DESIGN & ANALYSIS OF ALGORITHMS

UNIT-1

→ Introduction
Algorithm definition and Specification
Asymptotic analysis - best, average and worst case
Performance measurements of Algorithms
Time and Space Complexities
Analysis of recursive algorithms

→ Basic Data Structures
Disjoint set operations
Union and find algorithms
Dictionaries
Graphs
Trees

Introduction :

→ What is An Algorithm ?

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. All algorithms must satisfy the following criteria :

① Input - Zero or more quantities are externally supplied.

② Output - At least one quantity is produced.

③ Definiteness - Each instruction is clear and unambiguous.

④ Finiteness - The algorithm terminates after a finite number of steps while tracing.

⑤ Effectiveness - Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper.

+ Algorithms that are definite and effective are also called Computational procedures.

eg : operating system of a digital computer. It is designed to control the execution

of jobs, in such a way that when no jobs are available, it does not terminate but continues in a waiting state until a new job is entered.

+ A program is the expression of an algorithm in a programming language.

The study of algorithms includes many important and active areas of research. These are:

(1) How to devise algorithms - Creating an algorithm is an art. There are various design strategies to devise new and useful algorithms.

(2) How to validate algorithms - Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. This process is 'algorithm validation'. The purpose of the validation is to assure that the algorithm works correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as 'program proving' or 'program verification'. A proof of correctness requires that the solution be stated in two forms. One form is a program

which is explained by a set of assertions about the input and output variables of a program. The second form is called a specification. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct. All these details may cause a proof to be very much longer than the program.

(3) How to analyze algorithms – This field of study is called analysis of algorithms. 'Analysis of algorithms' or 'performance analysis' refers to the task of determining how much computing time and storage an algorithm requires. It requires mathematical skills. This study allows to make quantitative judgements about the value of one algorithm over another. It also allows to predict whether the software will meet any efficiency constraints that exist. Questions such as how well does an algorithm perform in the best case, in the worst case, or on the average are typical.

(4) How to test a program – Testing a program consists of two phases :

- debugging
- profiling

'Debugging' is the process of executing programs

on sample data sets to determine whether faulty results occur and, if so, to correct them.

'Profiling' or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

→ Algorithm Specification : Algorithm is described in many ways –

- using a natural language like English
- graphic representations called flowcharts

Algorithms use a pseudocode resembling C.

(1) Comments begin with // and continue until the end of line.

(2) Blocks are indicated with matching braces: { and }. A compound statement can be represented as a block.

(3) An identifier begins with a letter. The data types of variables are not explicitly declared. Compound data types can be formed with records.

(4) Assignment of values to variables is done using the assignment statement

<variable> := <expression> ;

(5) There are two boolean values 'true' and 'false'.

In order to produce these values, the logical operators and, or, and not and the relational operators $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$ are provided.

⑥ Elements of multidimensional arrays are accessed using [and]. Array indices start at zero.

⑦ The following looping statements are employed:
for, while, and repeat-until

The <u>while</u> loop takes the following form:

```
while <condition> do
{
    <statement 1>
        :
    <statement n>
}
```

The general form of a <u>for</u> loop is:

```
for variable := value 1 to value 2 step step do
{
    <statement 1>
        :
    <statement n>
}
```

A <u>repeat-until</u> statement is constructed as follows:

```
repeat
    <statement 1>
        :
    <statement n>
until <condition>
```

(8) A conditional statement has the following forms:

```
if <condition> then <statement>
if <condition> then <statement 1> else
                    <statement 2>
```

(9) Input and output are done using the instructions 'read' and 'write'. No format is used to specify the size of input or output quantities.

(10) There is only one type of procedure: Algorithm. An algorithm consists of a heading and a body. The heading takes the form

Algorithm Name (<parameter list>)

where Name is the name of the procedure (<parameter list>) is a listing of the procedure parameters. The body has one or more statements enclosed within braces { and }. An algorithm may or may not return any values. Simple variables

to procedures are passed by value. Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type.

Examples:

Algorithm finds and returns the maximum of n given numbers

```
1  Algorithm Max (A, n)
2   // A is an array of size n.
3  {
4     Result := A[1];
5     for i := 2 to n do
6        if A[i] > Result then Result := A[i];
7     return Result;
8  }
```

Algorithm to perform Selection Sort

```
1  Algorithm SelectionSort (a, n)
2  // Sort the array [1:n] into non decreasing order.
3  {
4     for i := 1 to n do
5     {
6        j := i;
7        for k := i+1 to n do
```

```
8    if (a[k] < a[j]) then j := k;
9    t := a[i]; a[i] := a[j]; a[j] := t;
10   }
11 }
```

→ Recursive Algorithms - An algorithm is said to be recursive if the same algorithm is invoked in the body.

- An algorithm that calls itself is 'direct recursive'.
- Algorithm A is said to be 'indirect recursive' if it calls another algorithm which in turn calls A.

Examples :

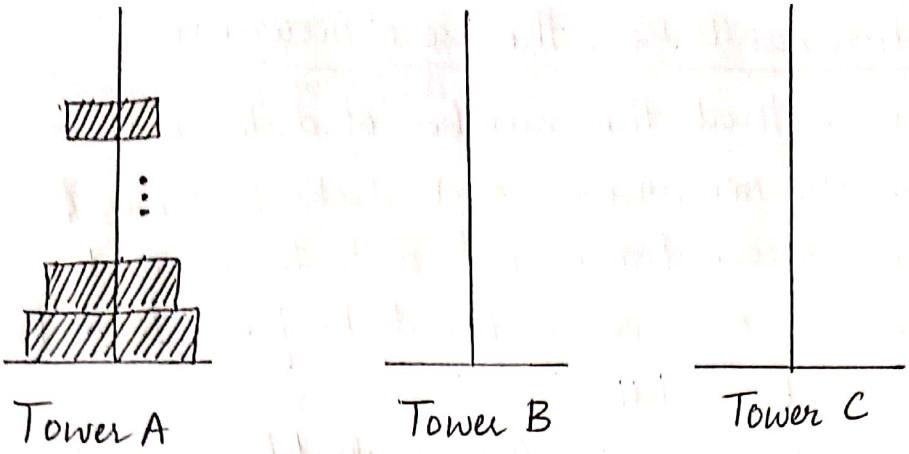
Towers of Hanoi —

There is a tower A with 64 golden disks. The disks are of decreasing size and stacked on the tower in decreasing order of size bottom to top. There are two other towers labeled B and C. The disks need to be moved from tower A to tower B using tower C for intermediate storage. As the disks are heavy they can be moved only one at a time. At no time can a disk be on top of a smaller disk.

A solution results from the use of recursion.

- Assume that the number of disks is n.
- Move the remaining n-1 disks to tower B.
- Then move the largest disk to tower C.
- The task of moving the disks from tower B to tower C is left.
- To do this, tower A is available.
- Tower C has the disk larger than the disks being moved from Tower B, and so any disk can be placed on top of it..
- This solution is recursive in nature.

```
1  Algorithm TowersOfHanoi (n, A, B, C)
2  // Move the top n disks from tower A to tower C.
3  {
4      if (n ≥ 1) then
5      {
6          TowersOfHanoi (n-1, A, C, B);
7          write ("move top disk from tower", A,
8                  "to top of tower", C);
9          TowersOfHanoi (n-1, B, A, C);
10     }
11 }
```



Permutation Generator —

Given a set of $n \geq 1$ elements, the problem is to print all possible permutations of this set.

- If the set is $\{a, b, c\}$, then the set of permutations is $\{(a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)\}$.

  For n elements, there are $n!$ different permutations.

- If the set is (a, b, c, d), then answer is

  a followed by all the permutations of (b, c, d)
  b followed by all the permutations of (a, c, d)
  c followed by all the permutations of (a, b, d)
  d followed by all the permutations of (a, b, c)

The expression "followed by all the permutations" indicates recursion which implies that the problem for a set with n elements can be solved with algorithm that works on $n-1$ elements.

```
1   Algorithm Perm(a, k, n)
2   {
3     if (k=n) then write (a[1:n]); // Output permutation
4     else // a[k:n] has more than one permutation.
5          // Generate these recursively.
6          for i:=k to n do
7          {
8            t:= a[k]; a[k]:= a[i]; a[i]:=t;
9            Perm (a, k+1, n);
10           // All permutations of a[k+1 :n]
11         t:= a[k]; a[k]:= a[i] ; a[i]:=t;
12         }
13  }
```

Performance Analysis of Algorithm

The criterias for judging algorithms have to do with their computing time and storage requirements.

The "space complexity" of an algorithm is the amount of memory it needs to run to completion.
The "time complexity" of an algorithm is the amount of computer time it needs to run to completion.

Performance evaluation can be loosely divided into two major phases :

(1) a priori estimates - performance analysis
(2) a posteriori testing - performance measurement

Space Complexity

The space needed by each of these algorithms is seen to be the sum of the following components :

(1) A fixed part that is independant of the characteristics of the inputs and outputs. This part typically includes the instruction space, space for simple variables and fixed-size component variables, space for constants.

(2) A variable part that consists of the space needed by component variables whose size is dependant on the particular problem instance being solved, the space needed by referenced variables, and the recursion stack space.

The space requirement $S(P)$ of any algorithm P may therefore be written as

$$S(P) = c + S_p(\text{instance characteristics})$$

where c is a constant.

Estimating S_p(instance characteristics) is solely concentrated when analyzing the space complexity of an algorithm.

Examples :

(1)

```
1  Algorithm abc(a, b, c)
2  {
3      return a+b+b*c+(a+b-c)/(a+b)+4.0;
4  }
```

The problem instance is characterized by the specific values of a, b, and c. Assuming one word is adequate to store the values of each of a, b, c, and the result. The space needed by 'abc' is independant of the instance characteristics. Consequently, $Sp = 0$.

(2)

```
1  Algorithm Sum(a, n)
2  {
3      s := 0.0;
4      for i := 1 to n do
5          s := s + a[i];
6      return s;
7  }
```

The problem instances are characterized by n, the number of elements to be summed. The space needed by n is one word, since it is of type integer. The space needed by a is

the space needed by variables of type array of floating point numbers. This is atleast n words, since a must be large enough to hold the n elements to be summed. So,

$S_{sum}(n) \geq (n+3)$ [n for a[ ], one each for n, i and s].

(3)

```
1 Algorithm RSum(a, n)
2 {
3   if (n ≤ 0) then return 0.0;
4   else return RSum(a, n-1) + a[n];
5 }
```

The problem instances of 'Sum' are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assuming the return address requires only one word of memory. Each call to 'RSum' requires atleast three words (including space for n, the return address, and a pointer to a[]). Since the depth of recursion is n+1, the recursion stack space needed is

$S_{RSum}(n) \geq 3(n+1)$.

Time Complexity

The time T(P) taken by a program P is the sum of the compile time and the run time. The compile time does not depend on the instance characteristics. Assuming a compiled program will run several times without recompilation. Consequently, only run time of a program is considered. This run time is denoted by t_p (instance characteristics).

Because many of the factors t_p depends on are not known at the time a program is conceived, t_p can be only estimated. Obtaining an exact formula is in itself an impossible task. To solve a problem instance with characteristics given by 'n', obtain a count for the total number of operations, and count only the number of program steps.

A "program step" is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independant of the instance characteristics. The number of steps any program statement is assigned depends on the kind of statement.

- Comments count as zero steps.
- assignment statement count as one step.
- iterative statements (for, while, repeat-until) step counts for the control part of the statement.

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

- while is given a step count equal to the number of steps counts assignable to <expr>.
- for step count is one, unless the counts attributable to <expr> are functions of the instance characteristics.

Determining the number of steps needed by a program to solve a problem instance

① First Method -

A new variable, 'count', is introduced into the program. This is a global variable with initial value 0. Statements to increment count by the appropriate amount are introduced into the program. Each time a statement in the original program is executed, count is incremented by the step count of that statement.

Example 1:

```
1 Algorithm Sum(a,n)
2 {
3       s:= 0.0;
4       for i:=1 to n do
5           s:= s+a[i];
6       return s;
7 }
```

```
1  Algorithm Sum(a,n)
2  {
3   s:= 0.0;
4   count:=count+1;
5   for i:= 1 to n do
6   {
7     count := count + 1;
8     s:= s+a[i]; count:=count+1;
9   }
10  count:= count+1;
11  count := count+1;
12  return s;
13 }
```

The change in the value of count by the time this program terminates is the number of steps executed by the algorithm. In the 'for' loop the value of count will increase by a total of $2n$. If the count is zero to start with, then it will be $2n+3$ on termination.

Example 2 :

```
1 Algorithm RSum(a,n)
2 {
3   if (n ≤ 0) then return 0.0;
4   else  return RSum(a, n-1) + a[n];
5 }
```

```
1 Algorithm  RSum(a, n)
2 {
3    count := count + 1;
4    if (n ≤ 0) then
5    {
6       count := count + 1;
7       return 0.0;
8    }
9    else
10   {
11      count := count + 1;
12      return RSum(a, n-1) + a[n];
13   }
15 }
```

When the statements to increment count are introduced into Algorithm. When $n > 0$, count increases by 2 plus whatever increase results from the invocation of RSum from within the 'else' clause. Let $t_{RSum}(n)$ be the increase in the value of count. $t_{RSum}(n) = 2$. From the definition of t_{RSum}, it follows that this additional increase is $t_{RSum}(n-1)$. So, if the value of count is zero initially, its value at the time of termination is $\underline{2 + t_{RSum}(n-1), n > 0}$.

When analyzing a recursive program for its step count, a recursive formula for the step count is obtained

$$t_{RSum}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{RSum}(n-1) & \text{if } n > 0 \end{cases}$$

These recursive formulas are referred to as "Recursive Relations".

One way of solving any such recurrence relation is to make repeated substitutions for each occurence of the function t_{RSum} on the right-hand side until all such occurences disappear:

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + 2 + t_{RSum}(n-2) \\ &= 2(2) + t_{RSum}(n-2) \\ &\vdots \\ &= n(2) + t_{RSum}(0) \\ &= 2n + 2 \qquad n \geq 0 \end{aligned}$$

Step count for RSum is $\underline{2n+2}$.

The stepcount tells how the run time for a program changes with changes in the instance characteristics. One of the instance characteristic frequently used is the input size. The input size of any instance of a problem is defined to be the number of words needed to describe that instance.

Example 3 :

```
1 Algorithm Add (a,b,c,m,n)
2 {
3    for i := 1 to m do
4      for j := 1 to n do
5         c[i,j] := a[i,j] + b[i,j];
6 }
```

```
1  Algorithm Add (a,b,c,m,n)
2  {
3    for i := 1 to m do
4    {
5      count := count + 1;
6      for j := 1 to n do
7      {
8        count := count + 1;
9        c[i,j] := a[i,j] + b[i,j];
10       count := count + 1;
11     }
12     count := count + 1;
```

```
    }
    count := count + 1;
}
```

Algorithm has to add two $m \times n$ matrices 'a' and 'b' together. Simplified version of the above Algorithm is as follows:

```
1  Algorithm Add (a, b, c, m, n)
2  {
3     for i := 1 to m do
4     {
5        count := count + 2;
6        for j := 1 to n do
7           count := count + 2;
8     }
9     count := count + 1;
10 }
```

The line 7 is executed 'n' times for each value of i, or a total of 'mn' times; line 5 is executed 'm' times; and line 9 is executed once. If count is 0 to begin with, it will be $2mn + 2m + 1$ when terminated. After analyzing, $m > n$, it is better to interchange the two 'for' statements, then the step count becomes $2mn + 2n + 1$.

(2) Second Method -

To determine the step count of an algorithm is to build a table in which the total number of steps contributed by each statement is listed. This is arrived by first determining the number of steps per execution (s/e) of the statement and the total number of times i.e. frequency each statement is executed. The s/e of a statement is the amount by which the count changes as a result of the execution of that statement. By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

Example 1:

Statement	s/e	frequency	Total steps
1 Algorithm Sum(a,n)	0	–	0
2 {	0	–	0
3 s:= 0.0;	1	1	1
4 for i:=1 to n do	1	n+1	n+1
5 s:= s+a[i];	1	n	n
6 return s;	1	1	1
7 }	0	–	0
TOTAL			2n+3

Example 2 :

	s/e	frequency n=0	frequency n>0	total steps n=0	total steps n>0
1 Algorithm RSum(a,n)	0	–	–	0	0
2 {	0	–	–	0	0
3 if (n<=0) then	1	1	1	1	1
4 return 0.0;	1	1	0	1	0
5 else return	0	–	–	0	0
6 RSum(a,n-1)+a[n];	1+x	0	1	0	1+x
7 }	0	–	–	0	0
TOTAL				2	2+x

Example 3 :

	s/e	frequency	total steps
1 Algorithm Add(a,b,c,m,n)	0	–	0
2 {	0	–	0
3 for i:=1 to m do	1	m+1	m+1
4 for j:=1 to n do	1	m(n+1)	mn+m
5 c[i,j] := a[i,j]+b[i,j];	1	mn	mn
6 }	0	–	0
TOTAL			2mn+2m+1

Asymptotic Notations / Order Notations (O, Ω, Θ):

Asymptotic Notations enables to make meaningful statements about the time and space complexities of an algorithm. It is a notation that allows to characterize the main factors affecting an algorithm's running time without going into all the details of exactly how many primitive operations are performed for each constant-time set of instructions.

The functions f and g are nonnegative functions.

① Big "oh" : The function $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") iff there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n, $n > n_0$.

$O(1)$ means a computing time that is constant.
$O(n)$ is called linear,
$O(n^2)$ is called quadratic.
$O(n^3)$ is called cubic
$O(2^n)$ is called exponential
$O(\log n)$ is called logarithmic
$O(n^k)(k \geq 1)$ is called polynomial
$O(n \log n)$ is called Linearithmic
$O(n!)$ is called Factorial

SUMRANA SIDDIQUI
Asst. Professor
CSE, DSCET

If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n, then if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$.

Example: $3n+2 = O(n)$

$3n+2 \leq 4n$ for all $n \geq 2$

$n_0 = 2 \quad c = 4 \quad g(n) = n \Rightarrow f(n) = O(g(n))$

$10n^2+4n+2 = O(n^2)$

$10n^2+4n+2 \leq 11n^2$ for all $n \geq 5$

$n_0 = 5 \quad c = 11 \quad g(n) = n^2$

$\Rightarrow f(n) = O(g(n))$

$3n+3 = O(n)$

$3n+3 \leq 4n$ for all $n \geq 3$

$n_0 = 3 \quad c = 4 \quad g(n) = n$

$\Rightarrow f(n) = O(g(n))$

[$f(n) = O(g(n))$ states that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$.]

② **Omega** : The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n > n_0$.

Example: $3n+2 = \Omega(n)$

$3n+2 \geq 3n$ for all $n \geq 1$

$n_0 = 1 \quad c = 3 \quad g(n) = n$

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

$f(n) = \Omega(g(n))$ states that $g(n)$ is only a lower bound on $f(n)$.

(3) Thelā : The function $f(n) = \theta(g(n))$ (read as "f of n is theta of g of n") iff there exists positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n > n_0$.

Example :

$3n + 2 = \theta(n)$

$3n+2 \geq 3n$ for all $n \geq 2$
$3n+2 \leq 4n$ for all $n \geq 2$
$\Rightarrow f(n) = \theta(g(n))$

$c_1 = 3$, $c_2 = 4$, $n_0 = 2$, $g(n) = n$

The 'theta' notation is more precise than both the 'big-oh' and 'omega' notations. The function $f(n) = \theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

(4) Little "oh" : The function $f(n) = o(g(n))$ (read as "f of n is little oh of g of n") iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

Example: $f(n) = 2^n$, then $f(n) = o(n!)$

$$f(n) = \lim_{n\to\infty} \frac{2^n}{n!}$$

$$= \lim_{n\to\infty} \frac{2^n}{n(n-1)(n-2)\ldots 1}$$

As the denominator is 0, as n is large and approaches infinity.

Therefore, the result = 0. This implies $f(n) = 2^n = o(n!)$

(5) **Little omega**: The function $f(n) = \omega(g(n))$ (read as "f of n is little omega of g of n") iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Example: Prove $n! = \omega(2^n)$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \quad \text{for } f(n) = \omega(g(n))$$

$$n! \ \underline{v} \ n^n$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{2^n}{n!} = \frac{2^n}{n(n-1)(n-2)\ldots 1}$$

As the Denominator becomes 0. Result = 0.

Thus $n! = \omega(2^n)$

Practical Complexities : The time complexity of an algorithm is generally some function of instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change.

The complexity function can also be used to compare two algorithms P and Q that perform the same task.

Assuming algorithm P has complexity $O(n)$ and algorithm Q has complexity $O(n^2)$. The algorithm P is faster than algorithm Q for sufficiently large n.

Analysis of how various functions grow with 'n'.

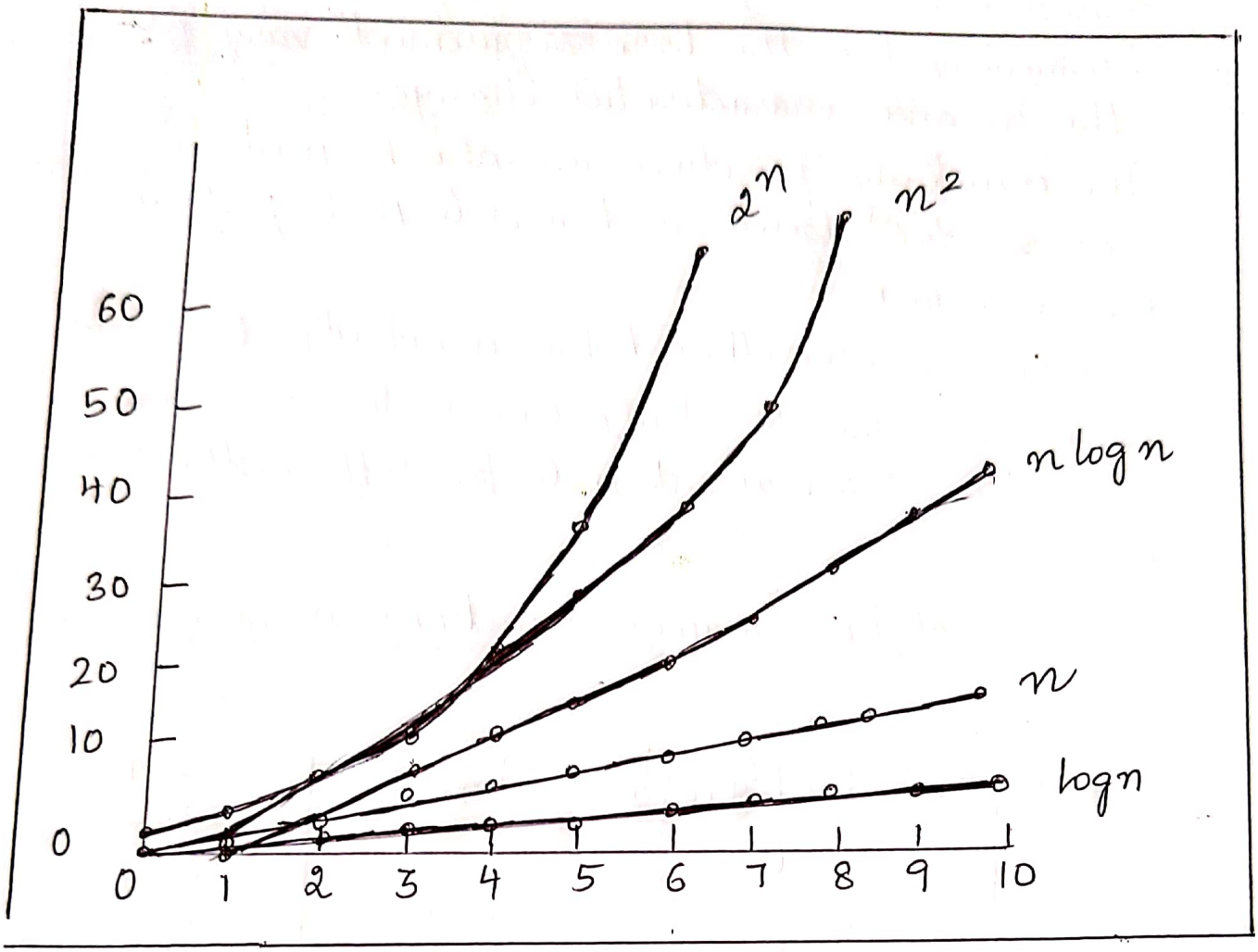
$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	68	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Table with Function values

The above table shows that the function 2^n grows very rapidly with n. It can be concluded that the utility of algorithms with exponential complexity is limited to small n.

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

Plot of the above function values



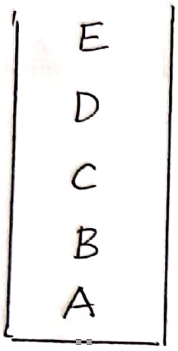
Elementary Data Structures:

→ Stacks and Queues

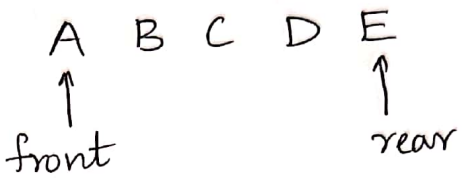
One of the most common forms of data organization in computer programs is the ordered or linear list, which is written as $a = (a_1, a_2, \ldots, a_n)$. The a_i's are referred to as atoms and they are chosen from some set. The null or empty list has $n = 0$ elements.

- A <u>stack</u> is an ordered list in which all insertions and deletions are made at one end, called the 'top'.
- A <u>queue</u> is an ordered list in which all insertions take place at one end, the 'rear', whereas all deletions take place at the other end, the 'front'.
- The operations of a stack imply that if the elements A, B, C, D, and E are inserted into a stack, in that order, then the first element to be removed (deleted) must be E. The last element to be inserted into the stack is the first to be removed. Stacks are sometimes referred to as Last In First Out (LIFO) lists.
- The operations of a queue require that

the first element that is inserted into the queue is the first one to be removed. Thus queues are known as First In First Out (FIFO) lists.



Stack



queue

The simplest way to represent a stack is by using a 1-D array. stack[0 : n-1], where n is the maximum number of allowable entries. The first or bottom element in the stack is stored at stack[0], the second stack[1], and the ith at stack[i-1]. "top" is a variable, which points to the top element in the stack. The stack is empty "if (top < 0)", if not, the topmost element is at stack[top]. The stack is full "if (top ≥ n-1)". Two more substantial operations are inserting and deleting elements. The corresponding algorithms are Add and Delete are as follows:

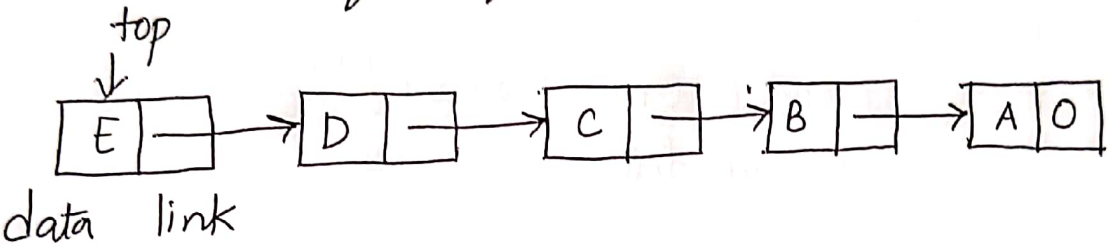
Operations on a stack

```
1 Algorithm Add (item)
2 // Push an element in stack. Return true if succesful.
3 // else return false. item is used as an input.
4 {
5    if (top ≥ n-1) then
6    {
7       write ("Stack is full!"); return false;
8    }
9    else
10   {
11      top := top + 1; stack[top] := item; return true;
12   }
13 }
```

```
1 Algorithm Delete (item)
2 // Pop the top element from stack. Return true if succesful
3 // else return false. item is used as an output.
4 {
5    if (top < 0) then
6    {
7       write ("Stack is empty!"); return false;
8    }
9    else
10   {
11      item := stack[top]; top := top-1; return true;
12   }
13 }
```

→ Another way to represent a stack is by using links (or pointers).

A node is a collection of data and link information. A stack can be represented by using nodes with two fields, called data and link. The data field of each node contains an item in the stack and the corresponding link field points to the node containing the next item in the stack. The link field of the last node is zero.



The variable top points to the topmost node in the list. The empty stack is represented by setting top := 0. Because of the way the links are pointing, insertion and deletion are easy to accomplish.

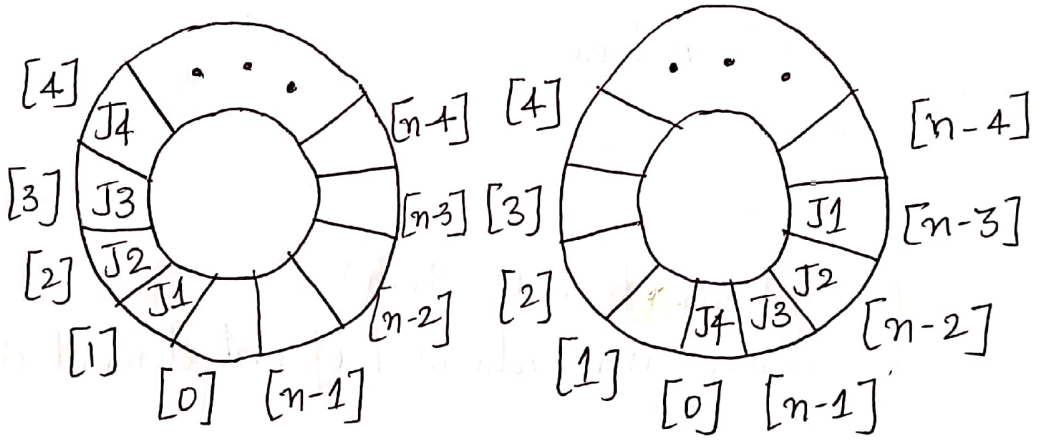
Linked representation of Stack

```
// Type is the type of data.
node = record
{
    Type data ; node *link ;
}
```

```
1  Algorithm Add(item)
2  {
3    // Get a new node.
4    temp := new node;
5     if (temp ≠ 0) then
6     {
7       (temp → data) := item ; (temp → link) := top;
8       top := temp ; return true ;
9     }
10    else
11    {
12       write ("Out of Space!");
13       return false;
14    }
15 }
```

```
1  Algorithm Delete (item)
2  {
3    if (top = 0) then
4    {
5       write ("Stack is Empty !");
6       return false;
7    }
8    else
9    {
10      item := (top → data); temp := top;
11      top := (top → link);
12      delete temp; return true;
13   }
14 }
```

An efficient queue representation can be obtained by taking an array q[0 : n-1] and treating it as if it were circular. Elements are inserted by increasing the variable 'rear' to the next free position. When 'rear = n-1' the next element is entered at q[0] in case that spot is free. The variable 'front' always points one position counterclockwise from the first element in the queue. The variable 'front = rear' if and only if the queue is empty and initially it is set as 'front := rear := 0'.



front = 0; rear = 4 front = n-4; rear = 0

Operations on Queue

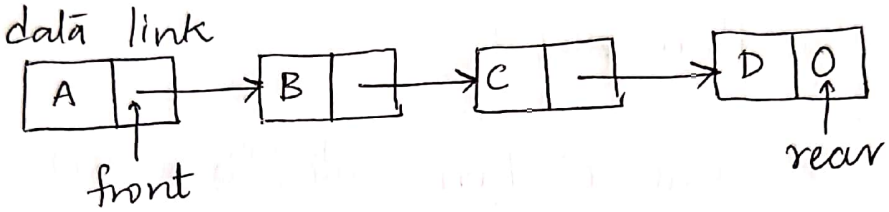
```
1 Algorithm AddQ(item)
2 // Insert item in the circular queue stored in q[0:n-1].
3 // rear points to the last item, and front is one
```

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

```
4 // position counterclockwise from the first item in q.
5 {
6   rear := (rear + 1) mod n;
7   if (front = rear) then
8   {
9     write ("Queue is full !");
10    if (front = 0) then rear := n - 1;
11    else rear := rear - 1;
12    // Move rear one position counterclockwise.
13    return false;
14  }
15  else
16  {
17    q[rear] := item; // Insert new item.
18    return true;
19  }
20 }
```

```
1  Algorithm DeleteQ(item)
2  // Removes and returns the front element of queue[0:n-1]
3  {
4    if (front = rear) then
5    {
6      write ("Queue is Empty !");
7      return false;
8    }
9    else
10   {
11     front := (front + 1) mod n; //Advance front clockwise.
12     item := q[front]; //Set item to front of queue.
13
14   } return true;
15 }
```

Another way to represent a queue is by using links. Each node of the queue is composed of the two fields "data" and "link".
A queue is pointed at by two variables, "front" and "rear". Deletions are made from the front and insertions at the rear. Variable "front=0" signals an empty queue.



Linked Representation of Queue

```
// Type is the type of data
node = record
{
   Type data ; node *link ;
}
```

```
1 Algorithm  AddQ(item)
2 {
3   // Get a new node
4   temp := new node;
5   temp → data := item;
6   if (front = 0) then
7     {
8       front := temp;
9       return false;
10    }
```

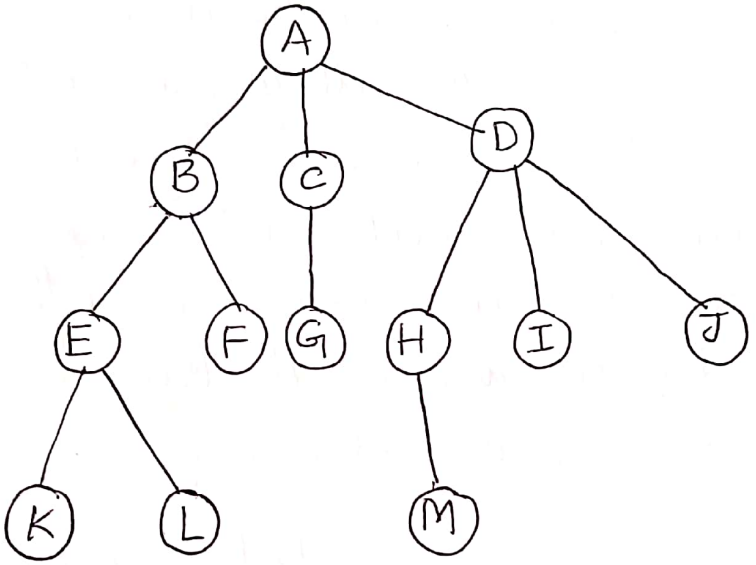
SUMRANA SIDDIQUI
Asst. Professor
CSE

```
11  else
12   {
13      rear -> link = temp; return true;
14   }
15    rear := temp;
16    rear -> link := front;
17  }
```

```
1  Algorithm DelQ()
2  {
3    if (front = 0) then write ("Queue is empty");
4    if (front = rear)
5     {
6       item := front -> data;
7       delete front; front := 0; rear := 0;
8     }
9    else
10    {
11      temp := front;
12      item := temp -> data;
13      front := front -> link;
14      rear -> link = front;
15      delete temp;
16    }
17   return item;
18 }
```

Trees

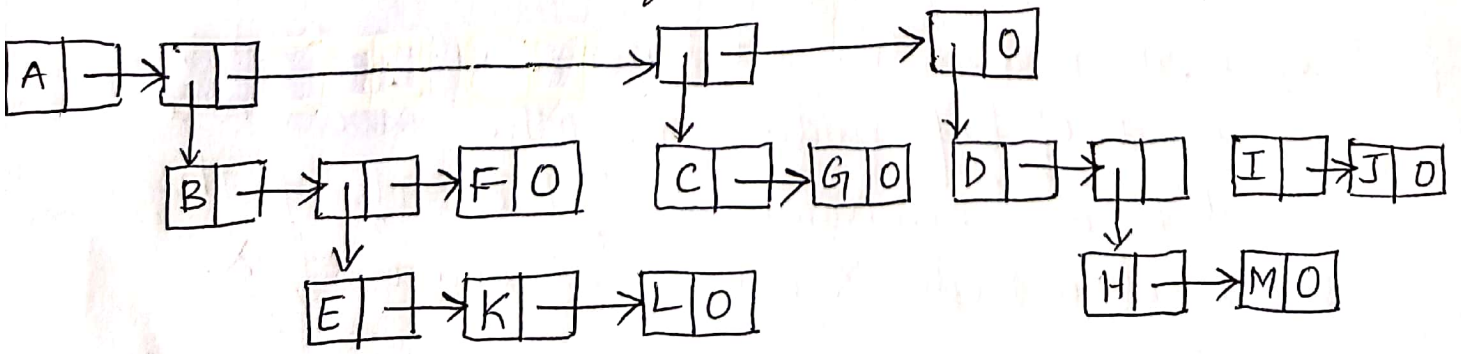
A tree is a finite set of one or more nodes such that there is a specially designated node called the root and the remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots T_n$, where each of these sets is a tree. The sets $T_1, \ldots T_n$ are called the subtrees of the root.



This tree has 13 nodes. The root contains A. The number of subtrees of a node is called its 'degree'. The degree of A is 3, of C is 1, and of F is 0. Nodes that have degree zero called 'leaf' or 'terminal' nodes. The set $\{K, L, F, G, M, I, J\}$ is the set of leaf nodes. The other nodes are referred to as 'non terminals'. The roots of the subtree of a node X are the 'children' of X.

The node X is the 'parent' of its children. Thus the children of D are H, I and J, and the parent of D is A. Children of the same parent are said to be 'siblings'. H, I and J are siblings. The 'degree' of a tree is the maximum degree of the nodes in the tree. The tree above has degree 3. The 'ancestors' of a node are all the nodes along the path from the root to that node. The ancestors of M are A, D, and H. The 'level' of a node is defined by initially letting the root be at level one. If a node is at level p, then its children are at level 'p+1'. The 'height' or 'depth' of a tree is defined to be the maximum level of any node in the tree. A 'forest' is a set of $n \geq 0$ disjoint trees.

Trees can be represented using a linked list in which one node corresponds to one node in the tree, then a node must have a varying number of fields depending on the number of children.



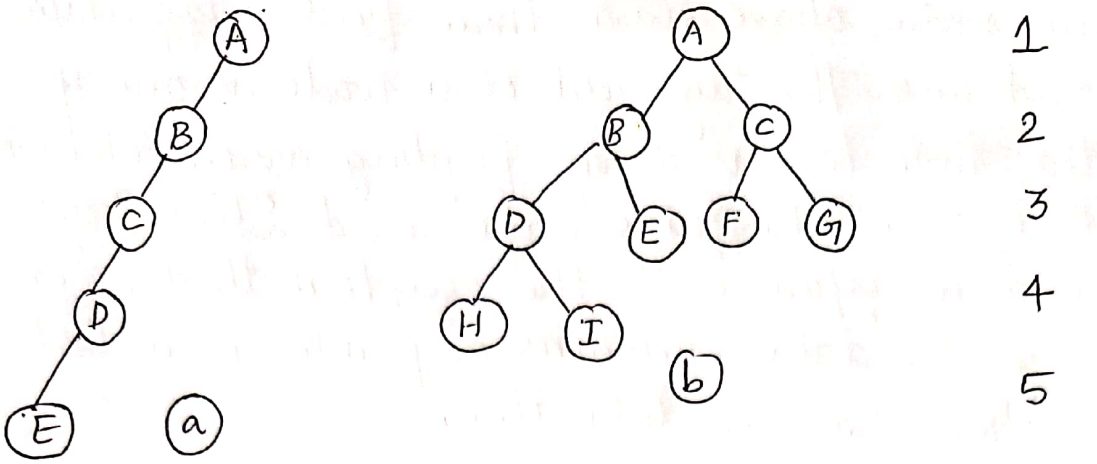
The nodes above have three fields : tag, data and link. The tag field of a node, is one if the node has a down-pointing arrow; otherwise it is zero. The fields "data" and "link" are used as before with the exception that when "tag = 1", "data" contains a pointer to a list rather than a data item.

Binary Trees

A binary tree is an important type of tree structure that occurs very often. It is characterized by the fact that any node can have at most two children i.e. there is no node with degree greater than two. For binary trees the subtree on the left and on the right can be distinguished, whereas for other trees the order of the subtrees is irrelevant. A binary tree is allowed to have zero nodes whereas any other tree must have at least one node.

A "binary tree" is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the "left" and "right" subtrees.

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

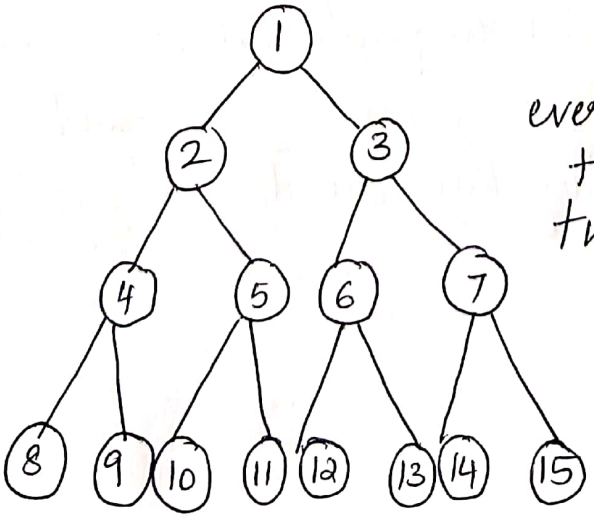


The above two trees are special kinds of binary trees. Fig (a) is a skewed tree, skewed to the left. Fig (b) is called a complete binary tree.

The maximum number of nodes on level 'i' of a binary tree is 2^{i-1}.

The maximum number of nodes in a binary tree of depth 'k' is $2^k - 1$, $k > 0$.

The binary tree of depth 'k' that has exactly $2^k - 1$ nodes is called a 'full binary tree' of depth k.



A tree in which every node other than leaves has two children.

Nodes on any level are numbered from left to right.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled.

→ Sequential representation can be used for all binary trees though in most cases there is a lot of unutilized space.

For complete binary trees the representation is ideal as no space is wasted.

For skewed tree less than a third of the array is utilized.

tree

A	B	-	C	-	-	-	D	-	. . .	E

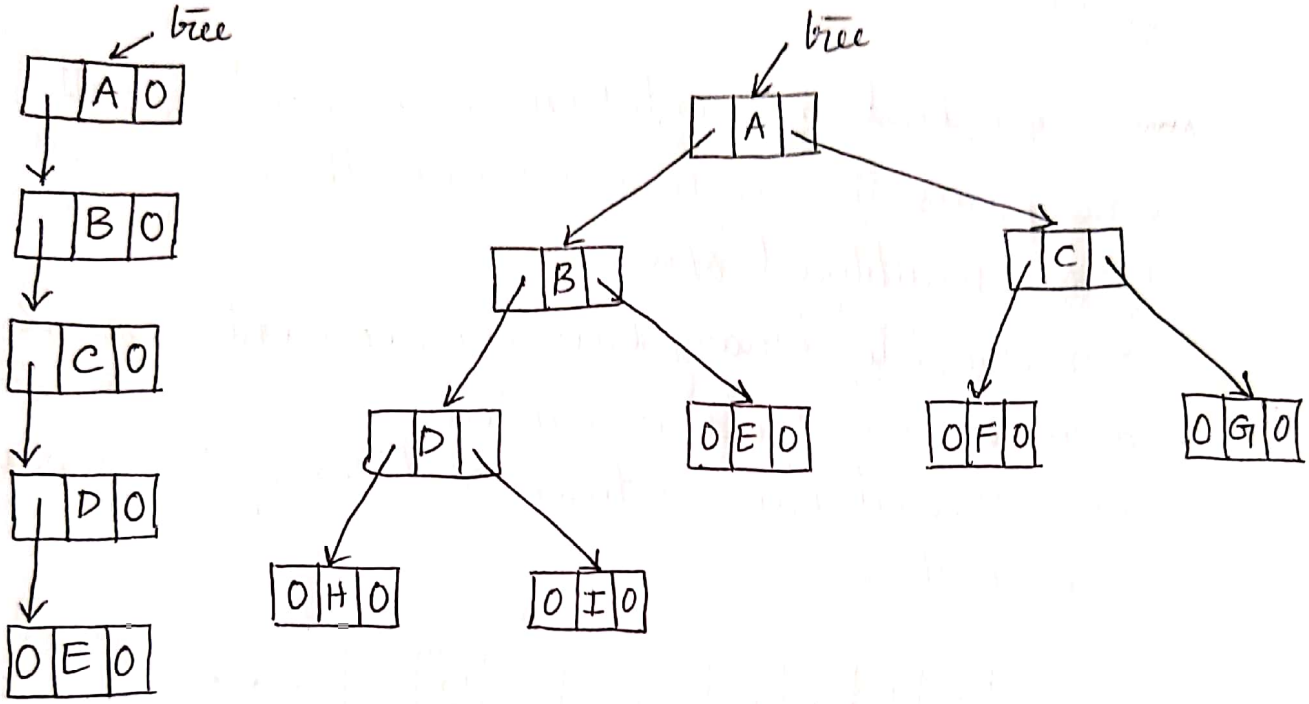
tree

A	B	C	D	E	F	G	H	I
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	. . . . .	(16)

Sequential representation suffers from the general inadequacies. Insertion or deletion of nodes requires the movement of many nodes to reflect the change in level number of the remaining nodes.

This can be easily overcome through the use of a linked representation.

⟹ In linked representation each node has three fields : 'lchild', 'data', 'rchild'. This node structure makes it difficult to determine the parent of a node. If necessary, to determine the parent of a node, a fourth field, 'parent', can be included.



Dictionaries

An abstract data type that supports the operations insert, delete, and search is called a dictionary. Dictionaries have found application in the design of numerous algorithms. A number of data structures have been devised to realize a dictionary. At a high level

Comparision Methods — eg: Binary Search Trees
Direct Access Methods — eg: Hashing

Hashing

A dictionary can even be represented by using a method called "hashing".

Hashing is a technique that makes use of a hash function for mapping pairs to the corresponding positions in a hash table. Ideally a pair 'p' with a key 'k' is stored at a position f(k) by the hash function 'f'. Each hash table position can store one pair.

In order to search for a pair with a key 'k' first f(k) is calculated to determine the existence of a pair at a position f(k) in the hash table. If a desired pair is found then it can be deleted, if required or else if it does not exist in a table then it can be inserted at a position f(k).

Hash function : A function that converts a big number to a small integer value. The mapped integer value is used as an index in hash table. A hash function maps a big number or string to a small integer that can be used as index in hash table. A good hash function should have the

following properties :

1) Efficiently computable
2) Should uniformly distribute the keys

Hash Table : An array that stores pointers to records corresponding to a given number. An entry in hash table is NIL if no existing number has hash function value equal to the index for the entry.

Collision Handling : The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions :

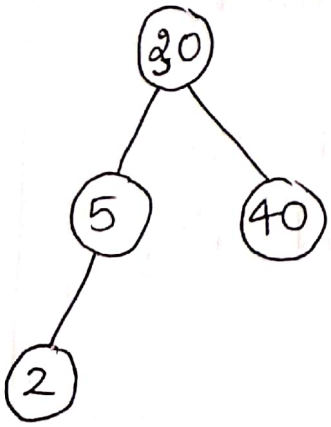
- Chaining : The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
- Open Addressing : In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, examine one by one table slots until the desired element is found or it is clear that the element is not in the table.

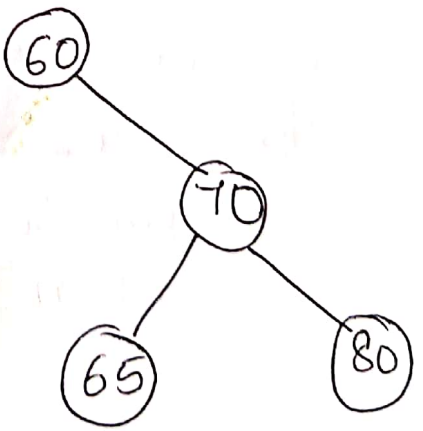
Binary Search Trees

A binary search tree is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:

1. Every element has a key and no two elements have the same key (i.e. keys are distinct)
2. The keys in the left subtree are smaller than the key in the root.
3. The keys in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

A binary search tree can support the operations search, insert and delete.





Searching a Binary Search Tree

To search for an element with key 'x', Begin at the root. If the root is 0, then the search tree contains no elements and the search is unsuccessful. Otherwise, compare 'x' with the key in the root. If 'x' equals this key, then the search terminates successfully. If 'x' is less than the key in the root, then no element in the right subtree can have key value 'x', and only the left subtree is to be searched. If 'x' is larger than the key in the root, only the right subtree needs to be searched.

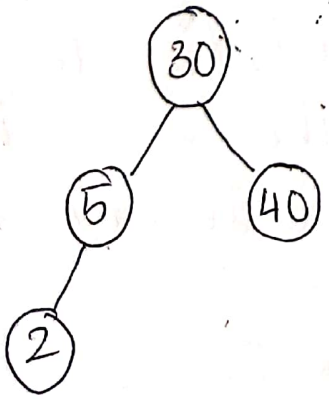
Recursive search of a binary search tree

```
1 Algorithm Search(t, x)
2 {
3   if (t = 0) then return 0;
4   else if (x = t → data) then return t;
5     else if (x < t → data) then
6       return Search(t → lchild, x);
7     else return Search(t → rchild, x);
8 }
```

Iterative search of a binary search tree

```
1  Algorithm ISearch (x)
2  {
3     found := false;
4     t := tree;
5     while ((t ≠ 0) and not found) do
6     {
7       if (x = (t → data)) then found := true;
8       else if (x < (t → data)) then t := (t → lchild);
9            else t := (t → rchild);
10    }
11    if (not found) then return 0;
12    else return t;
13 }
```

To search by rank, each node should have an additional field "leftsize", which is one plus the number of elements in the left subtree of the node.



The nodes with keys 2, 5, 30, and 40, respectively, have leftsize equal to 1, 2, 3 and 1.

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

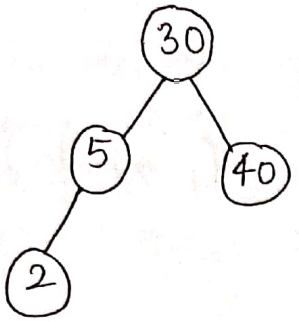
Searching a binary search tree by rank

```
1  Algorithm Searchk(k)
2  {
3   found := false ; t := tree ;
4   while ((t ≠ 0) and not found) do
5   {
6       if (k = (t → leftsize)) then found := true ;
7      else if (k < (t → leftsize)) then t := (t → lchild);
8        else
9        {
10          k := k - (t → leftsize);
11          t := (t → rchild);
12       }
13  }
14  if (not found) then return 0;
15  else return t;
16 }
```

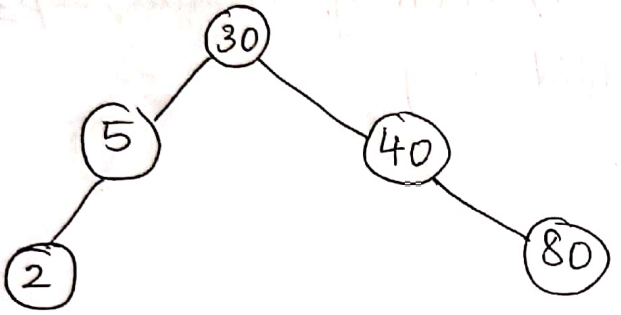
A Binary search tree of height 'h' can be searched by rank as well as key in O(h) time.

Insertion into a Binary Search Tree

To insert a new element 'x', first verify that its key is different from those of existing elements. To do this, a search is carried out. If the search is unsuccessful, then the element is inserted at the point the search terminated.



To insert 80 into the tree, search for 80. Search terminates unsuccessfully, and the last node examined is 40. 80 is inserted as the right child of this node.



This is the resulting search tree after inserting 80.

Insertion into a binary search tree

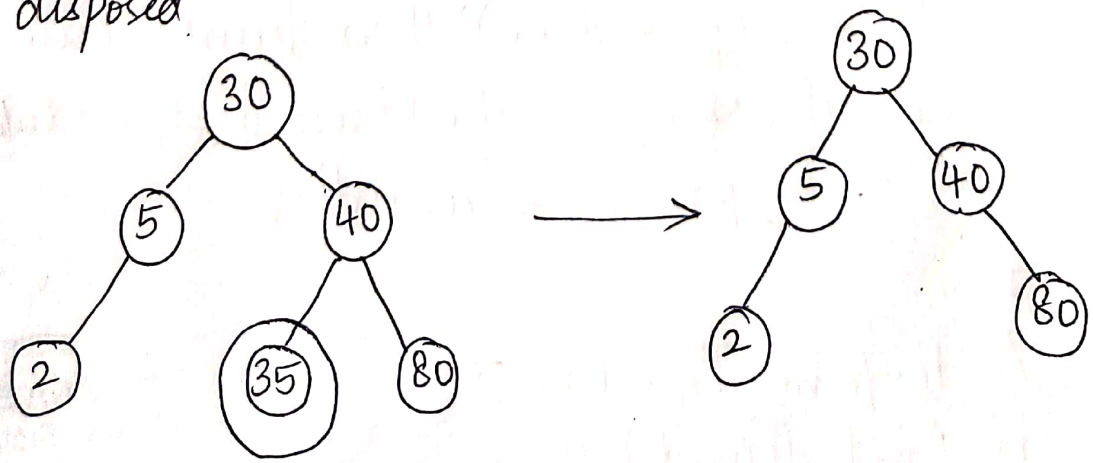
```
1 Algorithm Insert(x)
2 // Insert x into the binary search tree.
3 {
4     found := false;
5     p := tree;
6     // Search for x, q is the parent of p.
7     while ((p ≠ 0) and not found) do
8     {
9         q := p; // Save p
10        if (x = (p → data)) then found := true;
11        else if (x < (p → data)) then p := (p → lchild);
12            else p := (p → rchild);
13    }
14    // Perform Insertion
15    if (not found) then
```

```
16  {
17      p := new TreeNode;
18      (p → lchild) := 0; (p → rchild) := 0; (p → data) := x;
19      if (tree ≠ 0) then
20      {
21          if (x < (q → data)) then (q → lchild) := p;
22          else (q → rchild) := p;
23      }
24      else tree := p;
25  }
26  }
```

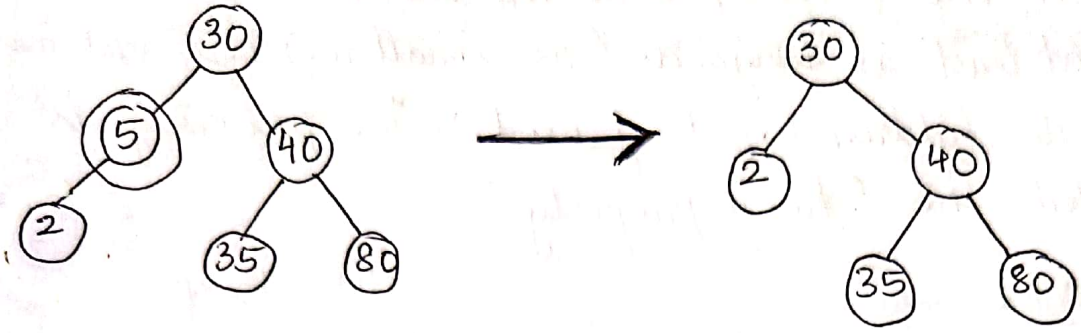
The insertion can be performed in O(h) time, where 'h' is the height of the search tree.

Deletion from a Binary Search Tree

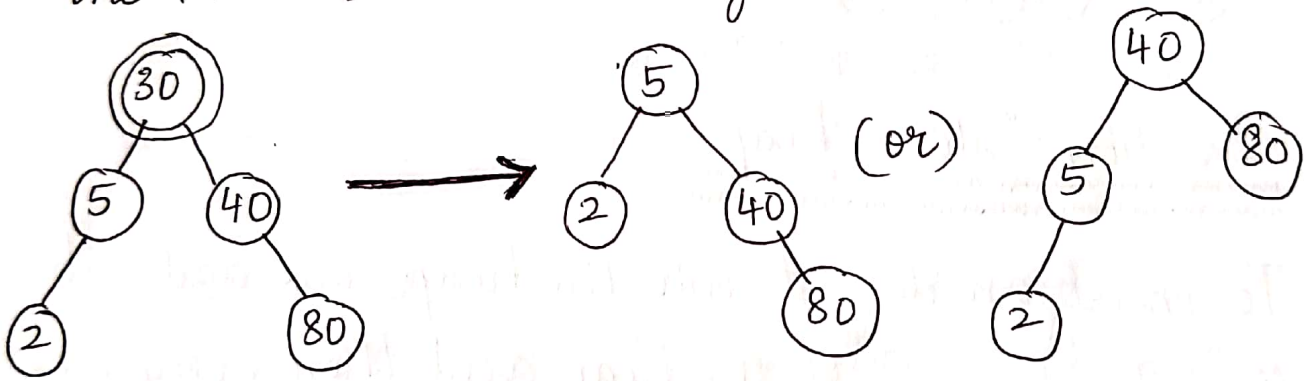
Deletion of a leaf element is easy.
To delete 35 from the tree below, the left child field of its parent is set to 0 and the node disposed



The deletion of a non leaf element that has only one child is done as: the node containing the element to be deleted is disposed, and the single child takes the place of the disposed node.



When the element to be deleted is in a nonleaf node that has two children, the element is replaced by either the largest element in its left subtree or the smallest one in its right subtree.

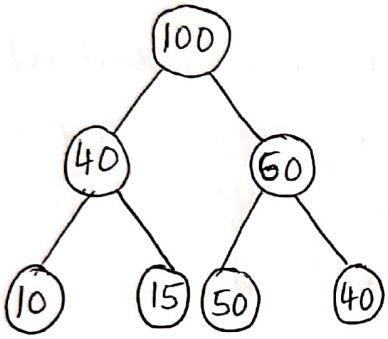


A deletion can be performed in O(h) time if the search tree has a height of 'h'.

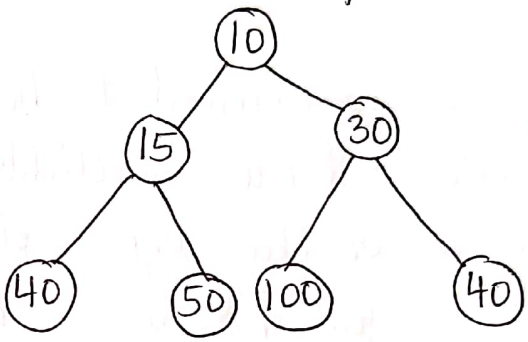
Heaps :

A max (min) heap is a complete binary tree with the property that the value at each node is at least as large as (as small as) the values at its children (if they exist). This property is called the "heap property".

Max Heap

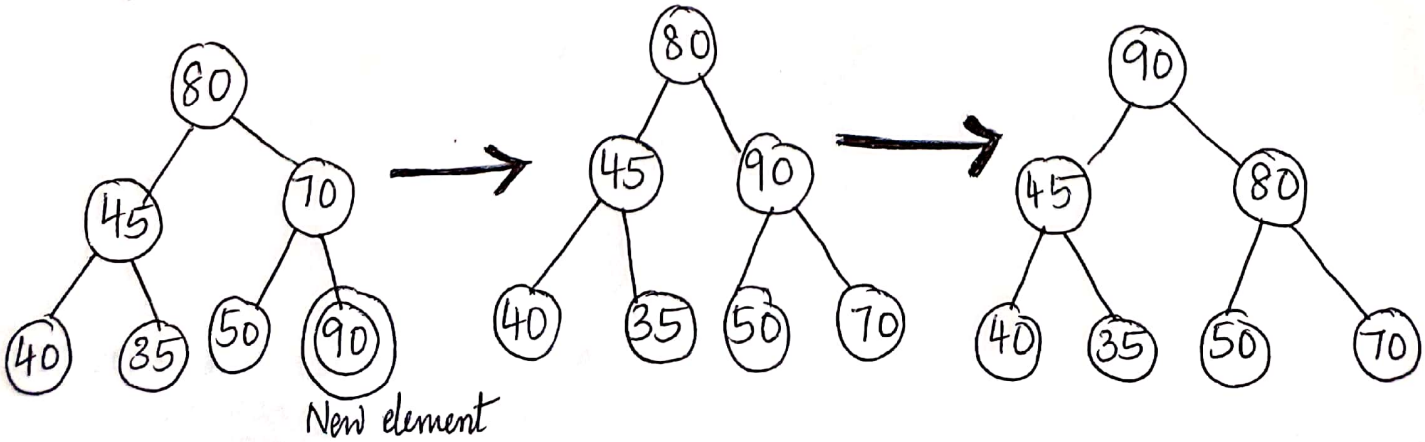


Min Heap



Insertion into a heap

To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, greatgrandparent, and so on, until it is less than or equal to one of these values.



Insertion into a Heap

```
1 Algorithm Insert (a, n)
2 {
3   // Inserts a[n] into the heap which is stored in a[1:n-1]
4     i := n ; item := a[n] ;
5     while ((i > 1) and (a[⌊i/2⌋] < item)) do
6     {
7         a[i] := a[⌊i/2⌋] ; i := ⌊i/2⌋ ;
8     }
9     a[i] := item ; return true ;
10 }
```

In the best case the new element is correctly positioned initially and no values need to be rearranged.
In the worst case the number of executions of the while loop is proportional to the number of levels in the heap.
Thus, if there are 'n' elements in the heap, inserting a new element takes $O(\log n)$ time in the worst case.

Deletion from a heap

To delete a maximum key from the max heap an algorithm called 'Adjust' is used.
'Adjust' takes as input the array a[ ] and the integers 'i' and 'n'. It regards a[1:n] as a complete binary tree. If the subtrees rooted at

$2i$ and $2i+1$ are already max heaps, then 'Adjust' will rearrange elements of a[] such that the tree rooted at i is also a max heap. The maximum element from the max heap a[1:n] can be deleted by deleting the root of the corresponding complete binary tree. The last element of the array, i.e., a[n] is copied to the root, and finally Adjust(a, 1, n-1) is called.

If there are 'n' elements in a heap, deleting the maximum can be done in O(log n) time.

```
1  Algorithm Adjust (a,i,n)
2  // The complete binary trees with roots 2i and 2i+1 are
3  // combined with node i to form a heap rooted at i.
4  // No node has an address greater than n or less than 1.
5  {
6      j := 2i ; item := a[i];
7      while (j≤n) do
8      {
9          if ((j<n) and (a[j] <a[j+1])) then j:=j+1;
10             // Compare left and right child
11             // and let j be the larger child.
12         if (item ≥ a[j]) then break;
13                 // A position for item is found.
14         a[⌊j/2⌋] := a[j] ;  j := 2j ;
15     }
16     a[⌊j/2⌋] := item ;
17 }
```

Deletion from a heap

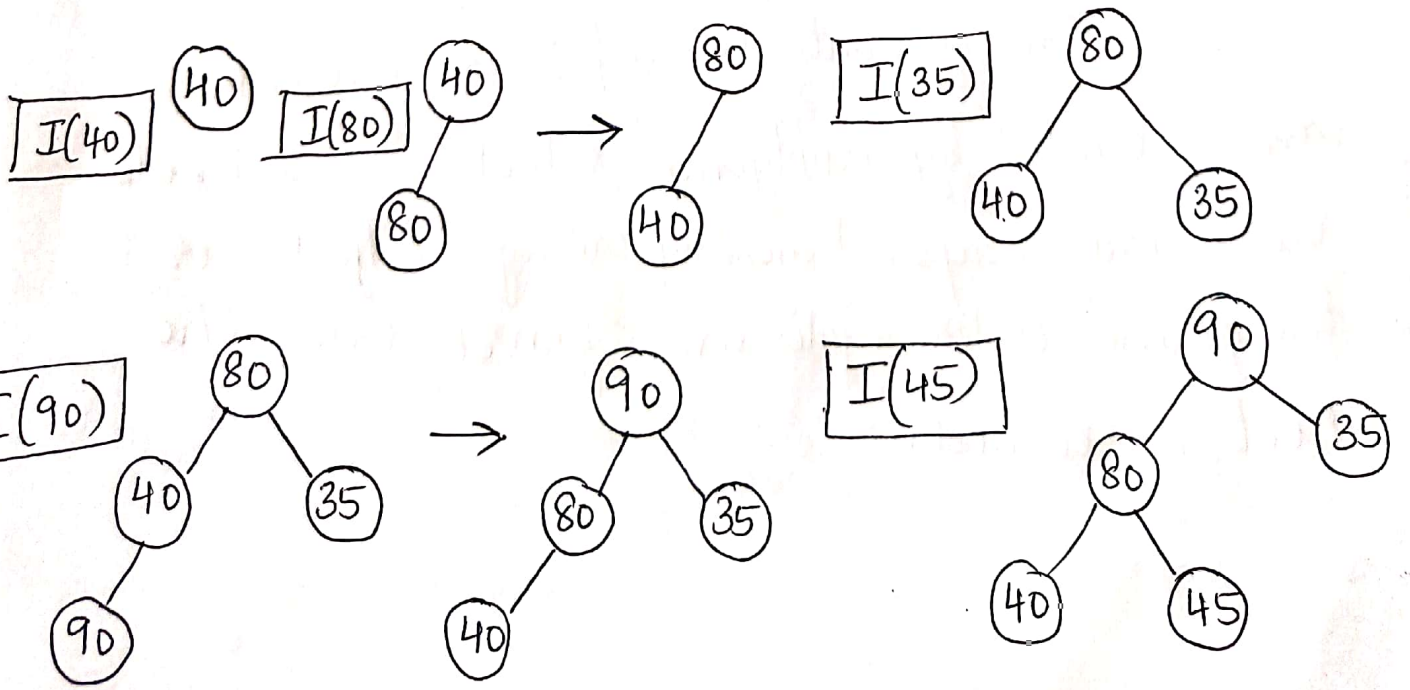
```
1  Algorithm DelMax (a, n, x)
2  // Delete the maximum from the heap a[1:n] and store it in x.
3  {
4      if (n = 0) then
5      {
6          write ("heap is empty"); return false;
7      }
8      x := a[1] ; a[1] := a[n] ;
9      Adjust (a, 1, n-1) ; return true ;
10 }
```

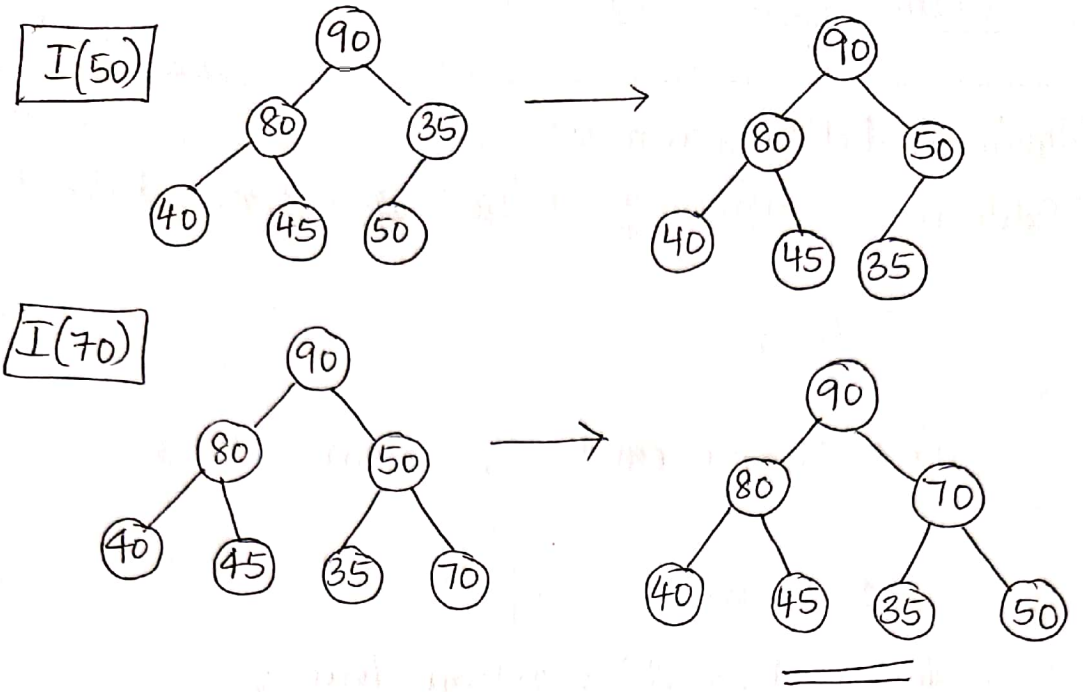
Forming a heap

It turns out n elements can be inserted into a heap faster than applying 'Insert' n times.

Example :

Form a heap from the set {40, 80, 35, 90, 45, 50, 70}



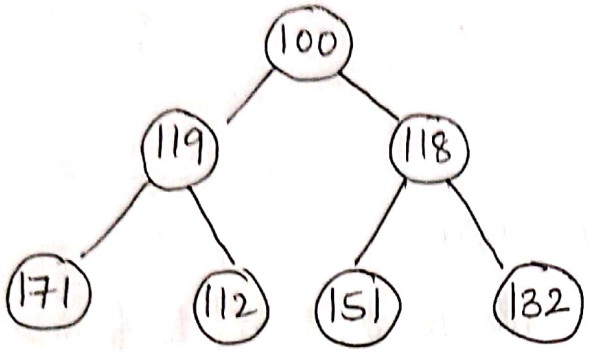


An algorithm can be devised that can perform 'n' inserts in $O(n)$ time rather than $O(n \log n)$. This reduction is achieved by an algorithm that regards any array $a[1:n]$ as a complete binary tree and works from the leaves upto the root, level by level. At each level, the left and right subtrees of any node are heaps. Only the value in the root node may violate the heap property.

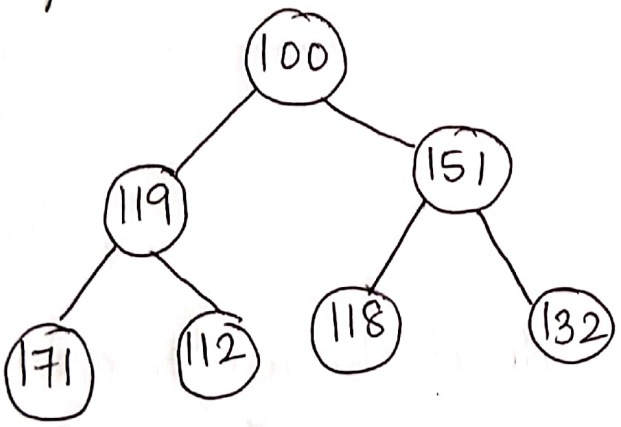
Given 'n' elements in $a[1:n]$, a heap can be created by applying 'Adjust'. The leaf nodes are already heaps. Begin by calling 'Adjust' for the parents of leaf nodes level by level, until the root is reached.

Example: "Heapify" creating a heap out of 7 elements : {100, 119, 118, 171, 112, 151 and 132}.

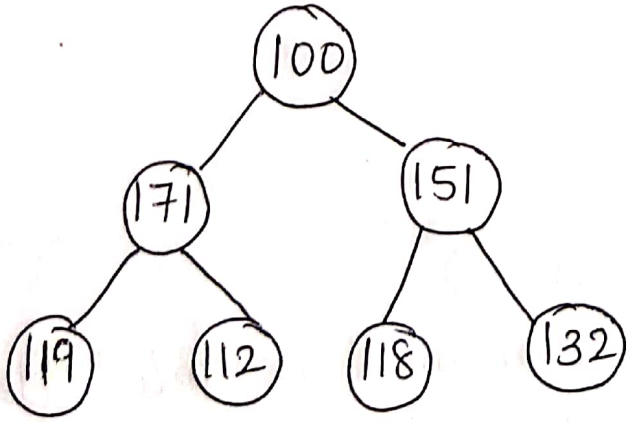
The initial tree is drawn



Since $n=7$, first call to Adjust has $i=3$. The three elements 118, 151 and 132 are rearranged to form a heap.

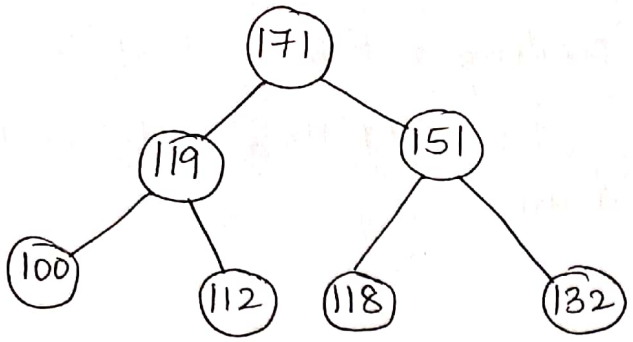


Adjust with $i=2$ gives the following tree :



SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

Adjust with $i=1$ gives the following tree :




```
1 Algorithm Heapify (a,n)
2 // Readjust the elements in a[1:n] to form a heap.
3 {
4     for i := ⌊n/2⌋ to 1 step -1 do Adjust(a,i,n);
5 }
```

→ Heap Sort

The best known example of the use of a heap arises in its application to sorting. A heap sort algorithm that incorporates 'n' elements can be inserted in O(n) is below:

Heap Sort

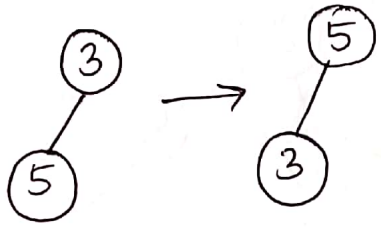
```
1 Algorithm HeapSort (a,n)
2 // a[1:n] contains n elements to be sorted. HeapSort
3 // rearranges them inplace into non decreasing order
4 {
5     Heapify (a,n); // Transform array into heap
```

```
6 // Interchange the new maximum with the element
7 // at the end of the array.
8 for i:=n to 2 step -1 do
9 {
10      t:=a[i]; a[i]:=a[1]; a[1]:=t;
11          Adjust (a, 1, i-1);
12 }
13 }
```

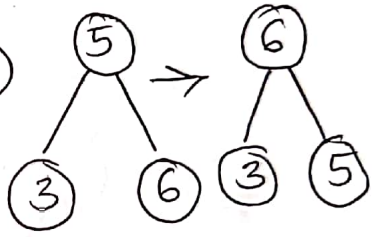
Example: Perform Heap Sort on the following numbers:
{3, 5, 6, 7, 20, 8, 2}

I(3)

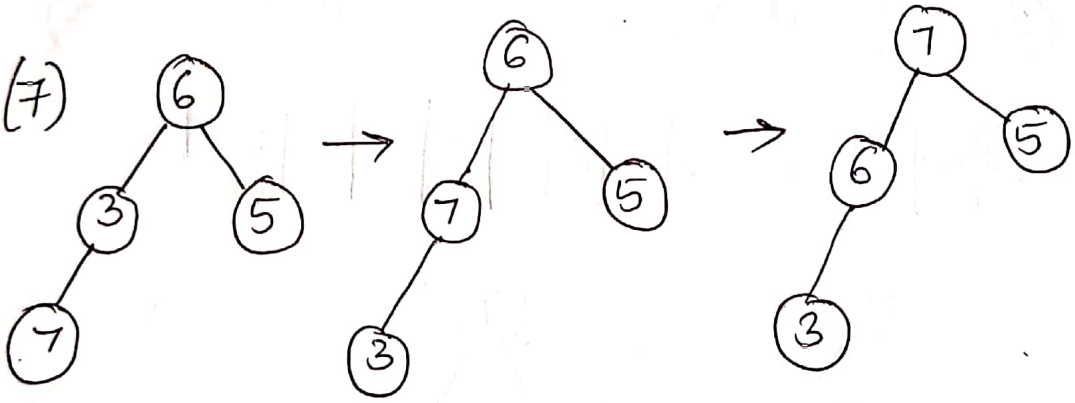
I(5)



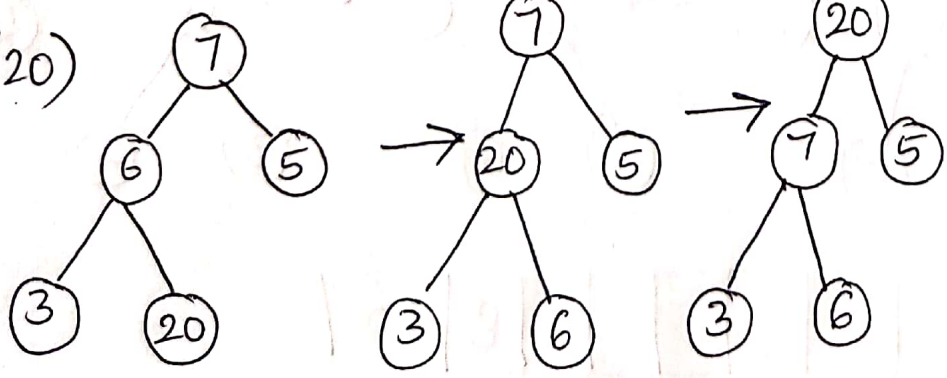
I(6)



I(7)

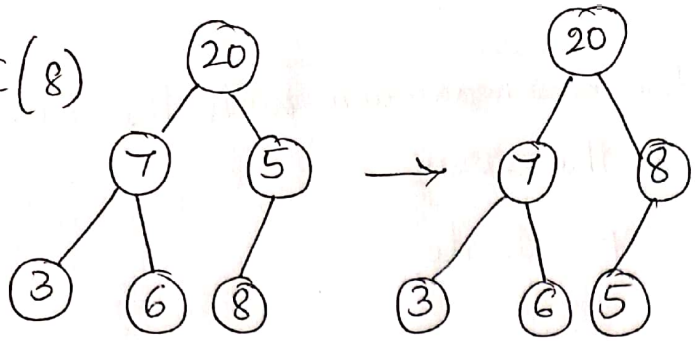


I(20)

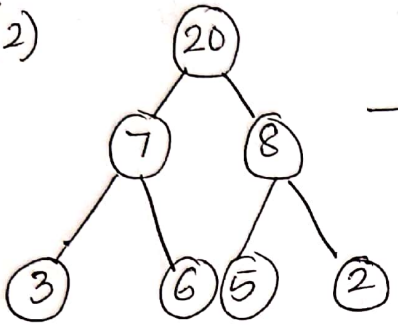


SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

I(8)

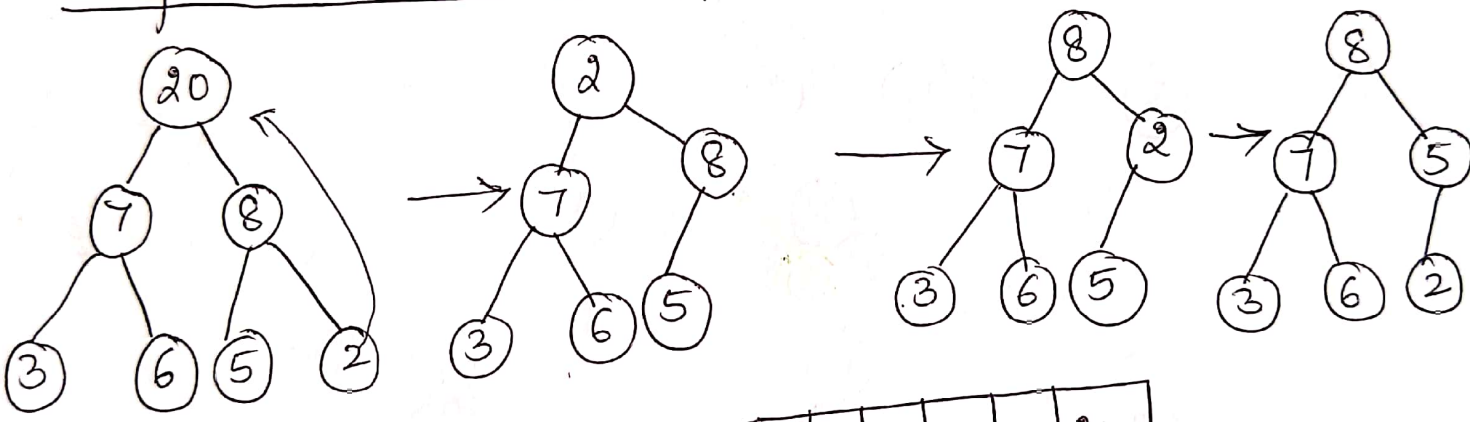


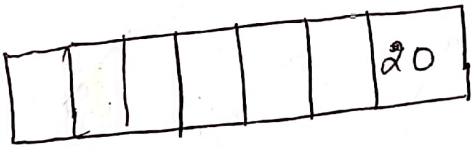
I(2)

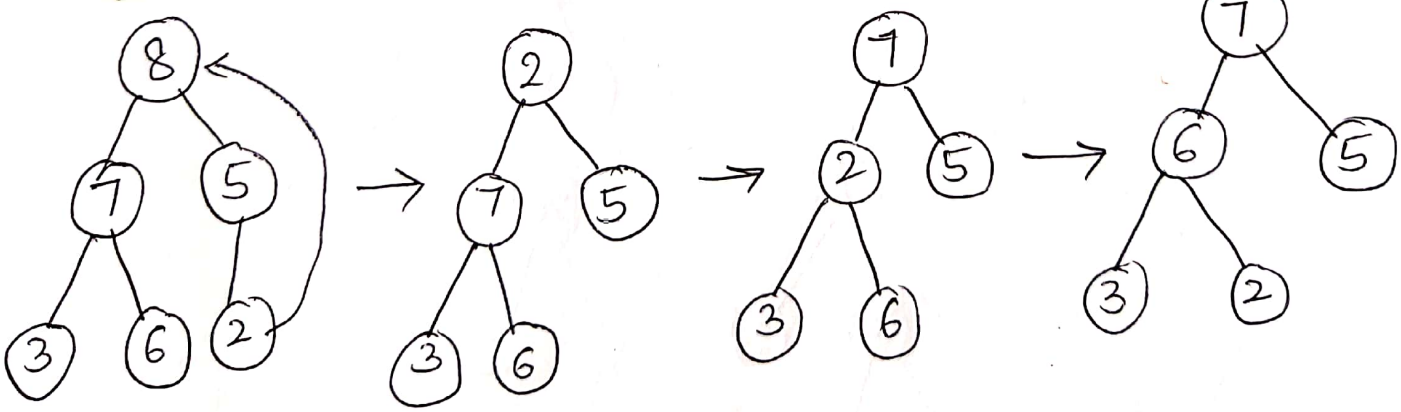


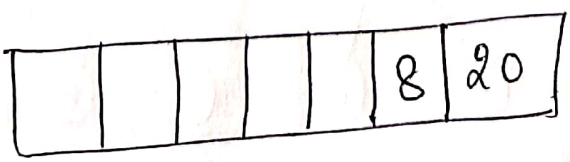
→ Max Heap

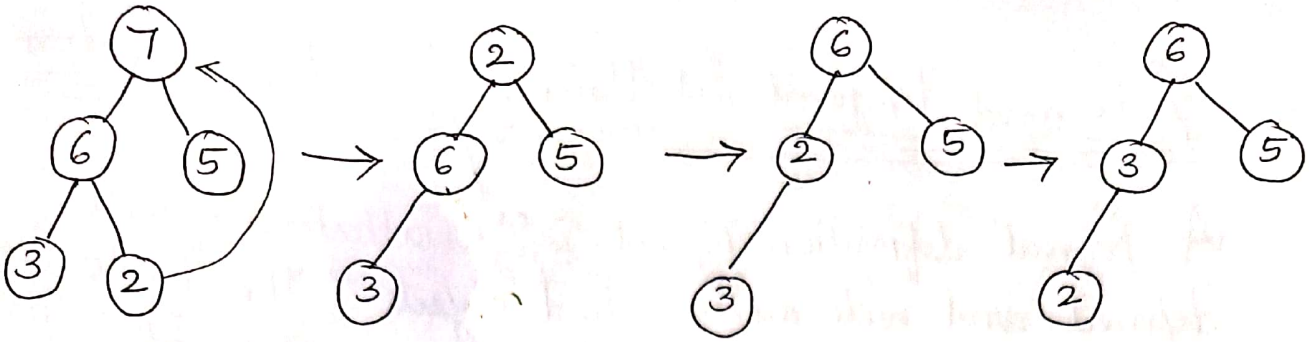
Heap Sort on the heap above is as follows :

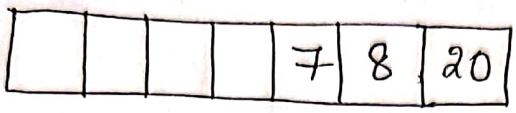


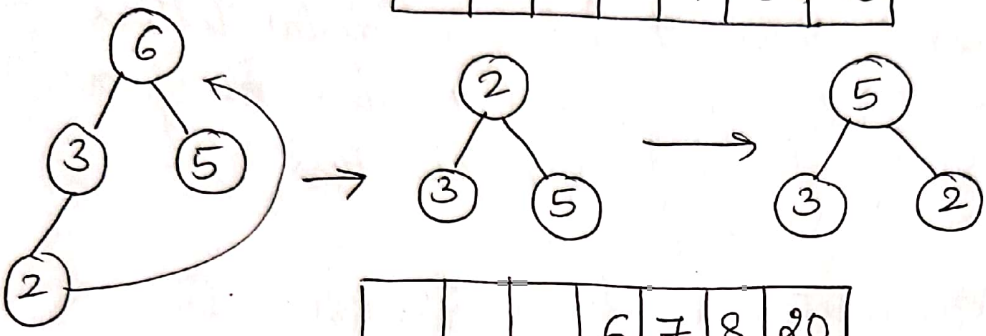


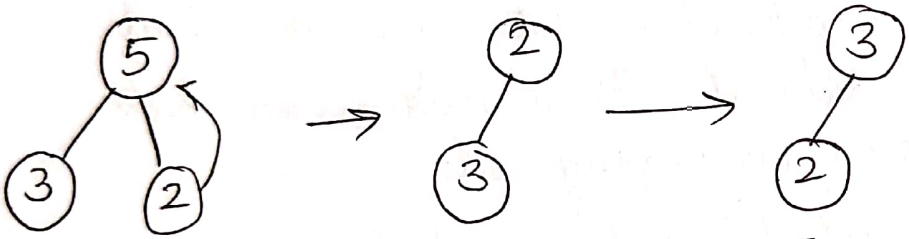


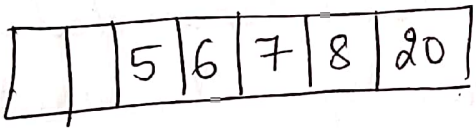


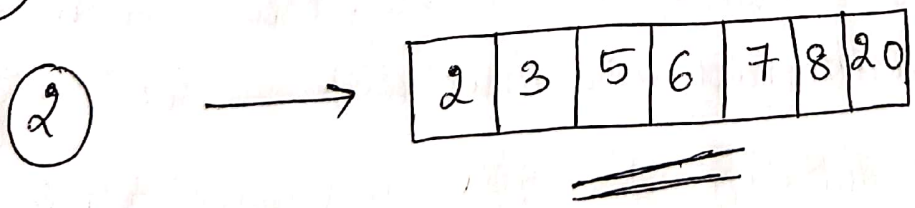












Sets and Disjoint Set Union / Set Representation

A formal definition of set is "a collection of well defined and well distinguished objects". The objects that make up a set are called the members of elements of the set. The standard mathematical symbols used to represent sets are capital letters A, B, C, P, S, X, Y etc. The elements of members of a set are usually denoted by small letters a, b, c, x, y etc.

eg : A is a set of vowels $A = \{a, e, i, o, u\}$

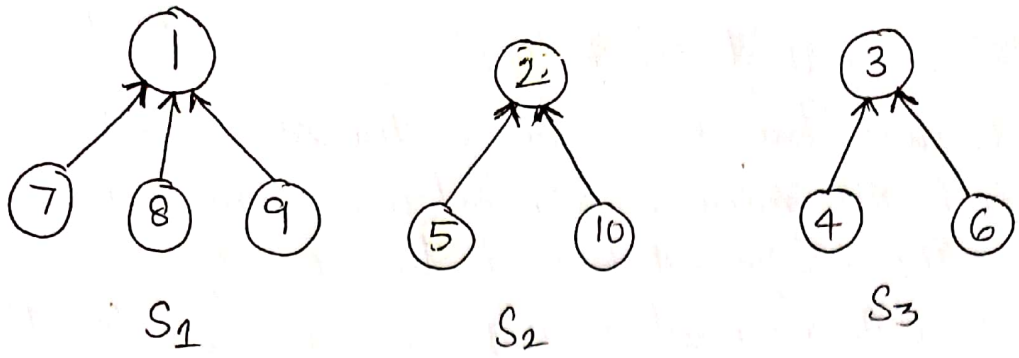
A disjoint set is a kind of data structure that contains partitioned sets. These partitioned sets are separate non-overlapping sets.

Assuming the sets being represented are pairwise disjoint i.e. if S_i and S_j, $i \neq j$, are two sets then there is no element that in in both S_i and S_j.

Example: Consider a set $S = \{1, 2, 3, 4 \ldots 10\}$, having 10 elements i.e. $n = 10$, the elements can be partitioned into three disjoint sets,

$S_1 = \{1, 7, 8, 9\}$ $S_2 = \{2, 5, 10\}$ $S_3 = \{3, 4, 6\}$

Each of the set can be represented as a tree, hence these trees corresponding to the sets are :



S_1 S_2 S_3

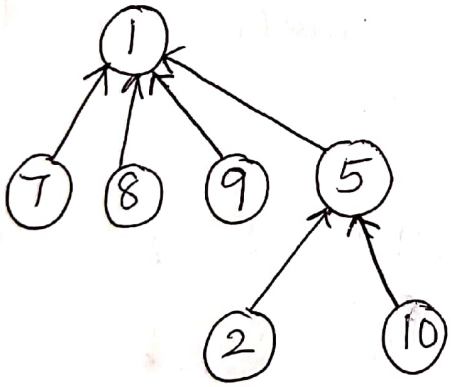
The nodes have been linked from the children to the parent.

The operations performed on these sets are:

① **Union**: If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all elements x such that x is in S_i or S_j.

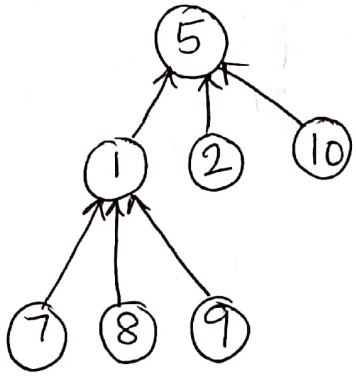
Example: $S_1 = \{1, 7, 8, 9\}$ $S_2 = \{2, 5, 10\}$

$S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$



$S1 \cup S_2$

or

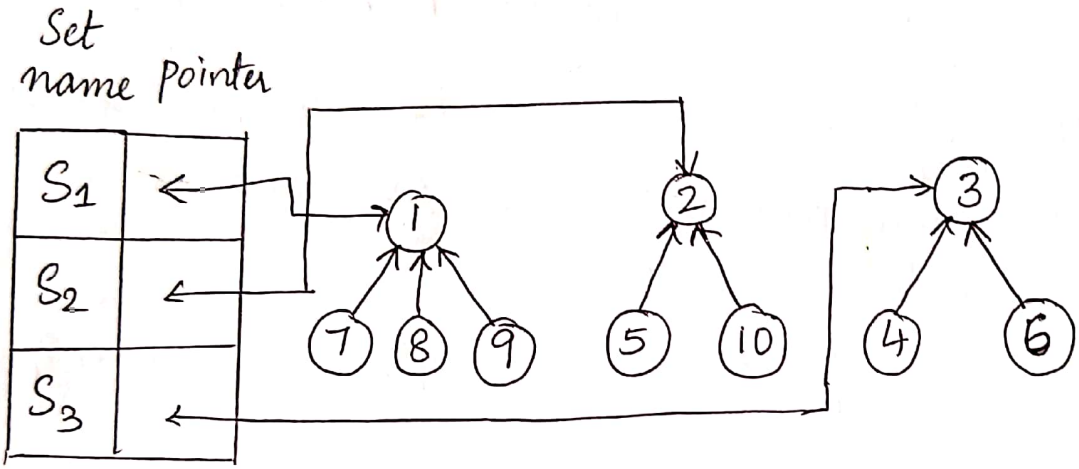


$S_1 \cup S_2$

To obtain the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root.

This can be accomplished easily if, with each set name, a pointer to the root of the tree

representing that set is kept. If, in addition, each root has a pointer to the set name, then to determine which set an element is currently in, parent links to the root of its tree are followed and the pointer is used to the set name. The data representation for S_1, S_2 and S_3 may take the form below:



(2) **Find**: Find operation i.e. find(i) is used to find the set containing i.

Example: $S_1 = \{1, 7, 8, 9\}$ $S_2 = \{2, 5, 10\}$

find(8) = $S_1 = 1$ find(10) = $S_2 = 2$

Array representation of S_1, S_2 and S_3

The tree nodes can be represented using an array $p[1:n]$, where n is the maximum number of elements. The ith element of this

array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree. The root nodes have a parent of -1.

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
P	-1	-1	-1	3	2	3	1	1	1	5

'Find (i)' can be implemented by following the indices, starting at i until a node with parent value -1 is reached.
Find (6) starts at 6 and then moves to 6's parent, 3. Since p[3] is negative, the root has been reached.
'Union (i, j)' can be implemented by passing two trees with roots i and j. Adopting the convention that the first tree becomes a subtree of the second, the statement p[i] := j; accomplishes the union.

Algorithms for union and find

```
1 Algorithm SimpleUnion (i, j)
2 {
3     p[i] := j;
4 }
```

```
1 Algorithm SimpleFind (i)
2 {
3   while (p[i] ≥ 0) do i := p[i];
4   return i;
5 }
```

Processing the following sequence of union-find operations:

Union (1,2), Union (2,3), Union (3,4), ..., Union (n-1, n)

Find (1), Find (2), ..., Find (n)

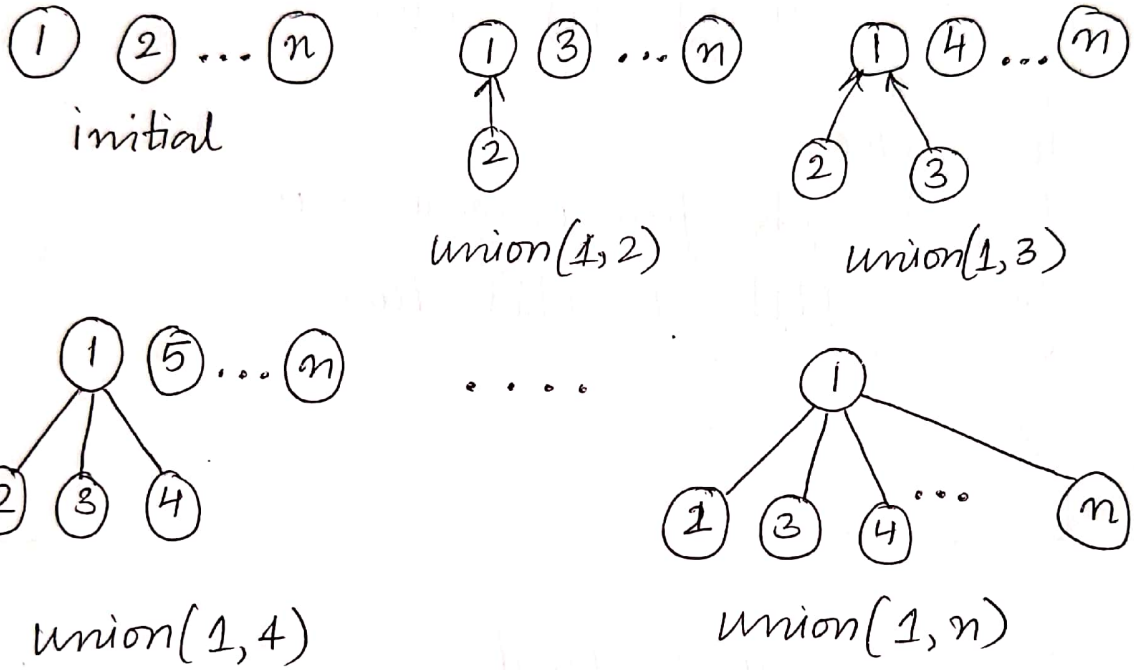
This sequence results in the degenerate tree:

The unions can be processed in time $O(n)$.
The time needed to process n finds is $O(n^2)$.
The performance of 'union' and 'find' algorithms can be improved by avoiding the creation of degenerate trees. To accomplish this, a weighting rule for Union (i, j) is used.

→ Weighting rule for Union(i,j) :

If the number of nodes in the tree with root 'i' is less than the number in the tree with root 'j', then make 'j' the parent of 'i'; otherwise make 'i' the parent of 'j'.

The weighting rule when performed on the sequence of set unions gives the trees below :



To implement the weighting rule, how many nodes are there in every tree must be known.

A 'count' field in the root of every tree is maintained. If 'i' is a root node, then count[i] equals the number of nodes in that tree. Since all nodes other than the roots of trees have a positive number in the 'p' field, the count in the 'p' field of the roots can be a negative number.

Union Algorithm with weighting rule

```
1 Algorithm WeightedUnion (i,j)
2 // Union sets with roots i and j, i≠j, using the
3 // weighting rule. p[i] = -count[i] and p[j] = -count[j]
4 {
5     temp := p[i] + p[j];
6     if (p[i] > p[j]) then
7     { // i has fewer nodes.
8         p[i] := j; p[j] := temp;
9     }
10    else
11    { // j has fewer or equal nodes.
12        p[j] := i; p[i] := temp;
13    }
14 }
```

→ **Collapsing rule for Find (i):**

If 'j' is a node on the path from 'i' to its root and p[i] ≠ root[i], then set p[j] to root[i].

Find algorithm with collapsing rule

```
1 Algorithm CollapsingFind (i)
2 // Find the root of the tree containing element i.
3 // Use the collapsing rule to collapse all nodes from i to root.
4 {
5     r = i;
```

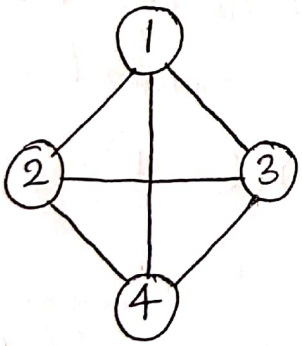
```
6  while (p[r]>0) do r:=p[r]; // Find the root
7  while (i≠r) do  // Collapse nodes from i to root r.
8  {
9      s:=p[i]; p[i]:=r; i:=s;
10 }
11 return r;
12 }
```

Graphs

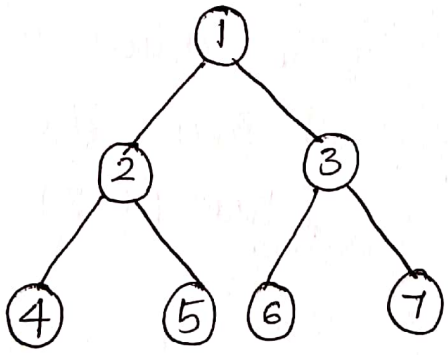
A graph G consists of two sets V and E. The set V is a finite, non empty set of vertices. The set E is a set of pairs of vertices, these pairs are called edges. A graph is represented as $G = (V, E)$.

In an "undirected graph" the pair of vertices representing any edge is unordered. Thus, the pairs (u, v) and (v, u) represent the same edge.

In a "directed graph" each edge is represented by a directed pair $\langle u, v \rangle$; u is the tail and v is the head of the edge. Thus, the pairs $\langle v, u \rangle$ and $\langle u, v \rangle$ represent two different edges.



G1



G2

G3

The graph G1 and G2 are undirected; G3 is directed. The set representations of these graphs are:

$V(G1) = \{1, 2, 3, 4\}$

$E(G1) = \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\}$

$V(G2) = \{1, 2, 3, 4, 5, 6, 7\}$

$E(G2) = \{(1,2), (1,3), (2,4), (2,5), (3,6), (3,7)\}$

$V(G3) = \{1, 2, 3\}$

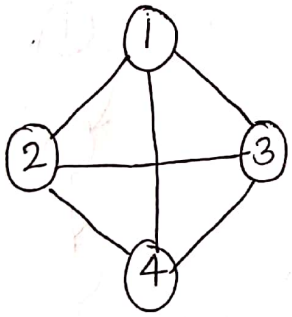
$E(G3) = \{\langle 1,2\rangle, \langle 2,1\rangle, \langle 2,3\rangle\}$

→ The following restrictions are imposed on graphs:

- A graph may not have an edge from a vertex 'v' back to itself.
- A graph may not have multiple occurrences of the same edge.

→ If (u, v) is an edge in $E(G)$, the vertices u and v are adjacent and edge (u, v) is incident on vertices u and v.

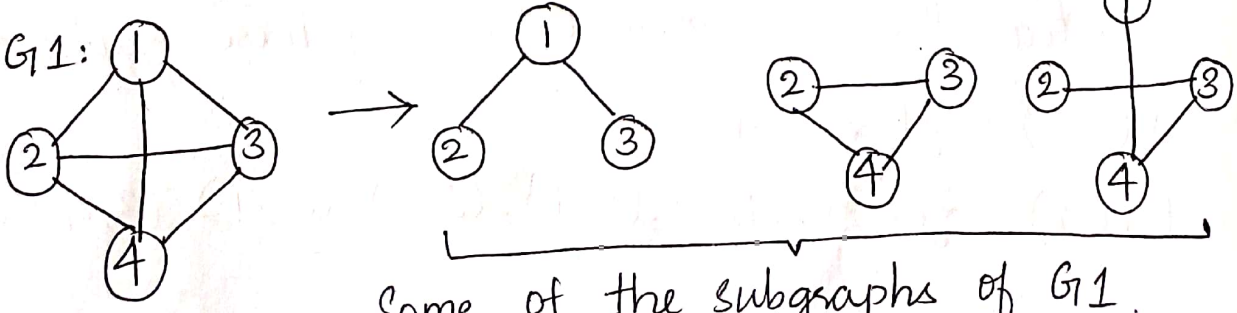
G1



The vertices adjacent to vertex 2 are 1, 3, 4.. The edges incident on vertex 2 are (1, 2), (2, 3), (2, 4).

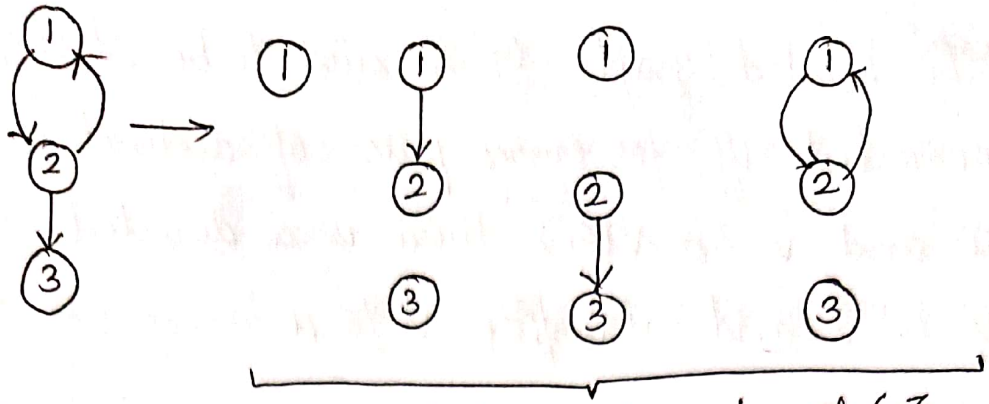
→ A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.

G1:



Some of the subgraphs of G1.

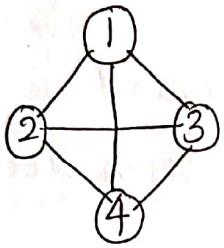
G3:



Some of the subgraphs of G3

→ A cycle is a simple path in which the first and last vertices are the same.

G1:

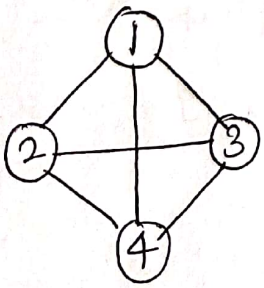


The path 1, 2, 3, 1 is a cycle in G1.

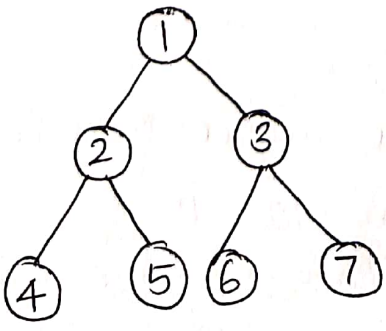
G3:

The path 1, 2, 1 is a cycle in G3.

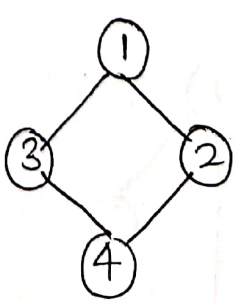
→ In an undirected graph G, two vertices u and v are said to be connected iff there is a path in G from u to v.



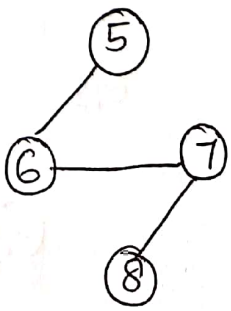
G1



G2



G4



→ A directed graph G is said to be strongly connected iff for every pair of distinct vertices u and v in V(G), there is a directed path from u to v and also from v to u.

G3:

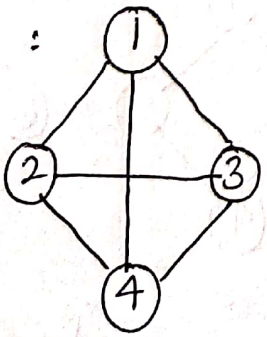
The graph G3 is not strongly connected, as there is no path from vertex 3 to 2.

→ A strongly connected component is a maximal subgraph that is strongly connected.

G3 has two strongly connected components

→ The degree of a vertex is the number of edges incident to that vertex.

G1 :



The degree of a vertex 1 in G1 is 3.

→ If G is a directed graph, the 'in-degree' of a vertex v is the number of edges for which v is the head. The 'out-degree' is defined to be the number of edges for which the v is the tail.

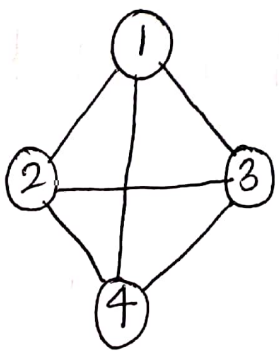
G3 :

Vertex 2 of G3 has in-degree 1, out-degree 2, and degree 3.

Graph Representation : The three most commonly used :
- adjacency matrices
- adjacency lists
- adjacency multilists

Adjacency Matrix – Let G = (V, E) be a graph with 'n' vertices, $n \geq 1$. The adjacency matrix of G is a two dimensional $n \times n$ array, say a, with the property that $a[i, j] = 1$ iff the edge (i, j) is in E(G). The element $a[i, j] = 0$ if there is no such edge in G.

G1 :



→

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

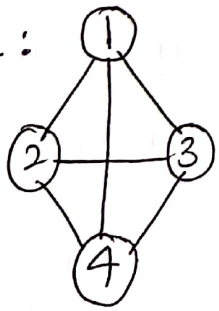
G3:

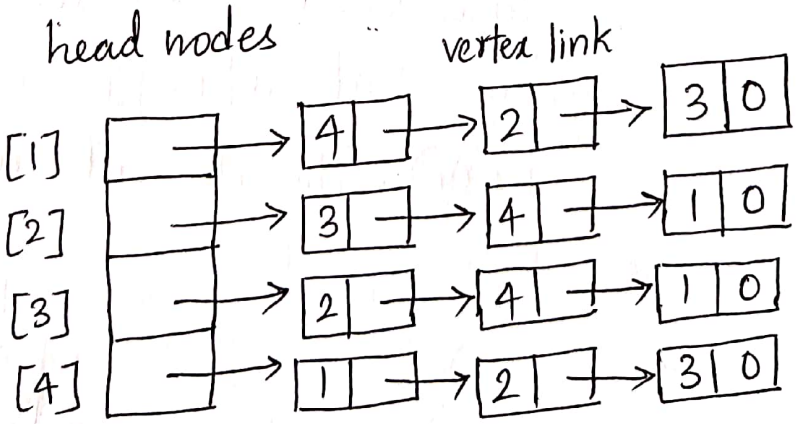
$$\rightarrow \quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

The adjacency matrix for an undirected graph is symmetric, as the edge (i,j) is in $E(G)$ iff the edge (j,i) is also in $E(G)$. The adjacency matrix for a directed graph may not be symmetric.

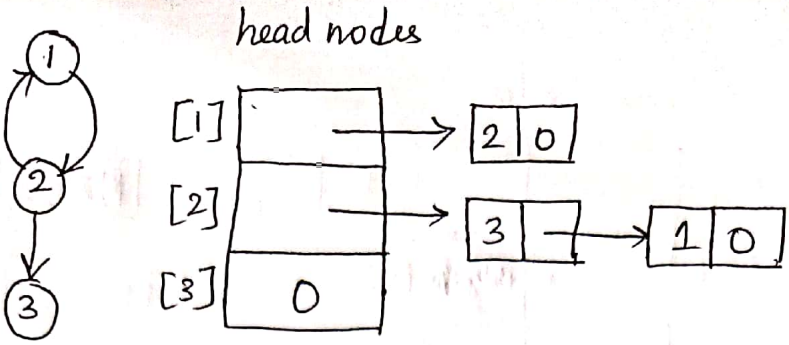
Adjacency Lists – In this representation of graphs, the n rows of the adjacency matrix, are represented as n linked lists. There is one list for each vertex in G. The nodes in list i represent the vertices that are adjacent from vertex i. Each node has atleast two fields : vertex and link. The vertex field contains the indices of the vertices adjacent to vertex i.

G1:



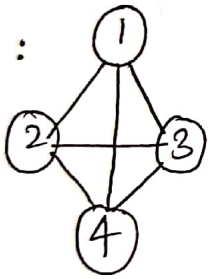


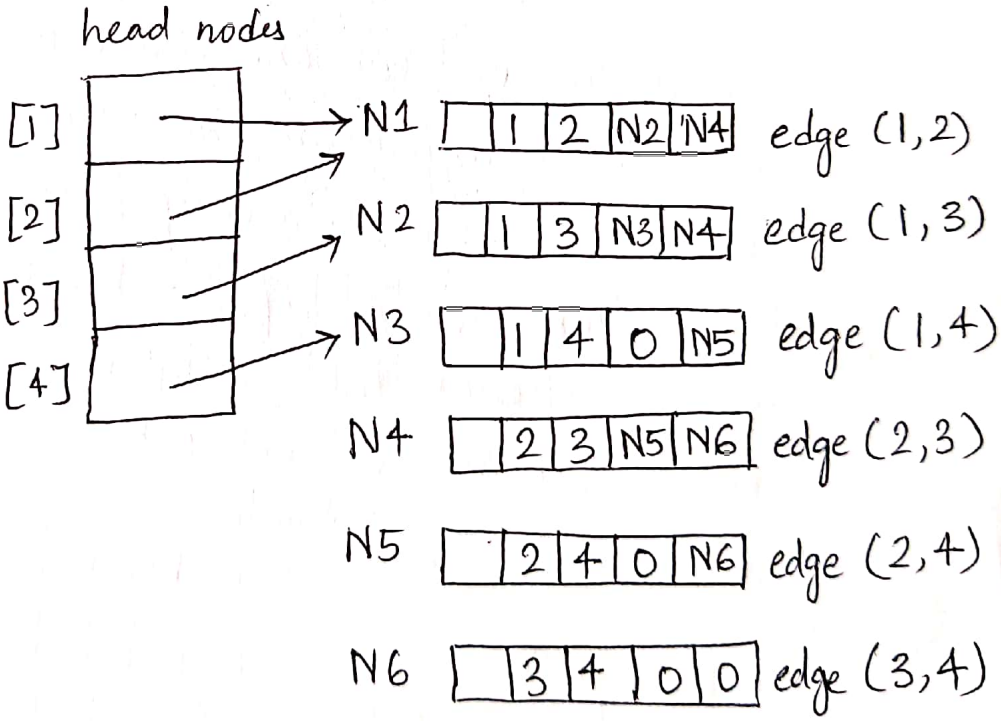
G3 :



Adjacency Multilists – In the adjacency list representation of an undirected graph, each edge (u, v) is represented by two entries, one on the list for u and the other on the list for v. If the adjacency lists are maintained as multilists, for each node there is exactly one node, but this node is in two lists. The

G1 :





The lists are

Vertex 1 : N1 → N2 → N3

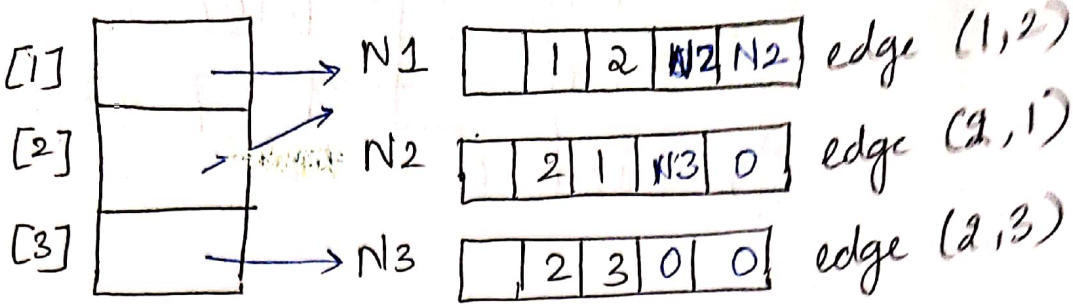
Vertex 2 : N1 → N4 → N5

Vertex 3 : N2 → N4 → N6

Vertex 4 : N3 → N5 → N6

G3:

head nodes



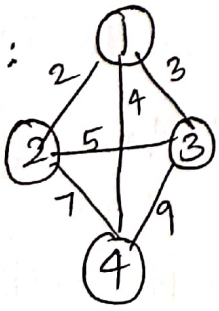
These lists are vertex 1 : N1 → N2

vertex 2 : N1 → N2 → N3

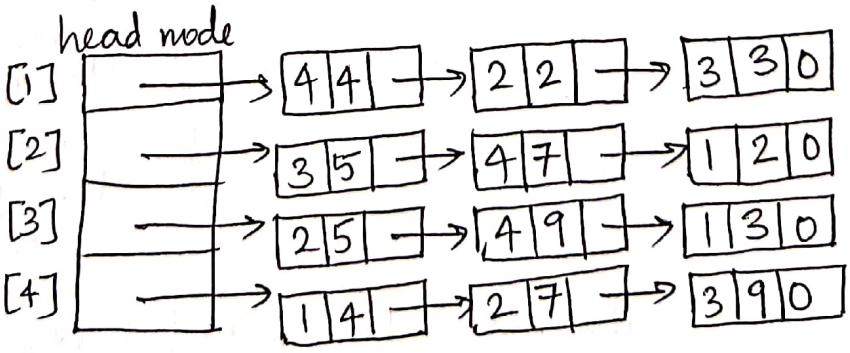
vertex 3 : N3

→ Weighted Edges - The edges of a graph have weights assigned to them. These weights represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. When adjacency lists are used, the weight information can be kept in the list nodes by including an additional field, 'weight'. A graph with weighted edges is called a 'network'.

G1:



head node



G3:

head node