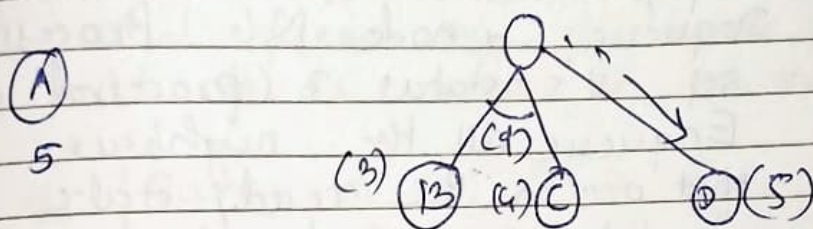
We must examine several nodes simultaneously when choosing the next move



* Traverse the graph (from the initial node) following the best current path

* Pick one one of the unexpanded nodes on that path and expand it: Add the

- Successors to the path and compute f for each of them
- Change the expanded node's f value to reflect its successors propagate the change up the graph.
- Reconsider the current best solution and repeat until the solution is found.

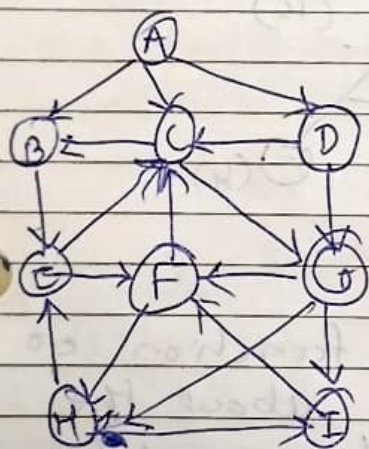


Heuristic function - Is a function or a way to inform a search about the direction to a goal. It provides an informed way to guess which neighbours of a node will lead to a ~~node~~ goal. It is just the information that can be readily obtained about a node.

DFS and BFS

BFS

- step 1: set status = 1 (ready state)
for each node in G
- step 2: Enqueue the first node 'A' and
set it's status = 2 (waiting state)
- step 3: Repeat step 4 and 5 until queue is
empty
- step 4: Dequeue a node 'N' Process it and
set it's status 3 (processed state)
- Step 5 Enqueue all the neighbours of 'N'
that are in the ready state
i.e status = 1 and set status = 2
- End of loop
- step 6 : exit



A → B, C, D

B → E

C → B, G

D → C, G

E → G, F

F → C, H

G → F, H, I,

H → E, I

I → F

Front = Rear = -1

Queue is used to hold the nodes that
we have processed. Origin is used
to keep track of the origin of
each edge.

- 1) Add A to queue & Null to Origin
- Front = 0 Queue = A
- Rear = 0 Origin = Null

- 1- Ready - Initial state of the node N
 2- Waiting - Node A is placed on queue/stack and waiting to be processed
 3- Processed - Completely processed
 Dequeue by setting $\text{Front} = \text{rear} + 1$
 Remove front element of a queue and enqueue neighbours of A

(a) D. $\text{Front} = 1$ | Queue = A B C D
 $\text{Rear} = 1$ | ORIG = \0 A A A

- c) Dequeue a node by setting $\text{FRONT} = \text{FRONT} + 1$ and enqueue the neighbours of B
 Also add B as origin of its neighbours

$\text{Front} = 2$ | Queue = A B C D E
 $\text{REAR} = 4$ | ORIG = \0 A A A B

- d) Dequeue a node by setting $\text{FRONT} = \text{FRONT} + 1$ and enqueue the neighbour of C. Also add C as a ORIG of neighbours
 C - has two neighbours B and G
 As B is already been added to queue and it is not in the ready state
 We do not add B
 add only G

$\text{Front} = 3$ | Queue = A B C D E G
 $\text{Rear} = 5$ | ORIG = \0 A A A B C

- e) Dequeue a node by setting $\text{Front} = \text{Front} + 1$ and enqueue neighbours of D. Also add D as origin of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to queues and they are not in ready state we will add them again

$\text{Front} = 4$ | Queue = A B C D E G
 $\text{Rear} = 5$ | ORIG = \0 A A A B C

f) Dequeue a node by setting
 $\text{Front} = \text{front} + 1$ and enqueue the
 neighbours of E
 Also add E as the origin of neighbours
 E has two neighbours C and F
 Since C has already been added to
 queue and is not in ready state
 So do not add C add only F

Front = 5	Queue = A	B	C	D	E	G	F
Rear = 6	ORIG = \0	A	A	A	B	C	E

g) Dequeue a node by setting $\text{Front} = \text{front} + 1$
 and enqueue neighbours of G Also
 add 'G' as the ORIG of it's neighbours
 Note that 'G' has three neighbours
 F, H and I

Front = 6	Queue = A	B	C	D	E	G	F	H	I
Rear = 9	ORIG = \0	A	A	A	B	C	E	G	G

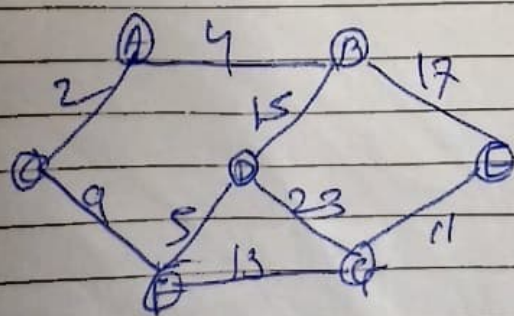
Since F has already added to the
 queue we will only add H and I
 I is final destination

The algo will terminate once
 I added to queue
 backtrack from I using ORIG
 $A \rightarrow C \rightarrow G \rightarrow I$

- 1) Set status = 1
 - 2) Push the starting node A on the stack and set status 2
 - 3) Repeat steps 4 and 5 until stack is empty
 - 4) Pop the top Node 'N' - process and set status = 3
 - 5) Push on the stack all the neighbours of 'N' that are in the ready state (whose status = 1) and set their status = 2
- end of loop
exit

Dijkstra's Algorithm

1. Select the source node also called as initial node
2. Define an empty set 'N' that will be used to hold nodes to which a shortest path has been found
3. Label the initial node with '0' and insert it into 'N'
4. Repeat the steps 5 to 7 until the destination node is in 'N' or there are no more labelled nodes in 'N'
5. Consider each node that is not in N and is connected by an edge from the newly inserted node
6. a) If the node that is not in 'N' has no label then set the label of the node = the label of the newly inserted node + length of the edge
b) Else if the node that is not in N was already labelled then SET its ~~new~~ new label = minimum (label of newly inserted vertex + length of edge, old label)
7. Pick a node not in 'N' that has the smallest label assigned to it and add it to 'N'



Bubble Sort

- simple method that sorts the array elements by repeated moving the largest element to the highest position
- consecutive adjacent elements are compared if the element is lower index is greater than the element in higher index then the positions of these are interchanged so that the element is placed before

This procedure continues till the last of unsorted elements exists.

In the first iteration the largest element will be kept in proper position

Bubble Sort

bubble sort (sort A.)

1 Repeat Step 2 for $I = 0$ to $N-1$

2 Repeat step 2 for $J = 0$ to $N-I$

IF $A[J] > A[J+1]$;

Swap $A[J]$ and $A[J+1]$

end of loop

end of loop

3 Exit

Complexity of sorting algorithm depends on comparisons

In bubble sort there are $N-1$ passes in total

In first pass $N-1$ comparisons are made to place highest element in correct position then in pass 2 $N-2$ comparisons to place 2nd highest

$$f(n) = (n-1) + (n-2) + (n-3) \dots 3+2+1$$

$$f(n) = n(n-1)/2 = n^2/2 + O(n)$$

$$= O(n^2)$$

* Insertion sort

- Simple sorting algorithm
- Sorted array is built one at a time
- Process is similar that we follow in ordering deck of cards.
- It inserts each item into it's proper place in the final list.

Technique

- The array of values to be sorted is divided into two sets one that stores sorted values and another that contains unsorted values
- The sorting algorithm will proceed until there are no elements in unsorted set.
- Suppose there 'n' elements in the array. Initially the element with index '0' (assuming $LB=0$) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 if ($LB=0$)
- During each iteration of algorithm the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

39 9 45 69 18 81 108 54 72 36

A[0] is only element in sorted list

9 39 45 69 18 81 108 54 72 36

step 1 Repeat Step 2 to 5 for $K=1$ to $N-1$
 2 Set $temp = ARR[K]$
 Set $J = K-1$
 Repeat while $temp < arr[J]$
 Set $ARR[J+1] = ARR[J]$
 Set $J = J-1$
 3 Set $ARR[J+1] = temp$
 end of loop
 4 exit

* Selection Sort

- It is simple
- It has a quadratic time complexity
- Inefficient than insertion sort
- Simple to implement

The technique

- First find the smallest value in the array and place it in the first position
- Then find second smallest value place it in the second position.

Repeat the procedure until the entire array is sorted.

39 9 81 45 90 27 72 18

9 39 81 45 90 27 72 18

9 18 81 45 90 27 72 39

9 18 27 45 90 81 72 39

9 18 27 39 90 81 72 45

9 18 27 39 45 81 72 90

9 18 27 39 45 72 81 90

Algorithm

Algorithm Selection Sort

~~Smallest (arr, k, N, pos)~~

- 1: Set $Small = Arr[k]$
- 2: Set $POS = k$
- * 3: Repeat for $J = k+1$ to $N-1$
if $small > Arr[J]$
set $small = Arr[J]$
set $POS = J$
end of if
end of loop
- step 4: Return POS

Selection Sort (Arr, N)

step 1: Repeat steps 2 and 3 For $k = 1$ to $N-1$

step 2: call $smallest(arr, k, N, POS)$

step 3 swap ($Arr[k]$, with $Arr[POS]$)
end of loop

step 4: exit

- The algorithm is independent of original order of elements
- ~~Best~~ suitable for small
- More efficient than bubble sort

MERGE SORT

The algorithm uses divide and conquer algorithmic strategy.

Divide - Means partitioning the n -element array to be sorted into two sub-arrays of $n/2$ elements. If A is an array containing 0 or 1 element then it is already sorted

However if there are more elements in array divide 'A' into two subarrays A_1 and A_2 each containing about

half of the elements of 'A'

Conquer - means sorting the two subarrays recursively using merge sort.

Combine - Merging the two sorted subarrays of size $n/2$ to produce the sorted array of n elements.

Merge Sort - focuses on two main concepts to improve the performance

- A smaller list makes the fewer steps and thus less time to sort than large list.
- As number of steps is relatively less thus less time is needed to create a sorted list from two unsorted lists

Basic steps

- If the array is of length 0 or 1 then the list is sorted
- otherwise divide the unsorted array into two subarrays of about half the size
- Use merge sort algorithm recursively to sort each subarray.
- Merge the two sub-arrays to form a single sorted list.

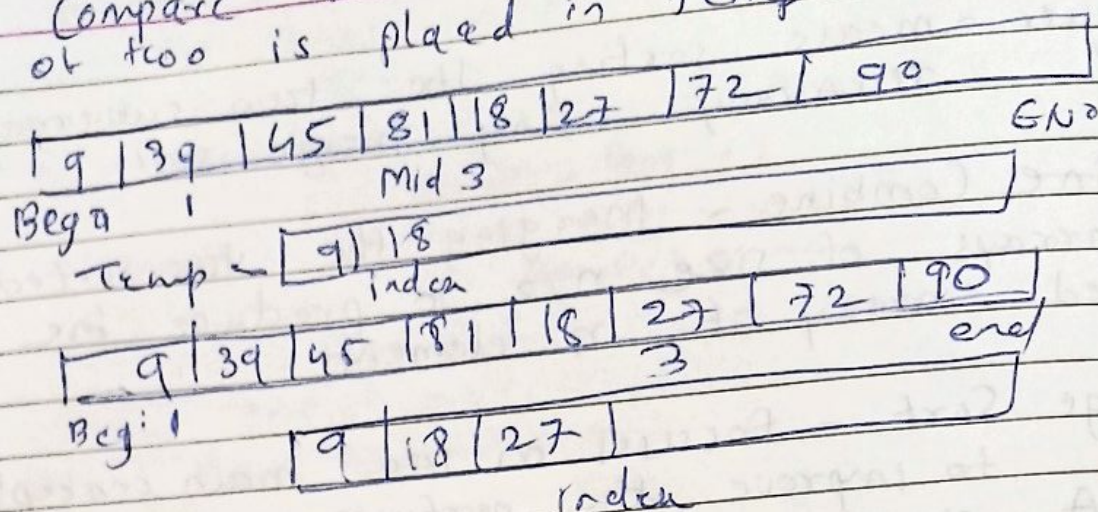
39 9 81 45 90 27 72 18

39	9	81	45	90	27	72	18
----	---	----	----	----	----	----	----

39	9	81	45	90	27	72	18
----	---	----	----	----	----	----	----

39	9	81	45	90	27	72	18
----	---	----	----	----	----	----	----

Compare $ARR[I]$ and $ARR[J]$ the smaller of two is placed in temp



when I is greater than Mid then copy the remaining elements of the right sub-array in temp.

MERGE (arr, beg, mid, end)

1. Set $I = BEG$ $J = Mid + 1$ Index = 0
2. Repeat while ($I \leq Mid$) and ($J \leq END$)

if $arr[I] < arr[J]$

set $temp[index] = arr[I]$

Set $I = I + 1$

else

set $temp[index] = arr[J]$

set $J = J + 1$

end of if

set $index = index + 1$

end of loop

Step 3 : (Copy the remaining elements right sub-array if any)

If $I > Mid$

Repeat while $J \leq END$

set $Temp(index) = arr[J]$

set $Index = index + 1$

Set $J = J + 1$

end of loop

Copy the remaining elements of left subarray if any
else

repeat while $I \leq \text{Mid}$

set $\text{temp}[\text{index}] = \text{arr}[I]$

set $\text{index} = \text{index} + 1$

set $I = I + 1$

end of loop

end of if

step 4: Set $K = 0$ (Copy temp array to arr)

step 5: Repeat-while $K \leq \text{index}$

set $\text{arr}[K] = \text{temp}[K]$

Set $K = K + 1$

end of loop

step 6: end.

Algorithm for Merge Sort

Merge Sort (arr, beg, end)

Step 1 if $\text{Beg} \leq \text{end}$

set $\text{mid} = (\text{beg} + \text{end}) / 2$

call merge sort (arr, beg, mid)

call merge sort (arr, mid+1, end)

merge (arr, beg, mid, end)

end of if

step 2: end

- The running time complexity in average and worst case is $O(n \log n)$
- optimal time
- Need additional space $O(n)$ to store temporary array

Quick Sort

- Divide and conquer strategy
- Divide single unsorted array into two smaller subarrays.
- Select an element pivot from array elements
- Re-arrange the elements in the array in such a way that all elements in the array that are less than pivot appear before the pivot and all elements greater than pivot come after it. After such a partitioning the pivot is placed in its final position. This is called partition operation.
- Recursively sort the two arrays thus obtained.

* Radix Sort

- Is a linear sorting algorithm.
- Can also be called as bucket sort.
- For letters 26 alphabets 26 buckets.
- For integer 10 buckets 0-9.
- By sorting least significant ~~bit~~ digit to most significant ~~bit~~ digit.
- No. of passes depends on length of the no. having max. no. of digits.

Algorithm

1. Find the largest no. in the array as large.
2. Set $NOP = \text{Number of digits in large}$.
3. Set Pass = 0.
4. Repeat step 5 while Pass $\leq$ NOP.
5. Set $I = 0$ and initialize buckets.
6. Repeat step 7 to 9 while $I < N$.

- 7: Set digit = digit at posth place in ACIJ
- 8: Add ~~ACIJ~~ to the bucket numbered digit
- 9: Increment bucket count for bucket numbered Digit
- 10: Collect the numbers in the bucket - end of loop
- 11: End

Quick Sort

- 1) Set index of the first element in the array to Loc & left variable
Set index of last as right
Loc = 0 left = 0 and right = n - 1
- 2) Scan from right to left compare each element on the way with Loc element
i.e. $a[\text{Loc}]$ must be less than $a[\text{right}]$
 - a) if so continue until right becomes equal to Loc
once $\text{right} = \text{Loc}$
that means pivot placed at correct place
 - b) However if any point we have $a[\text{Loc}] \geq a[\text{right}]$ then interchange the two values and go to step 3
 - c) set $\text{Loc} = \text{right}$
- 3) Start from element pointed by left and scan the array from left to right

compare each element with Loc element

$a[Loc]$ must be $>$ than $a[left]$

a) if so then simply compare
until left becomes equal to Loc
once $left = Loc$

means the pivot has been placed
at right place

b) if any point we have
 $a[Loc] < a[left]$ then
interchange two values
jump to step

c) Set $Loc = left -$

* Topological Sort

Topological sort of a directed acyclic graph (DAG) is defined as linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges.

A topological sort of a DAG is an ordering of the vertices of G such that if G contains an edge (u, v) then u appears before v in the ordering.

Topological sort is possible only on directed acyclic graph that

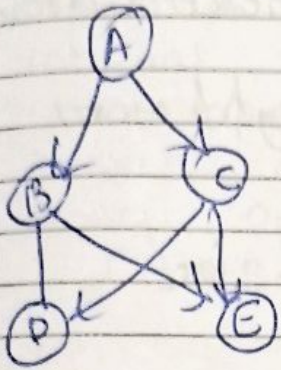
In a directed cyclic graph no linear ordering is possible.

In simple words a topological ordering of DAGs is an ordering of its vertices such that any directed path in G traverses the vertices in increasing order.

These types of sorting are used in job scheduling to indicate

dependency of execution

It gives order in which the jobs need to be performed



It focuses on selecting a node 'N' with zero in-degree that is a node that has no predecessors

Two main steps include

- selecting a node with zero in-degree
- Deleting 'N' from the graph along with its edges

1. Find the in-degree $\text{indeg}(N)$ of every node in the graph
 2. Enqueue all the nodes with a zero in-degree
 3. Repeat steps 4 and 5 until the queue is empty
 4. Remove the front node 'N' of the queue by setting
 $\text{Front} = \text{Front} + 1$
 5. Repeat for each neighbour M of Node 'N'
 - a) Delete the edge from N to M by setting $\text{indeg}(M) = \text{indeg}(M) - 1$
 - b) If $\text{indeg}(M) = 0$, then enqueue M that is add M to the rear of the queue.
 6. end of loop
 7. end of loop
 8. exit
- Q is used to hold the nodes with zero in-degree

- complexity is $O(n^2)$

- Initializing indegree takes time $O(n)$
- Loop n times to enumerate vertices inside loop, identifying next vertex is $O(n)$
- update indegrees of neighbours is $O(n)$

- Every directed acyclic graph can be topologically ordered

* Strategy

- First list the vertices with no incoming edges
- List the vertices whose incoming neighbours are already listed

Algo

- Start with any vertex 'v' such that $\text{indegree}(v) = 0$

- walk backwards to the predecessor so long as $\text{indegree}(u) > 0$

- If no vertex has indegree 0 within n steps we will complete a cycle

- Pick a vertex with indegree 0

- no dependencies

- enumerate it & delete from the graph

- what remains again in graph

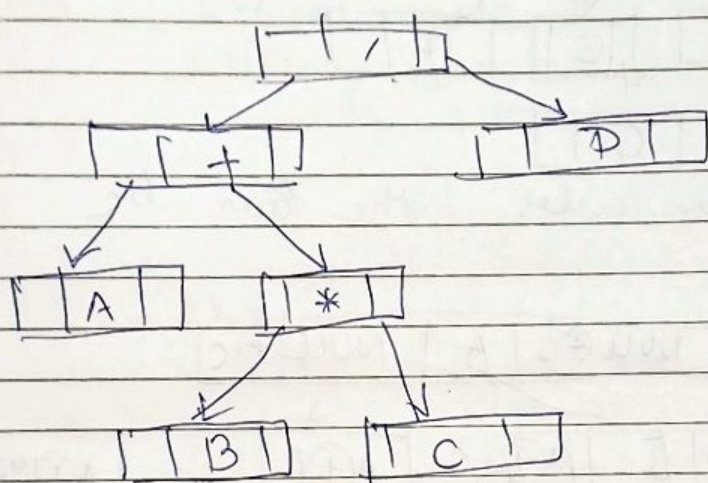
Repeat the step above
Stop when resulting DAG is empty

DAG is empty

* Expression Trees

A tree representing an expression is called an expression tree. An expression tree is a binary tree where internal nodes are operators and leaves are operands. An expression tree consists of a binary expression but for n -ary operator one subtree will be empty.

$(A + B * C) / D$



XOR Tree

This concept is similar to memory efficient doubly linked list. Similar to threaded binary trees, this representation does not need stacks or queues for traversal.

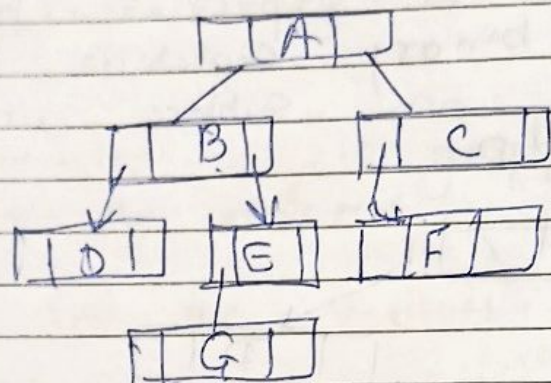
This representation is used for traversing back (parent) and forth (children) using $\oplus$ operators. To represent the same in XOR tree, for each node, below are the rules used for representation.

- Each node left will have the $\oplus$ of its parent and its left children.

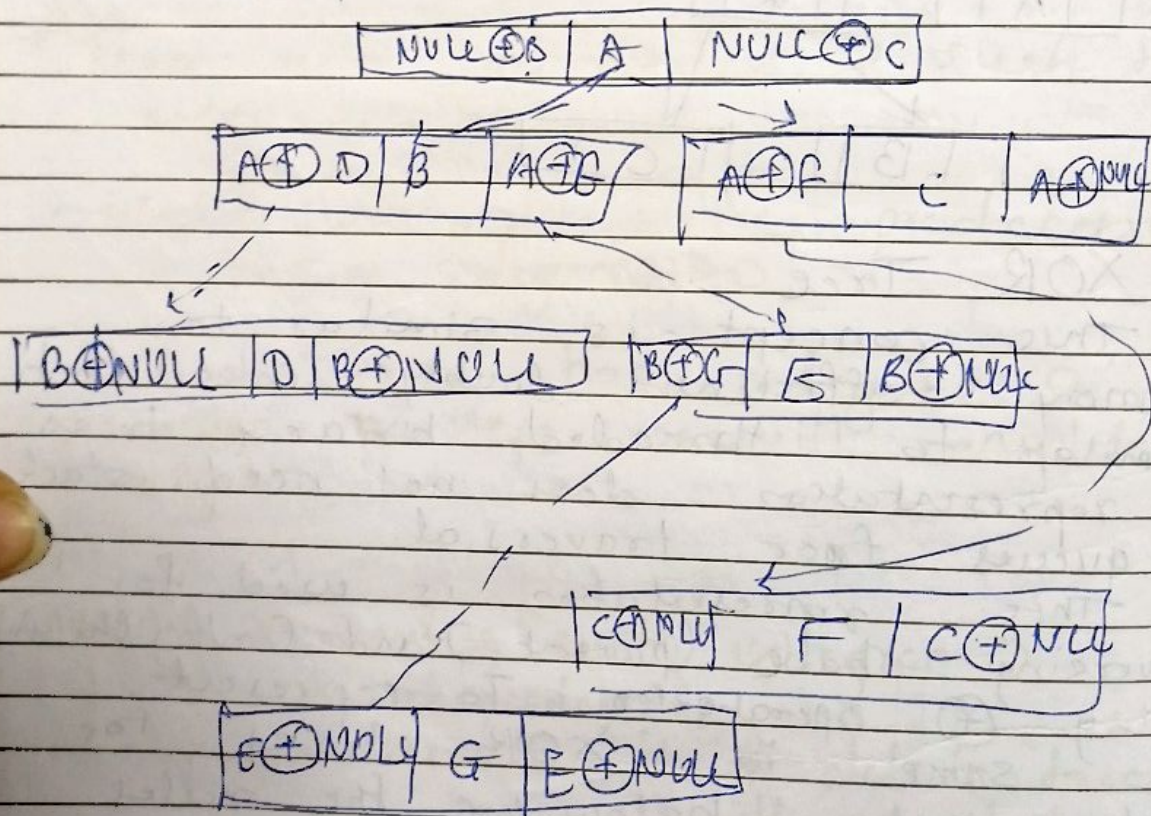
- Each node right will have the $\oplus$ of its parent and its right children.

children

- The root node's parent is NULL and also leaf nodes children are NULL nodes



Based on the above rules the tree can be represented as



The major objective is to traverse a tree

Recursion & Backtracking

Any function which calls itself is called recursive. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called a recursion step. It can result in many more recursive calls.

```
if (test for the base case)
    return some base case value
else if
    (test for another base case)
    return
    (some other base case value)
// the recursive case
else
    order (some work and
    then recursive call)
```

$$n! =$$
$$n = 0 \quad n! = 1$$
$$n = 1 \quad n! = 1$$
$$n! = n \times (n-1)! \quad \text{if } n > 1$$

problem determines factorial of n
subproblem " " of $n-1$

- Recursion terminates when base case is reached
- Each recursive call requires extra space on the stack frame
- If we get infinite recursion the program may run out of memory and result in stack overflow
- Solutions to some problem are easier to formulate recursively

R

Iteration

- Terminates when a condition is proved to be false
- Each iteration does not require extra space
- An infinite loop could loop forever since there is no extra memory being created
- Iterative solutions to a problem may not always be as obvious as a recursive solution

Backtracking

Backtracking is an improvement of the brute force approach. It systematically searches for a solution to a problem among all available options -

In backtracking we start with one possible option out of many available options and try to solve the problem. If we are able to solve the problem with available options, the selected move then we will print the solution, else we will backtrack and select some other option and try to solve it.

If none of the options can give a solution, then we will say no solution possible.

Backtracking is a form of recursion. The usual scenario is not you are faced with no. of options and you must choose one of these. The procedure is repeated till you reach final state.

Shell Sort

The algorithm is generalization of insertion sort.

main observations

- Insertion sort works well when the input data is almost sorted
- Insertion sort is quite inefficient to use as it moves the values just one position at a time

Technique

To visualize the way in which shell sort works perform the following steps

step 1: Arrange the elements of an array in the form of a table (sort solⁿs)

step 2: Repeat step 1 each time with smaller number of longer colⁿs. in such a way that at the end there is only one column of data to be sorted

Shell Sort (Arr, n)

step 1: Set Flag = 1 Gap size = N

step 2: Repeat steps 3 to 6 while Flag = 1 or Gap size ≥ 1

step 3: Set Flag = 1

step 4: Set Gap size = $(\text{Gap size}) / 2$

step 5 Repeat step 6 for $I = 0$ to $J < (N - \text{Gap size})$

step 6: IF $\text{Arr}[I + \text{Gap size}] > \text{Arr}[I]$

Swap $\text{Arr}[I + \text{Gap size}]$, $\text{Arr}[I]$

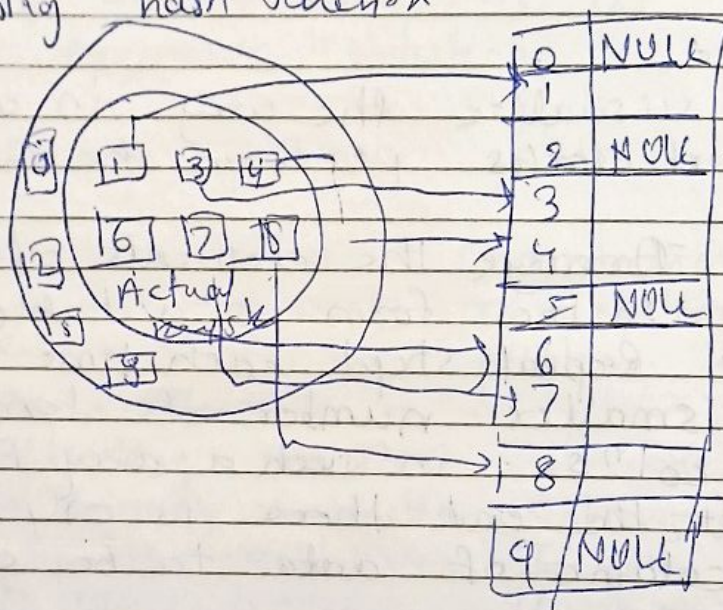
Set Flag = 0

step 7: END

Hashing

Hash table is a data structure in which keys are mapped to array positions by a hash function. In the example discussed here we will use a hash function that extracts the last two digits of the key.

Therefore we map the keys to array location or indices. A value stored in hash table can be searched in $O(1)$ time by using hash function.



Direct Relationship between key & index in the array

Hash Functions

A hash function is a mathematical formula which when applied to a key produces an integer which can be used as an index for the key in the hash table.

The main aim of a hash function is that elements should be relatively randomly and uniformly distributed. It produces a unique set of integers within some suitable range in order

to reduce collision If k_1 & k_2 keys point to same location that is called collision.

A good hash function can only minimise the no of collisions by spreading the elements uniformly throughout the array.

Properties of good hash function

Low cost - The cost of executing hash function must be low so that using the hashing technique is preferable

Deterministic - Same hash value must be generated for given value

Uniformity - a good hash function must map the values uniformly

Different hash functions

We will see hash functions which use numeric key. However there can be cases in real world applications where we can have alphanumeric keys rather than simple numeric keys. In such cases the ASCII value of character can be used to transform it into its equivalent numeric keys.

Once this transformation is done any of hash function given below can be applied to generate the hash value.

Division method

It is the most simple method of hashing as an integer x .

This method divide x by M and then uses the remainder only.

In this case the hash function can be

$$h(x) = x \bmod M$$

Division method is quite good for just about any value of M and since it requires only a single division operation. The method works very fast.

But value of ' M ' has to be chosen carefully.

* Multiplication method

* Mid-square method

* Folding method

Collision

Collision occurs when hash function maps two different keys to the same location.

Obviously two records cannot be stored in the same location.

1. method applied to resolve collisions are

1. Open addressing
2. Chaining

Collision

Resolution by Open Addressing

Collision resolution by chaining
Combines linked representation with hash table

When two or more records

hash to the same location these records are constituted into a singly linked list called a chain.

Open Addressing

In open addressing all keys are stored in the hash table itself. This approach is also known as closed hashing.

Linear probing

The interval between probes is fixed at 1.

*Collision Resolution by open

Addressing. Once a collision takes place open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position.

In this technique all the values are stored in the hash table.

The hash table contains two types of values

Sentinel values - eg -1
and data values.

The presence of a sentinel value indicates that the location contains no data at present but can be used to hold the data in future.

When the key is mapped to a particular location then value it holds is checked. If it contains

Prim's algorithm

- Greedy algorithm to obtain MST
 - connected weighted undirected graph
- Sets used

Tree vertices - vertices that are part of the minimum spanning tree 'T'.

Fringe vertices - vertices that are currently not part of a 'T' but are adjacent to some vertex in 'T'.

Unseen vertices - vertices that are neither tree vertices nor fringe vertices fall under this category.

step 1: select a starting vertex.

Step 2: Repeat steps 3 and 4 until there are fringe vertices.

Step 3: select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight.

Step 4: Add the selected edge and the vertex to the minimum spanning tree.

end of loop

Step 5: exit

Running time is $O(E \log V)$

where E: no. of edges

V: no. of vertices of graph.

* Edges in a set 'A' always maintain a tree

- * Start from an arbitrary root vertex 'r' and grow until tree spans all the vertices in 'V'
- * During algorithm vertices not in 'A' are in min. priority queue ordered by edge weight
 - after algorithm finishes queue is empty
- * maintains 3 loop invariants

a) $A = \{(u, v, w) : u, v \in V - Q, w \text{ is root}\}$

b) vertices in MST are those in $V - Q$

c) For all vertices in 'Q' if $v, w \neq \text{NIL}$ then v Key $< w$ and v is weight of a light edge connecting 'v' to some vertex already in the

Kruskal's algorithm

The algorithm is used to find the minimum spanning tree for a weighted connected graph.

The algorithm aims to find the subset of edges that forms a tree that includes every vertex.

The total weight of all the edges in the tree is minimized.

If the graph is not connected it finds minimum forest.

Step 1: Create a

Sort all the edges of the given graph in the increasing order and then keep on

adding new edges and nodes in the MST if the newly added edge does not form a cycle
- Repeat the steps till the no. of edges of the spanning is $n-1$ where ' n ' is vertices of the graph

Kruskal MST (G, n, f)

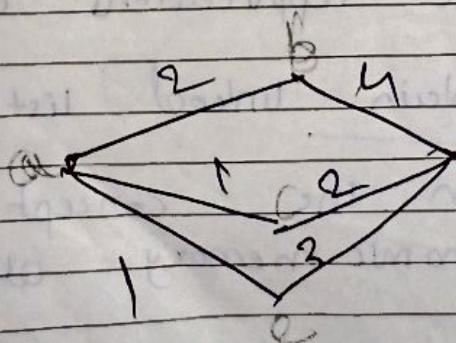
int count;
build a minimizing priority queue pq of edges of ' G ' prioritized by weight

Initialize a Union find structure sets in which each vertex of ' G ' is in it's own set

$F = \emptyset$;
while in Empty pq $\&$

Assignment - 1

- Q-1 Using 'Big Oh' notation specify the running time of the following code
- ```
void fun (int n)
{
 int a, b, c;
 for (a=1; a ≤ 2 * n; a++)
 for (b=1; b ≤ 2 * n; b++)
 printf ("Hello world \n");
 for (c=1; c ≤ 2 * n; c++)
 printf ("Hello world");
}
```
- Q-2 Define the three notations  $O$ ,  $\Omega$ ,  $\Theta$  with suitable examples.
- Q-3 Compare the algorithms: linear search and binary search to search for any element from a given set of 'n' elements.
- Q-4 Explain how stack is used in a non-recursive program.
- Q-5 Define ADT. Describe stack and queue as ADT.
- Q-6 Write an algorithm to perform operation of addition on two polynomials.
- Q-6 Explain the term frequency count. Explain why frequency count is only important in deciding the time complexity of algorithm.
- Q-7 Write an algorithm for Fibonacci search and calculate its time complexity.





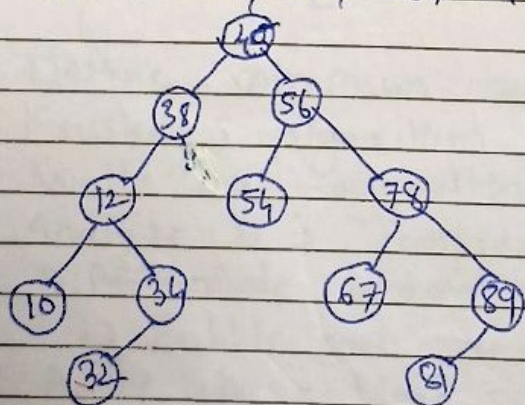
## Unit - II

- Q.1 Write an algorithm to perform operation of addition on polynomials using linked list.
- Q.2 Write an algorithm to reverse the elements of linked list using only pointer manipulation (do not use additional data structure).
- Q.3 Write a function to merge two sorted lists, destroying the original list.
- Q.4 Compare linear data structures using linked organization and sequential organization.
- Q.5 Define linked list. Describe different types of linked lists. Give advantages and disadvantages.
- Q.6 Describe the structure of circular doubly linked list. Write an algorithm to insert a node at the end of this list.
- Q.7 Write an algorithm to insert an element 'P' after a key 'K' in the list 'L'.
- Q.8 <sup>Explain</sup> List the applications of <sup>linked</sup> linked list.
- Q.9 Explain linked list as an ADT.
- Q.10 Explain the concept of static and dynamic memory allocation.



## Unit - III

- Q1) Show that the minimum height of a binary tree with 'n' nodes is  $\log_2(n+1) - 1$
- Q2) Write recursive 'C' function to find
  - i) Height of a given binary tree
- Q3) Define binary search tree. Write an algorithm to create binary search tree
- Q4) Explain the concept of threaded binary tree. State its advantages
- Q5) Explain how binary tree can be represented using array.
- Q6) Discuss in detail AVL trees.
- Q7) Consider the binary search tree given below. Do the following operations
  - Find the result of inorder, preorder and postorder traversal.
  - Show the deletion of the root node
  - Insert 11, 22, 33, 44, 55, 66, 77 in the tree



- Q8) Write an algorithm to count the no. of terminal nodes in a binary tree 'T'. Also compute the computing time
- Q9) Explain binary search tree. Illustrate the application of the same set representation and operations on set. such checking the membership, insert element, delete element



Q.12 Sort the following list of elements using Radix sort, selection sort and Shell sort

i) 69, 21, 33, 15, 18, 11, 9

ii) 16, 19, 29, 67, 57, 91

iii) 6, 12, 34, 69, 72, 99

Q.13 Write an algorithm to sort an array of names using bucket sort.

Q.14 Write a recursive function to perform Selection sort.

Q.15 Compare time complexities of different sorting algorithms

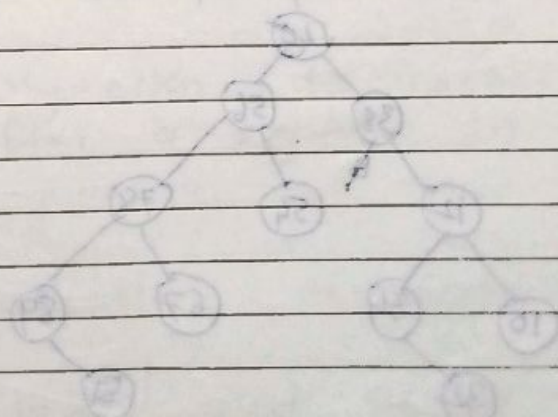
Q.16 Write a note on

i) Hashing

ii) Heap Sort.

iii) Selection of an algorithm considering data set.

iv) Limitations of Dijkstra's Algorithm





## Unit - IV

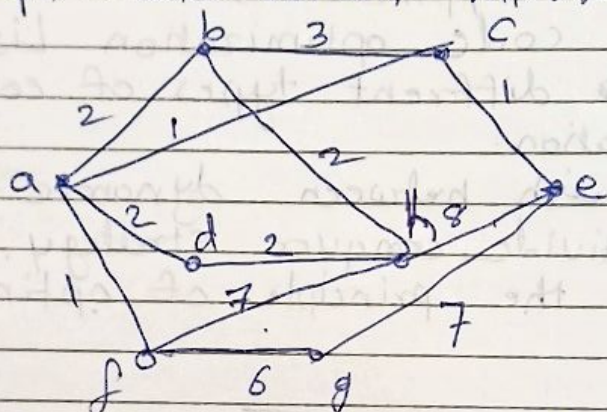
Q-1 Define graph list different types of graph. Draw a graph of each type

Q-2 Distinguish between BFS and DFS

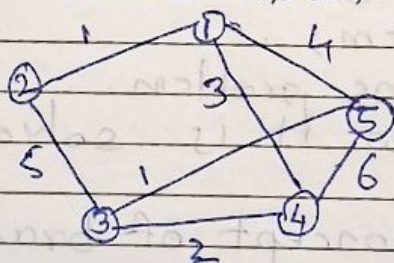
Q-3 Illustrate the working of BFS with suitable example

Q-4 Write algorithm for DFS and illustrate its working

Q-5 Apply Prim's algorithm on following graph and obtain minimal weight



Q-6 Find the shortest path from node 'a' to each vertex shown



Q-7 Define minimum spanning tree. Write Kruskal's algorithm

Q-8 Write an algorithm for 'Quick Sort'. Analyze its complexity

Q-9 Differentiate between

i) Bubble sort and Quick Sort

Q-10 Apply insertion sort on the following list:  
59, 61, 12, 17, 32, 14, 68, 7

Q-11 Apply Merge sort on the following list:

15, 7, 12, 19, 23, 64, 99, 8



## Unit - V

- Q-1 Illustrate working of dynamic programming strategy with suitable example.
- Q-2 Write the general strategy for Greedy method. Solve the knapsack problem using Greedy method (Assume suitable data).
- Q-3 Illustrate how divide-conquer strategy is applied on list of 'n' elements to sort using 'Merge Sort'.
- Q-4 Describe in detail AND/OR graphs. List its applications.
- Q-5 Define code optimization. List and explain different types of code optimization.
- Q-6 Distinguish between dynamic programming and divide conquer strategy.
- Q-7 Explain the principle of optimality.

## Unit - VI

- Q-1 Describe in detail NP-hard NP-complete problems.
- Q-2 Illustrate how it is solved using backtracking.
- Q-3 Explain the concept of branch-and-bound with suitable example.
- Q-4 Apply branch and bound strategy to apply 0/1 knapsack problem.
- Q-5 Apply graph coloring problem suitable method. Justify your answer.

$$\begin{aligned} n &= 3 & m &= 7 \\ (w_1, w_2, w_3) &= (2, 3, 4) \\ (p_1, p_2, p_3) &= (1.5, 2, 5) \end{aligned}$$