

Data Structure Using C

Contributed By:
Mamata Garanayak

Disclaimer

This document may not contain any originality and should not be used as a substitute for prescribed textbook. The information present here is uploaded by contributors to help other users. Various sources may have been used/referred while preparing this material. Further, this document is not intended to be used for commercial purpose and neither the contributor(s) nor LectureNotes.in is accountable for any issues, legal or otherwise, arising out of use of this document. The contributor makes no representation or warranties with respect to the accuracy or completeness of the contents of this document and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. By proceeding further, you agree to LectureNotes.in Terms of Use. Sharing of this document is forbidden under LectureNotes Term of use. Sharing it will be meant as violation of LectureNotes Terms of Use.

This document was downloaded by: gaurav rawat of Ajay Kumar Garg Engineering College with registered phone number 7906721987 and email grawat70@gmail.com on 08th Sep, 2018. and it may not be used by anyone else.

At LectureNotes.in you can also find

1. Previous Year Questions for BPUT
2. Notes from best faculties for all subjects
3. Solution to Previous year Questions



Data Structure Using C

Topic:
Algorithm

Contributed By:
Mamata Garanayak

Lesson Number : 1

Algorithm :

- The word algorithm comes from the name of a persian author, Abujafor Mohamod ibn Musa al Khwarzimi.
- This word has taken a special significance in computer science as it is used as a tool for solving a well specified computational problem.
- This is what makes algorithm different from words such as process techniques or methods.

LectureNotes.in

Definition :

" An algorithm is a finite set of instruction that if followed, accomplishes a particular task."

or

" An algorithm is any well defined computational procedure that takes some value or set of values as input and produces some value or set of values as output."

Characteristics Of an Algorithm :

An algorithm must have the following characteristics.

1. Input
2. Output
3. Definiteness
4. Finiteness
5. Effectiveness

LectureNotes.in

1. Input : There are some (or may be empty) I/P data which are externally supplied to algorithm to perform a task. Algorithm may have zero I/P.

Example : 1 Suppose we want to add 2 no. then the algorithm is

Step 1 : start

Step 2 : Read 2 no. a and b .

Step 3 : $s = a + b$.

Step 4 : Addition result is 's' i.e. display s.

Step 5 : Stop.

Explanation: Here 2 inputs are being supplied to algorithm to get addition result.

Example : 2 If our task is to print a name, then the algorithm is

Step 1 : Start

Step 2 : display "ALGORITHM"

Step 3 : Stop

Explanation: Here we do not require any I/P that is to be supplied for displaying "ALGORITHM". So the algorithm over here do not require any I/P, but task (displaying "ALGORITHM") is performed.

2. Output: There will be at least one O/P as a result. So no. of O/P for an algorithm may be '1' or more than one. Consider the example : 2. Here we are getting an O/P. If an algorithm is not supplying any O/P or result then what is the necessity of writing an algorithm.

Example : 3

Step 1 : Start

Step 2 : for($i=1$; $i < 5$; $i++$)

display "ALGORITHM"

Step 3 : Stop

Explanation: Here no. of outputs are 4 i.e. ALGORITHM will be printed for '4' times

3. Definiteness: Each instructions / steps of the algorithm must be clear and unambiguous.

Example : 4

Step 1 : Start

Step 2 : Read a and b

Step 3 : $C = a +$

Step 4 : Stop

Explanation: Here step 3 is ambiguous.

4. Finiteness: The algorithm should terminate after a finite no. of steps. e.g. if we will consider the example 3, the algorithm is going to terminate after '4' no. of iterative steps.

Example-5 1. while (1)

2. printf ("ALGORITHM")

Explanation: Here the algorithm will not come to an end, because of infinite loop. So it violate the definiteness criteria.

LectureNotes.in

5. Effectiveness: The statements/instructions of an algorithm should be simple that, the output can be found out by pen and paper.

Example-6 Step 1 : Start

Step 2 : Read 2 numbers a and b.

Step 3 : $S = a + b$

Step 4 : Addition result is 'S' i.e. display S.

Step 5 : Stop

The above algorithm satisfying the effectiveness criteria.

Example-7

Step 1 : Step start

Step 2 : for ($i=1$; $i \leq 5$; $i++$)

display "ALGORITHM".

Step 3 : Stop

The above algorithm satisfying the effectiveness criteria i.e. we can carried out this algorithm by using pen and paper.

Space Complexity:

- "The space Complexity of an algorithm is the amount of memory it needs to run to completion."
- Example :
 1. Algorithm abc (a, b, c)
 2. {
 3. return $a+b+b*c + (a+b-c) / (a+b) + 4 \cdot 0 ;$
 4. }
- This problem consists of 3 variables a, b, c. So fixed part is 3 and variable part is 0.
So space complexity is $3+0 = 3$!
- Space complexity consists of two parts.

- 1> A fixed part is independent of the characteristics (e.g. numbers) of input and outputs. This part typically includes the instruction space, space for variables, space for constants etc.
- 2> A variable part consists of the space needed by component variables whose size is depended on the particular problem. The space required ($S(P)$) most of any algorithm P may therefore be written as ;

$$S(P) = C + SP$$

where $S(P)$ = space required | space complexity
 C = fixed part
 SP = variable part .

Example :

1. Algorithm sum (a, n)
2. {
3. $S = 0$.
4. for $i=1$ to n do
5. $S = S + a[i]$
6. return S
7. }

The space needed by s, i, a is 3, and variable part means that loop needs n memory.

$$\text{So } S_{\text{sum}}(n) = n + 3 //$$

Time Complexity :

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

The time $T(P)$ taken by a program P is the sum of the compile time and execution time.

Example :

Statements	Total steps
1. Algorithm sum(a, n)	0
2. {	0
3. $s = 0$.	1
4. for $i = 1$ to n do	$n+1$
5. $s = s + a[i]$	n
6. return s .	1
7. }	0
	$\frac{2n+3}{}$

So the time complexity is $2n+3 //$

Example :

Statements	Total steps
1. Algorithm add (a, b, c, m, n)	0
2. {	0
3. for $i = 1$ to m do	$m+1$
4. for $j = 1$ to n do	$mn+m$
5. $c[i, j] = a[i, j] + b[i, j]$	mn
6. }	0
	$\frac{2mn+2m+1}{}$

So the time complexity is $2mn + 2m + 1$

Growth Of Function :

Different running time can be obtained for the algorithm. The running time of an algorithm depends upon various characteristics. And slight variation in the characteristics varies the running time. The algorithm efficiency and performance in comparison to alternate algorithm is best described by the order of growth of the running time of an algorithm.

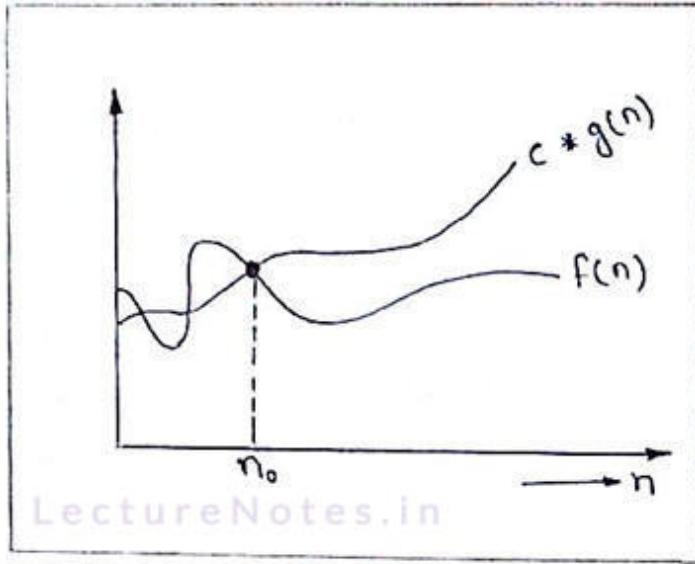
Asymptotic Notations:

- Asymptotic notation describes the algorithm efficiency and performance in a meaningful way. It describes the behaviour of time or space complexity for large instance characteristics.
- The order of growth of the running time of an algorithm gives a simple character of the algorithm's efficiency and also allows us to compare relative performance of alternative algorithm. We call it as growth function as we ignore the very small constant.
- The asymptotic running time of an algorithm is defined in terms of functions.
- The asymptotic notations of an algorithm is classified into 5 types.
 1. Big oh notation (O)
 2. Big Omega notation (Ω)
 3. Big theta notation (Θ)
 4. Little oh notation (o)
 5. Little omega notation (ω)

1. Big oh notation (O): (Asymptotic Upper bound)

" The function $f(n) = O(g(n))$ iff there exist a positive constant C and n_0 such that

$$f(n) \leq C * g(n) \text{ for all } n, n \geq n_0 .$$



[Big Oh notation ($f(n) \leq c * g(n)$)]

1. Constant Function :

Suppose given constant functions :

- $f(n) = 16$
- $f(n) = 27$
- $f(n) = 1627$

Then for satisfying the big oh condition, the above function can be expressed as :

- $f(n) \leq 16 * 1$ where $c = 16$ and $n_0 = 0$
- $f(n) \leq 27 * 1$ where $c = 27$ and $n_0 = 0$
- $f(n) \leq 1627 * 1$ where $c = 1627$ and $n_0 = 0$.

Thus, for above functions we can specify the big oh notation as $O(1)$. So $f(n) = O(1)$,

2. Linear Function :

Consider the following linear functions .

- $f(n) = 3n + 5$
- $f(n) = 2n + 3$
- $f(n) = 7n + 5$

→ for $f(n) = 3n + 5$, when n is atleast 5, $n \geq 5$

$$3n + 5 \leq 4n \text{ where } c = 4 \text{ and } n_0 = 5$$

$$\text{So } f(n) = O(n) //$$

$$\rightarrow f(n) = 2n + 3$$

for $f(n) = 2n + 3$, when 'n' is at least 3, $n \geq 3$

$2n + 3 \leq 3n$ where $c = 3$ and $n_0 = 3$

so $f(n) = O(n)$ //

$$\rightarrow f(n) = 7n + 5$$

for $f(n) = 7n + 5$, when 'n' is at least 5, $n \geq 5$.

$7n + 5 \leq 8n$, where $c = 8$ and $n_0 = 5$.

so $f(n) = O(n)$ //

3. Quadratic Function:

Consider the following quadratic Functions :

- $f(n) = 27n^2 + 16n$

- $f(n) = 27n^2 + 16$

- $f(n) = 10n^2 + 7$

- $f(n) = 27n^2 + 16n + 25$

$$\rightarrow f(n) = 27n^2 + 16n$$

$27n^2 + 16n \leq 28n^2$ where $c = 28$ and $n_0 = 16$

so $f(n) = O(n^2)$ //

$$\rightarrow f(n) = 27n^2 + 16 \text{ for } n \geq 16$$

for $n \leq n^2$

$27n^2 + n \leq 27n^2 + n^2 \leq 28n^2$ where $c = 28$ and $n_0 = 1$

so $f(n) = O(n^2)$ //

$$\rightarrow f(n) = 10n^2 + 7, \text{ for } n \geq 7$$

$10n^2 + 7 \leq 10n^2 + n$

for $n \leq n^2$

$10n^2 + n \leq 10n^2 + n^2 \leq 11n^2$ where $c = 11$ and $n_0 = 1$.

so $f(n) = O(n^2)$ //

$$\rightarrow f(n) = 27n^2 + 16n + 25 \text{ for } n \geq 25$$

$27n^2 + 16n + 25$

$\leq 27n^2 + 16n + n$

$\leq 27n^2 + 17n$

Now for $n \geq 17$, $n^2 \geq 17$ thus

$$27n^2 + 17n \leq 27n^2 + n^2 \\ \leq 28n^2 \text{ where } c=28, n_0=17.$$

so $f(n) = O(n^2)$ //

4. Cubic Function :

consider the following cubic functions:

- $f(n) = 2n^3 + n^2 + 2n$
- $f(n) = 4n^3 + 2n + 3$
- $f(n) = 3n^3 + 4n$
- $f(n) = 5n^3 + n^2 + 3n + 2$

$$\rightarrow f(n) = 2n^3 + n^2 + 2n$$

for $2n \leq n^2$

$$2n^3 + n^2 + 2n \leq 2n^3 + n^2 + n^2 \\ \leq 2n^3 + 2n^2$$

Now for $2n^2 \leq n^3$

$$2n^3 + 2n^2 \leq 2n^3 + n^3 \\ \leq 3n^3 \text{ where } c=3 \text{ and } n_0=2$$

so $f(n) = O(n^3)$ //

$$\rightarrow f(n) = 4n^3 + 2n + 3$$

for $3 \leq n$

$$4n^3 + 2n + 3 \leq 4n^3 + 2n + n \\ \leq 4n^3 + 3n$$

Now for $3n \leq n^3$

$$4n^3 + 3n \leq 4n^3 + n^3 \\ \leq 5n^3 \text{ where } c=5 \text{ and } n_0=3.$$

so $f(n) = O(n^3)$ //

$$\rightarrow f(n) = 3n^3 + 4n$$

for $4n \leq n^3$

$$3n^3 + 4n \leq 3n^3 + n^3$$

$\leq 4n^3$ where $C=4$ and $n_0=4$

so, $f(n) = O(n^3)$,

$$\rightarrow f(n) = 5n^3 + n^2 + 3n + 2$$

for $2 \leq n$

$$5n^3 + n^2 + 3n + 2 \leq 5n^3 + n^3 + 3n + n$$

LectureNotes.in

$$\leq 5n^3 + n^3 + 4n$$

now for $4n \leq n^2$

$$5n^3 + n^2 + 4n \leq 5n^3 + n^2 + n^2$$
$$\leq 5n^3 + 2n^2$$

for $2n^2 \leq n^3$

$$5n^3 + 2n^2 \leq 5n^3 + n^3$$

$\leq 6n^3$ where $C=6$ and $n_0=2$

so $f(n) = O(n^3)$,

5. Exponential Function:

Consider the following exponential functions.

- $f(n) = 2^n + 6n^2 + 3n$

- $f(n) = 4 * 2^n + 3n$

- $f(n) = 5 * 2^n + 3n + 5$

$$\rightarrow f(n) = 2^n + 6n^2 + 3n$$

for $n^2 \geq 3n$

$$2^n + 6n^2 + 3n \leq 2^n + 6n^2 + n^2$$
$$\leq 2^n + 7n^2$$

Now for $n^2 \leq 2^n$ ($n \geq 4$)

$$2^n + 7n^2 \leq 2^n + 7 \cdot 2^n$$

$\leq 8 * 2^n$ where $C=8$ and $n_0=4$.

so $f(n) = O(2^n)$,

$$\rightarrow f(n) = 4 * 2^n + 3n$$

for $2^n > n$ ($n \geq 1$)

$$4 * 2^n + 3n \leq 4 * 2^n + 3 * 2^n$$

$$\leq 7 * 2^n \text{ where } C=7 \text{ and } n_0=1$$

$$\text{so } f(n) = O(2^n),$$

$$\rightarrow f(n) = 5 * 2^n + 3n + 5$$

for $n \geq 5$

$$5 * 2^n + 3n + 5 \leq 5 * 2^n + 3n + n$$
$$\leq 5 * 2^n + 4n$$

now for $n < 2^n$ ($n \geq 1$)

$$5 * 2^n + 4n \leq 5 * 2^n + 4 * 2^n$$

$$\leq 9 * 2^n \text{ where } C=9 \text{ and } n_0=1.$$

$$\text{so } f(n) = O(2^n),$$

Big Oh Ratio Theorem :

Let $f(n)$ and $g(n)$ be such that $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ exists, then a function $f \in O(g)$ if

$$\lim_{n \rightarrow \infty} f(n)/g(n) = C < \infty, \text{ also including the case in}$$

which limit is 0.

Growth Rates :

$$\lim_{n \rightarrow \infty} f(n)/g(n) = \begin{cases} \underline{0} & f(n) \text{ grows slower than } g(n). \\ \underline{\infty} & f(n) \text{ grows faster than } g(n). \\ \underline{\text{otherwise}} & f(n) \text{ and } g(n) \text{ have the same growth rates.} \\ \bullet & \end{cases}$$

Examples :

$$\rightarrow f(n) = 2n + 3$$

$$\lim_{n \rightarrow \infty} \frac{2n+3}{n} = 2, \text{ so } f(n) = O(n),$$

$$\rightarrow f(n) = 27n^2 + 16n + 25$$

$$\lim_{n \rightarrow \infty} \frac{27n^2 + 16n + 25}{n^2} = 27$$

$$\text{so. } f(n) = O(n^2).$$

$$\rightarrow f(n) = 5 * 2^n + 3n + 5$$

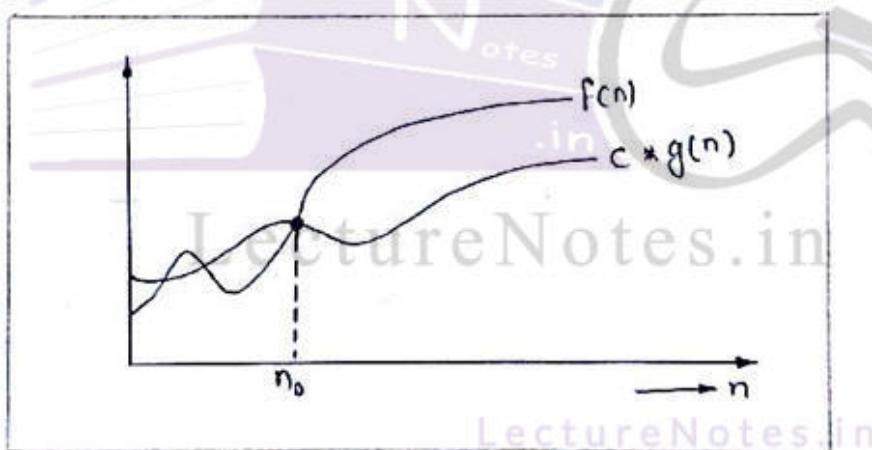
$$\lim_{n \rightarrow \infty} \frac{5 * 2^n + 3n + 5}{2^n} = 5$$

$$\therefore f(n) = O(2^n).$$

2. Big Omega notation (Ω) : (Asymptotically lower bound)

"The function $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that

$$f(n) \geq c * g(n) \text{ for all } n, n \geq n_0.$$



[Omega ($f(n) = \Omega(g(n))$) notation]

1. Constant Function :

Consider the following constant functions :

- $f(n) = 16$
- $f(n) = 27$

\rightarrow For satisfying the big omega condition

$$f(n) \geq 15 * 1, \text{ where } c = 15, n_0 = 0$$

$$\text{so } f(n) = \Omega(1) //$$

$$\rightarrow f(n) = 16 + 11 = 27$$

$f(n) \geq 26 * 1$, where $C = 26$, $n_0 = 0$

$$\text{so } f(n) = \Omega(1) //$$

2. Linear Function :

Consider the following linear functions :

- $f(n) = 3n + 5$

- $f(n) = 2n + 3$

$$\rightarrow f(n) = 3n + 5$$

LectureNotes.in

$$3n + 5 \geq 3n \text{ for all } n, \{C = 3\}$$

$$\text{so } f(n) = \Omega(n) //$$

$$\rightarrow f(n) = 2n + 3$$

LectureNotes.in

$$2n + 3 \geq 2n \text{ for all } n, \{C = 2\}$$

thus $f(n) = \Omega(n) //$

3. Quadratic Function :

Consider the following quadratic functions.

- $f(n) = 27n^2 + 16n$

- $f(n) = 10n^2 + 7$

- $f(n) = 27n^2 + 16n + 25$

$$\rightarrow 27n^2 + 16n = f(n)$$

$$27n^2 + 16n \geq 27n^2 \text{ for all } n \text{ and } \{C = 27\}$$

$$\text{so } f(n) = \Omega(n^2) //$$

$$\rightarrow f(n) = 10n^2 + 7$$

$$10n^2 + 7 \geq 10n^2 \text{ for all } n, \{C = 10\}$$

$$\text{so, } f(n) = \Omega(n^2) //$$

$$\rightarrow f(n) = 27n^2 + 16n + 25$$

$$27n^2 + 16n + 25 \geq 27n^2 \text{ for all } n, \{C = 27\}$$

thus $f(n) = \Omega(n^2) //$

4. Cubic Function :

Consider the following functions :

- $f(n) = 2n^3 + n^2 + 2n$
- $f(n) = 5n^3 + n^2 + 3n + 2$
- $f(n) = 3n^3 + 4n$
- $f(n) = 4n^3 + 2n + 3$.

$$\rightarrow f(n) = 2n^3 + n^2 + 2n$$

$$2n^3 + n^2 + 2n \geq 2n^3 \text{ for all } n, \{c=2\}.$$

$$\text{so } f(n) = \Omega(n^3), //$$

$$\rightarrow f(n) = 5n^3 + n^2 + 3n + 2$$

$$5n^3 + n^2 + 3n + 2 \geq 5n^3 \text{ for all } n, \{c=5\}$$

$$\text{so } f(n) = \Omega(n^3), //$$

$$\rightarrow f(n) = 3n^3 + 4n$$

$$3n^3 + 4n \geq 3n^3 \text{ for all } n, \{c=3\}$$

$$\text{so } f(n) = \Omega(n^3), //$$

$$\rightarrow f(n) = 4n^3 + 2n + 3$$

$$4n^3 + 2n + 3 \geq 4n^3 \text{ for all } n, \{c=4\}$$

$$\text{Thus } f(n) = \Omega(n^3), //$$

5. Exponential Function :

Consider the following exponential functions :

- $f(n) = 2^n + 6n^2 + 3n$
- $f(n) = 5 * 2^n + 3n + 5$
- $f(n) = 3 * 2^n + 4n^2 + 5n + 3$

$$\rightarrow f(n) = 2^n + 6n^2 + 3n$$

$$2^n + 6n^2 + 3n \geq 2^n \text{ for all } n, c=1$$

$$\text{so } f(n) = \Omega(2^n), //$$

$$\rightarrow f(n) = 5 \times 2^n + 3n + 5$$

$$5 \times 2^n + 3n + 5 \geq 5 \times 2^n \text{ for all } n, c = 5$$

$$\text{so } f(n) = \Omega(2^n) //$$

$$\rightarrow f(n) = 3 \times 2^n + 4n^2 + 5n + 3$$

$$3 \times 2^n + 4n^2 + 5n + 3 \geq 3 \times 2^n \text{ for all } n, c = 3$$

$$\text{so } f(n) = \Omega(2^n) //$$

Big Omega (Ω) Ratio Theorem :

let $f(n)$ and $g(n)$ be such that $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ exists, then function $f \in \Omega(g)$ if

$$\boxed{\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0}, \text{ including the case in which the}$$

limit is ∞ .

Example :

$$\rightarrow f(n) = 2n + 3$$

$$\text{now } \lim_{n \rightarrow \infty} \frac{1}{\frac{2n+3}{n}} = \frac{1}{2} > 0 \text{ i.e. } \lim_{n \rightarrow \infty} \frac{n}{2n+3} = \frac{1}{2}$$

$$\text{so } f(n) = \Omega(n) //$$

$$\rightarrow f(n) = 27n^2 + 16n + 5$$

$$\text{now } \lim_{n \rightarrow \infty} \frac{1}{\frac{27n^2 + 16n + 5}{n^2}} = \frac{1}{27} > 0 \text{ i.e. } \lim_{n \rightarrow \infty} \frac{n^2}{27n^2 + 16n + 5} = \frac{1}{27}$$

$$\text{so } f(n) = \Omega(n^2) //$$

$$\rightarrow f(n) = 3n^3 + 4n$$

$$\text{now } \lim_{n \rightarrow \infty} \frac{1}{\frac{3n^3 + 4n}{n^3}} = \frac{1}{3} \text{ i.e. } \lim_{n \rightarrow \infty} \frac{n^3}{3n^3 + 4n} = \frac{1}{3}$$

$$\text{so } f(n) = \Omega(n^3) //$$

$$\rightarrow f(n) = 5 \times 2^n + 3n + 5$$

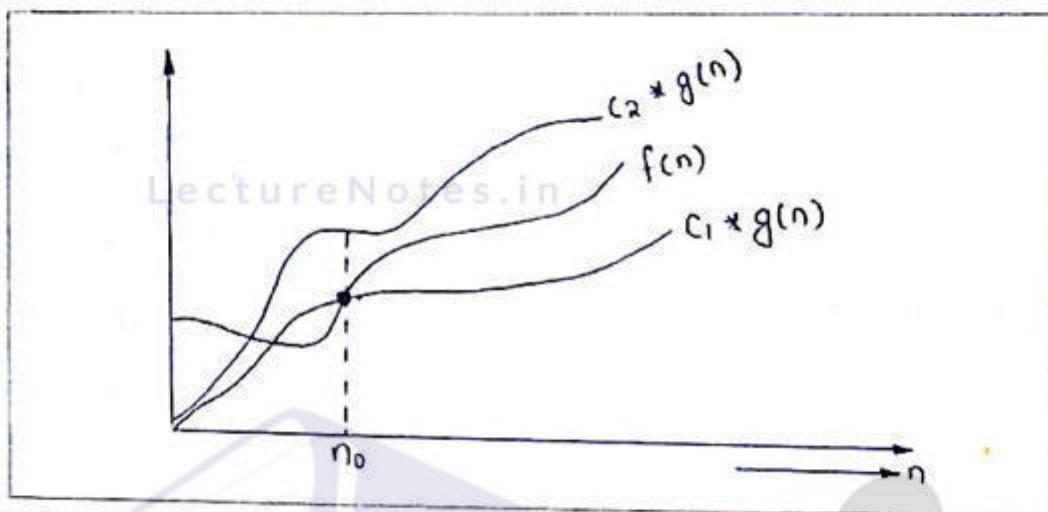
$$\text{now } \lim_{n \rightarrow \infty} \frac{1}{\frac{5 \times 2^n + 3n + 5}{2^n}} = \frac{1}{5} \text{ i.e. } \lim_{n \rightarrow \infty} \frac{2^n}{5 \times 2^n + 3n + 5} = \frac{1}{5}$$

$$\text{so } f(n) = \Omega(2^n) //$$

3. Big Theta notation (Θ): (Asymptotically tight bound)

" The function $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n, n \geq n_0.$$



[Big Theta notation ($c_1 * g(n) \leq f(n) \leq c_2 * g(n)$)]

1. Constant Function:

Consider the following Constant functions.

$$\bullet f(n) = 16$$

$$\bullet f(n) = 27$$

$$\rightarrow f(n) = 16$$

$15 * 1 \leq 16 \leq 16 * 1$ where $c_1 = 15$, $c_2 = 16$ and $n_0 = 0$.

$$\text{so } f(n) = \Theta(1) //$$

$$\rightarrow f(n) = 27$$

$26 * 1 \leq 27 \leq 27 * 1$ where $c_1 = 26$, $c_2 = 27$ and $n_0 = 0$

$$\text{thus } f(n) = \Theta(1) //$$

2. Linear Function:

Consider the following linear functions.

$$\bullet f(n) = 3n + 5$$

$$\bullet f(n) = 23n + 5$$

$$\rightarrow f(n) = 3n + 5$$

$3n < 3n + 5$ for all n , $c_1 = 3$

also, $3n + 5 \leq 4n$ for $n \geq 5$, $c_2 = 4$, $n_0 = 5$

thus $3n < 3n + 5 \leq 4n$ where $c_1 = 3$, $c_2 = 4$, $n_0 = 5$

so $f(n) = \Theta(n)$ //

$$\rightarrow f(n) = 23n + 5$$

$23n < 23n + 5$ for all n , $c_1 = 23$

also $23n + 5 \leq 24n$ for all $n \geq 5$, $c_2 = 24$, $n_0 = 5$

so $23n < 23n + 5 \leq 24n$ where $c_1 = 23$, $c_2 = 24$, $n_0 = 5$

so $f(n) = \Theta(n)$ //

3. Quadratic Function :

Consider the following quadratic functions :

$$\bullet f(n) = 27n^2 + 16n$$

$$\bullet f(n) = 10n^2 + 7$$

$$\bullet f(n) = 27n^2 + 16n + 25$$

$$\rightarrow f(n) = 27n^2 + 16n$$

$27n^2 < 27n^2 + 16n \leq 28n^2$ for all $n \geq n_0 = 16$, $c_2 = 28$, $c_1 = 27$

so $f(n) = \Theta(n^2)$ //

$$\rightarrow f(n) = 10n^2 + 7$$

$10n^2 < 10n^2 + 7 \leq 11n^2$ for $c_1 = 10$, $c_2 = 11$, $n_0 = 7$

so $f(n) = \Theta(n^2)$ //

$$\rightarrow f(n) = 27n^2 + 16n + 25$$

$27n^2 < 27n^2 + 16n + 25 \leq 28n^2$ for $c_1 = 27$, $c_2 = 28$, $n_0 = 17$

so $f(n) = \Theta(n^2)$ //

4. Cubic Function:

Consider the following cubic functions:

$$\bullet f(n) = 2n^3 + n^2 + 2n$$

$$\bullet f(n) = 3n^3 + 4n$$

$$\bullet f(n) = 5n^3 + n^2 + 3n + 2$$

$$\rightarrow f(n) = 2n^3 + n^2 + 2n$$

$$2n^3 < 2n^3 + n^2 + 2n \leq 3n^3 \text{ for } c_1=2, c_2=3, n_0=2$$

$$\text{so } f(n) = \Theta(n^3), //$$

$$\rightarrow f(n) = 3n^3 + 4n$$

$$3n^3 < 3n^3 + 4n \leq 4n^3 \text{ for } c_1=3, c_2=4, n_0=4$$

$$\text{so } f(n) = \Theta(n^3), //$$

$$\rightarrow f(n) = 5n^3 + n^2 + 3n + 2$$

$$5n^3 < 5n^3 + n^2 + 3n + 2 \leq 6n^3 \text{ for } c_1=5, c_2=6, n_0=2$$

$$\text{thus } f(n) = \Theta(n^3), //$$

5. Exponential Function:

Consider the following exponential functions:

$$\bullet f(n) = 2^n + 6n^2 + 3n$$

$$\bullet f(n) = 4 * 2^n + 3n$$

$$\rightarrow f(n) = 2^n + 6n^2 + 3n$$

$$2^n < 2^n + 6n^2 + 3n \leq 8 * 2^n \text{ for } c_1=1, c_2=8, n_0=4$$

$$\text{so } f(n) = \Theta(2^n), //$$

$$\rightarrow f(n) = 4 * 2^n + 3n$$

$$4 * 2^n < 4 * 2^n + 3n \leq 7 * 2^n \text{ for } c_1=4, c_2=7, n_0=1$$

$$\text{so } f(n) = \Theta(2^n), //$$

Big Theta (Θ) Ratio Theorem :

Let $f(n)$ and $g(n)$ be such that $\lim_{n \rightarrow \infty} f(n)/g(n)$ exists, then function $f \in \Theta(g)$ if

$$\boxed{\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c} \quad \text{For some constant } c \text{ such that}$$

$$0 < c < \infty.$$

Example :

$$\rightarrow f(n) = 2n + 3$$

$$\lim_{n \rightarrow \infty} \frac{2n+3}{n} = 2$$

$$\text{also } \lim_{n \rightarrow \infty} \frac{n}{2n+3} = \frac{1}{2}$$

$$2 > \frac{1}{2}$$

$$\text{thus } f(n) = \Theta(n) //$$

$$\rightarrow f(n) = 27n^2 + 16n + 25$$

$$\lim_{n \rightarrow \infty} \frac{27n^2 + 16n + 25}{n^2} = 27$$

$$\text{also } \lim_{n \rightarrow \infty} \frac{n^2}{27n^2 + 16n + 25} = \frac{1}{27}$$

$$27 > \frac{1}{27}, \text{ so } f(n) = \Theta(n^2) //$$

$$\rightarrow f(n) = 5 \cdot 2^n + 3n + 5$$

$$\lim_{n \rightarrow \infty} \frac{5 \cdot 2^n + 3n + 5}{2^n} = 5$$

$$\text{also } \lim_{n \rightarrow \infty} \frac{2^n}{5 \cdot 2^n + 3n + 5} = \frac{1}{5}$$

$$\frac{1}{5} > 5 > \frac{1}{5}$$

$$\text{so } f(n) = \Theta(2^n) //$$

4. Little oh (o) Notation :

" The function $f(n) = \text{o}(g(n))$ iff there exist a positive constant $c > 0$ and $n_0 > 0$ such that

$$f(n) < c * g(n) \text{ for all } n, n \geq n_0.$$

Example :

$$\rightarrow f(n) = 3n + 5 = \text{o}(n^2), \text{ as } 3n + 5 = O(n^2).$$

$$\rightarrow f(n) = 27n^2 + 16n = \text{o}(n^3), \text{ as } 27n^2 + 16n = O(n^3).$$

$$\rightarrow f(n) = 10n^2 + 7 = \text{o}(n^3), \text{ as } 10n^2 + 7 = O(n^3)$$

$$\rightarrow f(n) = 4n^3 + 2n + 3 = \text{o}(n^4), \text{ as } 4n^3 + 2n + 3 = O(n^4)$$

Little oh (o) Ratio Theorem :

Considering 'g' be a set of function from the non-negative integers into the positive real numbers. Then $\text{o}(g)$ is the set of function f , also from the non-negative integers into the positive real numbers, such that $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$.

Example :

$$\rightarrow f(n) = 3n + 5$$

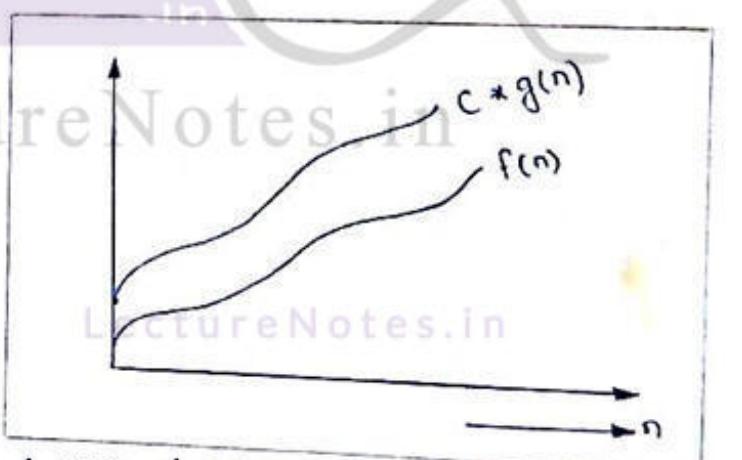
$$\text{now } \lim_{n \rightarrow \infty} \frac{3n+5}{n} = 3$$

$$3 \neq 0.$$

$$\text{thus } f(n) \neq \text{o}(n)$$

$$\text{if } \lim_{n \rightarrow \infty} \frac{3n+5}{n^2} = 0$$

$$\text{so } f(n) = \text{o}(n^2),$$



$$\rightarrow f(n) = 4n^3 + 2n + 3$$

$$\text{now } \lim_{n \rightarrow \infty} \frac{4n^3 + 2n + 3}{n^3} = 4, 4 \neq 0$$

$$\text{so } f(n) \neq \text{o}(n^3).$$

$$\text{But } f(n) = \text{o}(n^4),$$

5. Little omega (ω) Notation :

" The function $f(n) = \omega(g(n))$ iff there exist a positive constant $c > 0$ and $n_0 > 0$ such that

$$f(n) > c * g(n) \text{ for all } n, n \geq n_0 ."$$

Example :

$$\rightarrow f(n) = 3n + 5 = \omega(1)$$

$$\rightarrow f(n) = 2n + 5 = \omega(1)$$

$$\rightarrow f(n) = 27n^2 + 16n = \omega(n)$$

$$\rightarrow f(n) = 3n^3 + 4n = \omega(n^2)$$

Little omega (ω) Ratio Theorem :

Considering 'g' be a set of function from the non-negative integers into the positive real numbers then $\omega(g)$ is the set of function 'f' also from the non-negative integers into the positive real numbers, such that

$$\lim_{n \rightarrow \infty} f(n) / g(n) = \infty$$

Example :

$$\rightarrow f(n) = 3n + 5$$

$$\text{now } \lim_{n \rightarrow \infty} \frac{3n+5}{1} = \infty$$

$$\text{so } f(n) = \omega(1) //$$

$$\rightarrow f(n) = 27n^2 + 16n$$

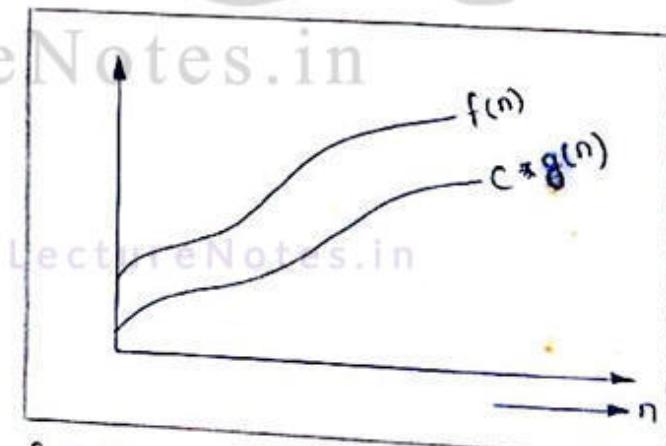
$$\text{now } \lim_{n \rightarrow \infty} \frac{27n^2 + 16n}{n} = \infty$$

$$\text{so } f(n) = \omega(n) //$$

$$\rightarrow f(n) = 27n^3 + 2n + 3$$

$$\text{now } \lim_{n \rightarrow \infty} \frac{27n^3 + 2n + 3}{n^2} = \infty$$

$$\text{so } f(n) = \omega(n^2) //$$



[Little omega (ω) notation ,
 $f(n) > c * g(n)$]



Data Structure Using C

Topic:
Data Structure

Contributed By:
Mamata Garanayak

Lesson Number : 2Data Structure :

Definition : The logical or mathematical model of a particular organization of data is called data structure.

Objective of Data structure :

The study of data structure has two objectives.

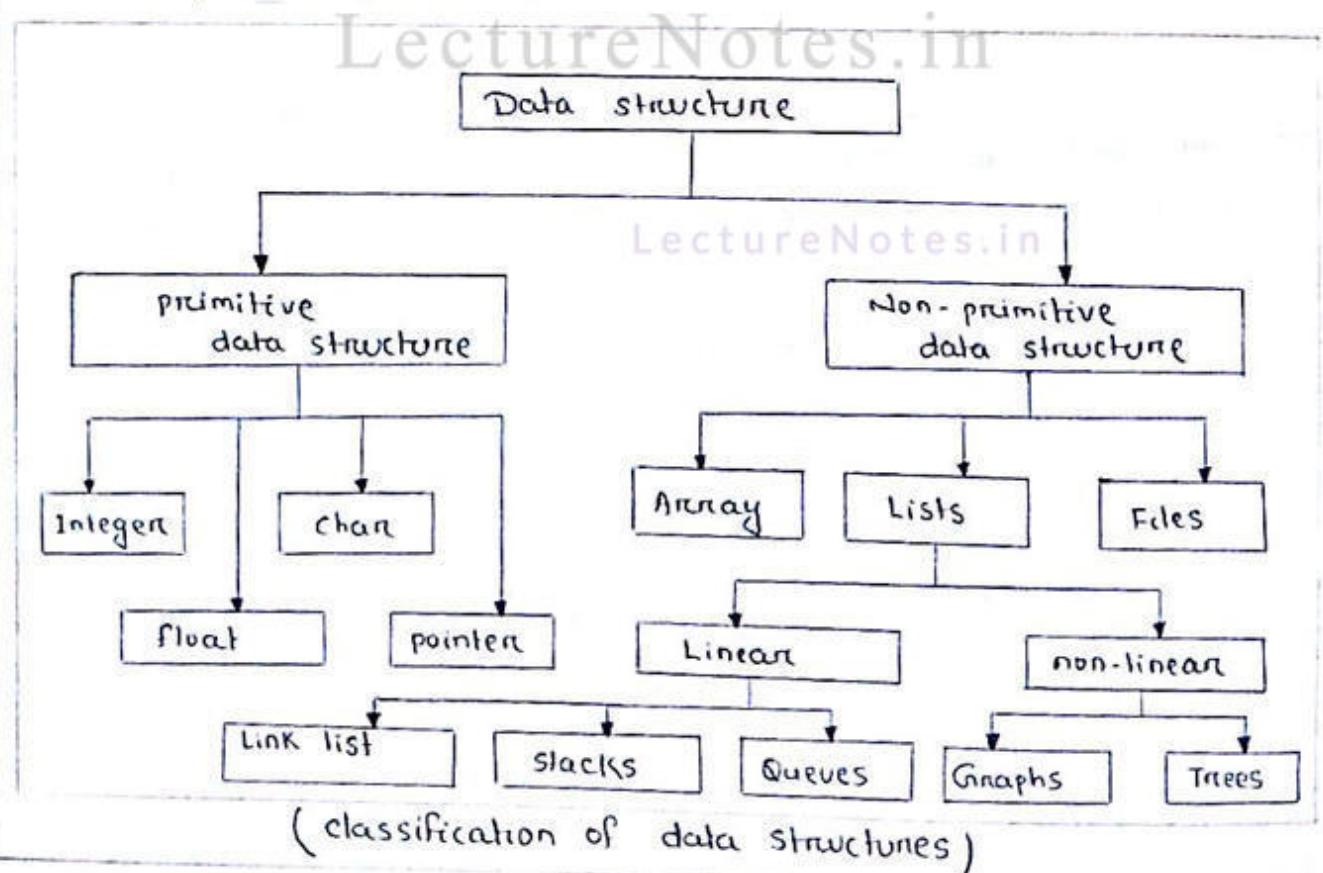
- First is to identify and create useful mathematical entities and operations for the problem which need to be solved.
- The second is to determine the representation of entities and the implementation of operations on these problems.

Data structure represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increase the efficiency.

Classification of Data structure :

Data structures are normally divided into two broad categories.

- 1) primitive data structure
- 2) Non-primitive data structure



(1) primitive data structures:

- The primitive data structures are the basic structures and are directly operated upon by the machine instructions.
- These are having different representation in different computers.
- Example : Integer takes 2-bytes in memory in a 16-bit compiler, whereas in 32-bit compiler it takes 32-bit.
- Examples of primitive data structures are : integer, float, char pointers.

(2) Non-primitive data structures :

- The non-primitive data structures are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- For the non-primitive data structures, the memory for a data item get allocated as per the user requirement.
- Example : Array, lists, files.

Linear

In linear data structures, the data items are arranged in a linear sequence.

Example : Array, Linked List

Non-linear

In non-linear data structures, the data items are not arranged in a linear sequence.

Example : Tree, Graph

Homogeneous

In homogeneous data structure all the elements are of same type.

Example : Array

Non-homogeneous

In Non-homogeneous data structure the elements may or may not be of the same type.

Example : Records

Static

Static structures are ones whose size and structures associated memory locations are fixed at compile time.

Example : Array

Dynamic

Dynamic structures are ones which expand or shrink as required during program execution and their associated memory locations change.

Example : Linked list created using pointers .

LectureNotes.in

Data Structure Operations :

Large number of operations can be performed on data structures. Some of the important operations are ;

1. Creating : Creation of a data structure
2. Inserting : New items are added to the data structure
3. Updating : Modifying the data
4. Traversing : visiting each data element once in a given data structure .
5. Searching : Search for a given data in data structure .
6. Deleting : Removing an item from the data structure
7. Sorting : Arrangement of data element in a give data structure
8. Merging : Merge of one or more than one homogeneous data structure .
9. Copying : Data items from one structure are copied to another structure .
10. Concatenating : Data items of a structure are appended at the end of another same type of structure .
11. Splitting : Data items in a very big structure are split into small structures for processing .

Assignment: what do you mean by static data structure and dynamic data structure.

- static data structures are ones whose sizes and structures associated memory locations are fixed at compile time.
- Dynamic data structures are ones which expand or shrink as required during program execution and their associated memory location change.

Example: linked list created using pointers.

Assignment: What is 'program proving'?

Verification of algorithm whether it is producing correct output for all possible combination of input values or not is called program proving or program verification.

Assignment: What are the two issues which is to be deal properly during algorithm design?

The two issues which is to be deal properly during algorithm design is

- 1) Validation of algorithm.
- 2) Evaluating the complexity of algorithm.

Assignment: List out the areas in which data structures are applied extensively.

- Compiler design → Graphics
- Operating system → AI
- DBMS

Assignment: what are the major D.S. used in following areas.
RDBMS, Network data model, Hierarchical data model.

RDBMS → Array.

n/w data model → Graph

Hierarchical data model → Tree

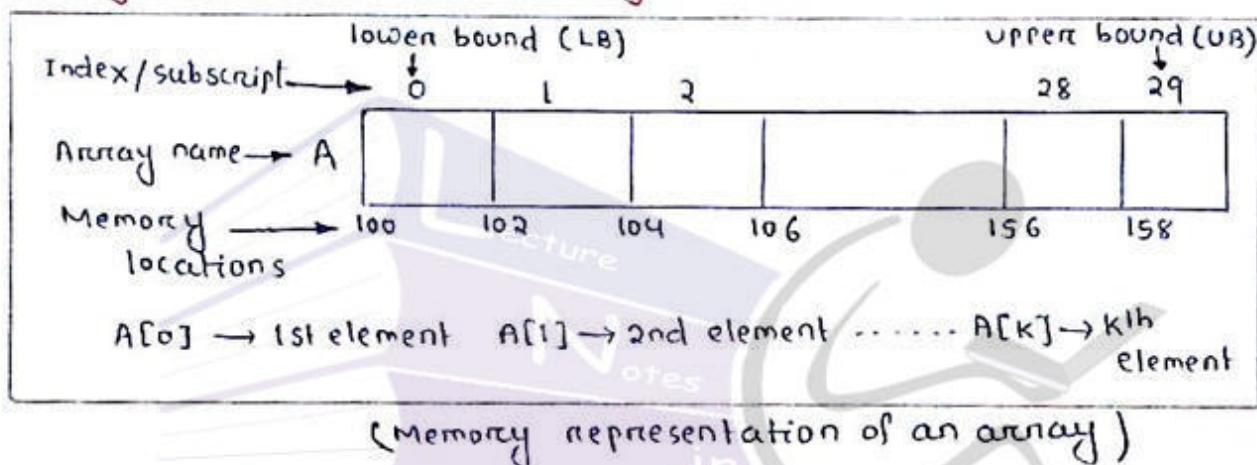
word : word denotes the size of an element. In each memory location, computer can store an element of word size w , say. This word size varies from machine to machine.

One-Dimensional Array :

" If only one subscript / index is required to reference all the elements in an array then the array will be termed as one-dimensional array or simply an array."

Example : `int a[50];`

Memory allocation for an array :



Size or length :

→ The size or length of the array is the highest number of the element, that the array can accommodate.

$$\text{Size} = \text{UB} - \text{LB} + 1$$

→ In the above declaration of the array

$$\text{UB} = 29$$

$$\text{LB} = 0$$

$$\begin{aligned}\text{size} &= \text{UB} - \text{LB} + 1 \\ &= 29 - 0 + 1 = 30\end{aligned}$$

$$\Rightarrow \text{size} = 30$$

NOTE : The LB can also other than the zero (i.e. any +ve no.).
But in programming concept we are considering the lower bound (LB) as 0.

Array :

Definition: "An array is a linear data structure having finite number 'n' of homogeneous data elements (i.e. data elements of the same type) stored in a contiguous memory locations with a single name."

Example: `int A[30]`

Here 'A' is an array name, which can hold 30 numbers of integer data.

Or "An array is a finite, ordered and collection of homogeneous data elements."

- Array is finite because it contains only limited number of elements; and ordered, as all the elements are stored one by one in contiguous locations of computer memory in a linear ordered fashion. All the elements of an array are of the same data type. Hence it is termed as collection of homogeneous elements.

Terminology :

Size : Number of elements in an array is called the size of the array. It is also alternatively termed as length or dimension.

Type : Type of an array represents the kind of data type it is meant for. For example array of integer, array of character string etc.

Base : Base of an array is the address of memory location where the first element in the array is located.

Index : All the elements in an array can be referenced by a subscript like a_i or $a[i]$, this subscript is known as index. Index is always an integer value. As each array element is identified by a subscript or index that is why an array element is also termed as subscripted or index variable.

Range of index : Indices of array elements may change from a lower bound (0) to an upper bound ($u-1$), which are called the boundaries of an array.

Example: An automobile company uses an array named AUTO to record the no. of automobile sold each year from 1932 through 1984. Rather than begining the index set with 1, it is more useful to begin the index set with 1932 so that

$\text{AUTO}[k] = \text{No. of automobiles sold in the year } 'k'$.

Hence $\text{UB} = 1984$

$\text{LB} = 1932$

$$\text{size} = 1984 - 1932 + 1 = 53$$

It shows that AUTO contains 53 elements and its index set consists of all integers from 1932 through 1984.

- The elements are stored in successive memory locations in an array. The computer can only keep track of base address.

$\text{Loc}(A[k]) = \text{Address of element } A[k]$ of the array A .

$$\boxed{\text{Loc}(A[k]) = \text{base}(A) + w(k - \text{lower bound})}$$

$$\begin{aligned}\text{Loc}(A[28]) &= 100 + 2(28 - 0) \\ &= 100 + 56 \\ &= 156\end{aligned}$$

A	0	1	2	...	28	29
	100	102	104	...	156	158

$$\begin{aligned}\text{Loc}(A[2]) &= 100 + 2(2 - 0) \\ &= 104\end{aligned}$$

For integer $w = 2$,
float $w = 4$,
double $w = 8$
char $w = 1$

LectureNotes.in

Various operations in an array :

The various operations in an array are,

- Traversing
- Deleting
- Updating
- searching
- sorting
- Inserting
- Merging

Traversal :

If we want to print the contents or search a given item or count the no. of elements of an array then this can be accomplished by traversing.

Definition: " Accessing and processing (i.e. visiting) each elements of an array exactly once is called traversing."

Algorithm for Traversing :

PROCEDURE TRAVERSE (LB, A, n, K)

// A : Name of the linear array .

// n : no. of elements

// K : Index variable

// LB : Lower bound

Step 1 : set K = LB [Initialize the Counter]

Step 2 : Repeat for K = LB to 'n'

a. Apply PROCESS to A[K]

End of for loop

Step 3 : Exit .

Hence PROCESS may be for displaying or counting or Searching .

C - program : Displaying the contents of an array .

Void traverse (int a[], int n)

{
 int K ;

```

for ( k = 0 ; k < n ; k++ )
    printf (" %d\t", a[k]);
}

```

Insertion : " Addition of a new element to the array in a given position is called insertion."

- Inserting an element at the end of a linear array can be easily done provided that the memory space allocated for the array is large enough to accomodate the additional element.
- Other than that inserting at any position required to move the elements of the array towards right to new locations to accomodate the new element and keep the order of the other elements.

NOTE : At the time of insertion we need to check the overflow, i.e. whether there exist enough memory space in the array to accomodate the new element.

Suppose 'n' is the number of element in an existing array. "size" is the total no of elements that can be accomodated in the array.

if ($n+1 > \text{size}$) then overflow occurs.

Algorithm for insertion:

LectureNotes.in

PROCEDURE INSERT (A, n, size, K, item, Loc)

// A : Linear array

// n : no. of elements in the existing array

// K : Index variable

// item : The new element that is to be inserted.

// Loc : The position where the item is to be inserted.

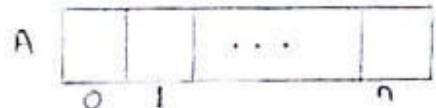
// size : Size of an array

Step 1 : [check for overflow]

if ($n+1 > \text{size}$)

a. print overflow

b. otherwise goto step 5



Step 2 : for $K = n-1$ to $\text{LOC}-1$

or [$\text{for}(K = n-1; K \geq \text{LOC}-1; K--)$]

a. $A[K+1] = A[K]$ // shifting the elements towards right

b. End for

Step 3 : $A[\text{LOC}-1] = \text{item}$

Step 4 : $n = n+1$ // incrementing the no. of element by '1'

Step 5 : exit

C - procedure :

```
int Insert [ int A[], int size, int n, int item, int loc )
{
    int K;
    if ( n+1 > size )
    {
        printf ("overflow");
        exit();
    }
    else
    {
        for ( K=n-1 ; K >= LOC-1 ; K-- )
            A[K+1] = A[K];
        A[LOC-1] = item;
        n = n+1 ;
    }
    return n ;
}
```

C - Program for Insertion :

```
# include <stdio.h>
# include <Conio.h>
# define SIZE 100
int insertion ( int a[], int, int, int); /* Function prototype */
void main() void display ( int a[], int); /* Function
prototype */
{
    int a[SIZE], n, item, loc, k;
    clrscr();  
LectureNotes.in
    printf (" Enter the no. of elements into an array : ");
    scanf ("%d", &n);
    printf (" Enter the items : ");
    scanf ("%d", &item);
    printf (" Enter the location at which item will be inserted : ");
    scanf ("%d", &loc);
    for ( k=0; k<n; k++) /* creation of an array */
    {
        printf (" Enter an element : ");
        scanf ("%d", &a[k]);
    }
    printf (" The elements before insertion are as follows :\n");
    for ( k=0; k<n; k++) /* display of array elements */
        printf ("%d\t", a[k]);
    n = insertion ( a, n, item, loc); /* Function call */
    printf (" The no. of element after insertion is %d ", n);
    display ( a, n);
    getch();
}
int insertion ( int a[], int n, int item, int loc)
{
    int k;
```

```
if ( n+1 > SIZE )
{
    printf (" Overflow");
    exit();
}
else {
    for ( k = n-1; k >= ( loc-1 ); k-- )
        a[k+1] = a[k];
    a[loc-1] = item;
}
return (n+1);
}

void display (int a[], n)
{
    int k;
    for (k=0; k<n; k++)
        printf ("%d\t", a[k]);
}
```

LectureNotes.in

LectureNotes.in

Assignment: write a c procedure to add all even elements and all odd elements, display the result.

```
#include <stdio.h>
#include <Conio.h>
void main()
{
    int a[30], i, Seven = 0, Sodd = 0, n;
    clrscr();
    printf("Enter the no of element in an array : ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        printf("Enter a number : ");
        scanf("%d", &a[i]);
    }
    for (i = 0; i < n; i++)
    {
        if (a[i] % 2 == 0)
            Seven = Seven + a[i];
        else
            Sodd = Sodd + a[i];
    }
    printf("sum of even numbers = %d\n", Seven);
    printf("sum of odd numbers = %d\n", Sodd);
    getch();
}
```

Assignment: write a c - procedure to count no. of +ve , -ve and zero elements in an array .

```
# include <stdio.h>
# include <conio.h>
void main()
{
    int a[30], en, on, zn, n;
    en = 0;
    on = 0; LectureNotes.in
    zn = 0;
    printf ("Enter the no. of elements in the array : ");
    scanf ("%d", &n);
    for (i=0; i<n; i++)
    {
        printf ("Enter a number : ");
        scanf ("%d", &a[i]);
    }
    for (i=0; i<n; i++)
    {
        if (a[i] > 0)
            en++;
        else if (a[i] < 0)
            on++;
        else
            zn++;
    }
    printf ("No. of +ve no = %d\n", en);
    printf ("No. of -ve no = %d\n", on);
    printf ("No. of zero = %d\n", zn);
    getch();
}
```



Data Structure Using C

Topic:
Deletion

Contributed By:
Mamata Garanayak

Lesson Number: 3

Deletion :

- Deletion means, removing an element from array . from a given position.
- Deleting an element at the end of an array presents no difficulties but deleting an element some where in the middle of the array would require that each subsequent element be moved one location left the array .

NOTE : LectureNotes.in

At the time of deletion , we need to check the "Underflow" condition i.e. if there exist no element in the array then no element can't be deleted .

$\text{if } (n == 0)$ is the underflow condition where
 n = no of element in the array then the following process is to be followed at the time of deletion .

- check the underflow condition .
- Before deletion store the item .
- Move the subsequent element to the left to the array .
- Decrement the no. of element in the array .

Algorithm for Deletion :

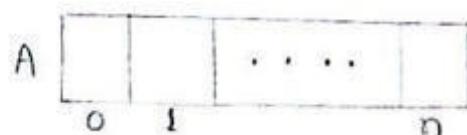
PROCEDURE Deletion (A, n, Ditem, Loc, K)

// Ditem = gt will store the item that is to be deleted from the position Loc .

Step 1 : $\text{if } (n == 0)$

a. print "underflow".

b. exit



Step 2 : Ditem = A[Loc-1]

Step 3 : for (K = Loc-1; K < n-1; K++)

a. $A[K] = A[K+1]$

b. End for

Step 4 : $n = n-1$ [Decrementing the no. of element]

Step 5 : The Deleted item is Ditem .

Step 6 : Exit .

C - procedure :

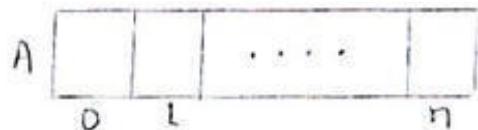
```
int deletion ( int A[], int n, int LOC )
```

```
{
```

```
    int K , Ditem ;
```

```
    if ( n == 0 )
```

```
        printf (" underflow " );
```



```
else {
```

```
    Ditem = A[LOC]-1 ]
```

```
    for ( K=LOC-1; K<n-1; K++ )
```

```
        A[K] = A[K+1] ;
```

```
    n = n-1 ;
```

```
    printf (" The deleted item = %d ", Ditem ) ;
```

```
}
```

```
return n ;
```

```
}
```

NOTE: For insertion or deletion it is being assumed that the position (ie LOC) is $\leq n$ (ie. $LOC \leq n$) where n is the no. of element in the given array.

Searching :

→ Finding the position of the element with a given value on the record with a given key is known as searching.

→ Example :

a	12	53	36	31	92	25
	0	1	2	3	4	5

Hence 'a' is an array having 6 elements. Suppose we want to search whether '36' is present in the array or not? We will start traversing the array from beginning and the "item = 36" will be checked with each element in the array. Here 36 found in the position '3', so we can say search is successful. Otherwise it fails.

Algorithm :

Search (A, n, item, pos)

// A : An existing linear array.

// n : no. of elements in the array.

// item : The item that is to be searched

// pos : The position where item is found.

Step 1 : pos = 0, k = 0

Step 2 : ~~for (k=0;~~ while (k < n && A[k] != item)

a. k = k + 1

b. End while

Step 3 : if (A[k] == item)

a. pos = ~~k+1~~ + 1

b. print the item is found in position "pos".

Step 4 : if (k == n)

a. The item is not found .

Step 5 : exit

C - Procedure :

```
void search (int A[], int pos, int n);  
{  
    int k;  
    for (k = 0; k < n && A[k] != item; k++)  
    ;  
    if (A[k] == item)  
    {  
        pos = k + 1; eNotes.in  
        printf ("The item is found = %d", pos);  
    }  
    if (k == n)  
        printf ("The item is not found");  
}
```

Update :

→ Modifying the existing data element in an array is called updating.

Algorithm : update (A, item, pos)

// A : An existing linear array.

// pos : which position . element is to be updated.

// item : The new item which will replace the old one

Step-1 : $A[POS] \leftarrow item$. $A[POS-1] = item$.

Step-2 : exit .

C - procedure :

```
void update (int a[], int item, int pos)  
{  
    a[pos - 1] = item;  
}
```

Updation:

Updation means modifying the data.

Algorithm:

PROCEDURE Updation (a, pos, new-item)

// a : Linear array i.e. existing once

// pos : position of data element which is to be updated

// new-item : value of new element by which old data element is
modified

Step 1 : if ($n == 0$)

a) print underflow .

b) exit .

Step 2 : $a[pos - 1] = \text{new-item}$

Step 3 : exit

C - procedure :

```
void Update ( int a[], int pos, int new-item )
```

```
{
```

```
    if ( n == 0 )
```

```
{
```

```
        printf (" Underflow " );
```

```
        exit ( ) ;
```

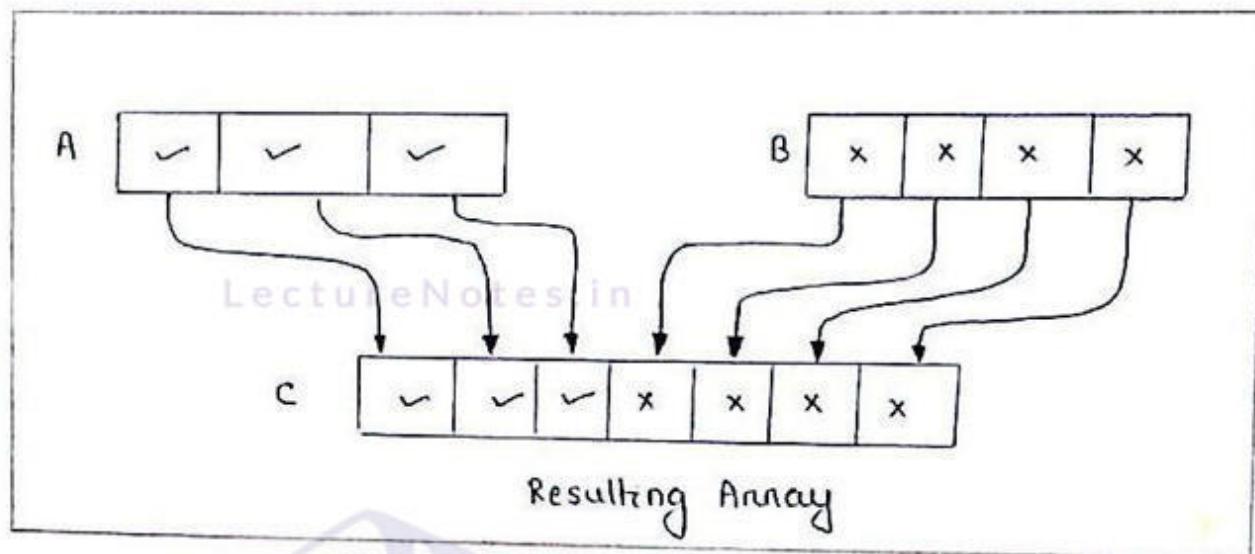
```
}
```

```
    a[ pos - 1 ] = new-item ;
```

```
}
```

Merging :

→ Combining two or more than one array to a single array is called as merging.



(Merging of two array)

Here, 'A' and 'B' are 2- linear existing array. 'A' having 3-elements and 'B' having 4-elements. 'C' is the resultant array which keeps the elements of 'A' as well as 'B'. Now the 'C'-array having 7 elements.

In first step, all the elements of 'A' will be copied to array 'C'. Then all the elements of 'B' will be copied to array 'C'.

NOTE : Length of Array ('A' + 'B') = length of array 'C'.

Algorithm :

Merge (A, B, C, m, n, i, j, k)

// A : Linear existing array

// B : Another linear array .

// m : No. of elements in an array 'A' ,

// n : No. of elements in an array 'B' .

// i, j, k : These are the index variable for the array A, B & C respectively .

// C : Resultant array .

Step 1 : i = j = k = 0

Step 2: while (i < m)

a. C[k] = A[i]

b. k++

c. i++

{ [end while] }

Step 3: while (j < n)

a. C[k] = B[j]

b. k++

c. j++

Step 4: Exit

C- procedure :

```
void merge ( int a[], int b[], m, n, int c[] )
```

```
{
```

```
    int i, j, k=0;
```

```
    for ( i=0; i < m ; i++ , k++ )
```

```
        c[k] = a[i];
```

```
    for ( j=0; j < n ; j++ , k++ )
```

```
        c[k] = b[j];
```

```
    printf ( " The resultant merged array is : " );
```

```
    for ( k=0; k < m+n ; k++ )
```

```
        printf ( "%d\t" , c[k] );
```

```
}
```

Assignment: write a program to search a given item and update it as per the user by a new item.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a[30], i, n, item, newitem, pos = -1;
    clrscr();
    printf("Enter the no. of elements in the array : ");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        printf("Enter a no : ");
        scanf("%d", &a[i]);
    }
    printf("Enter the item which is to be updated");
    scanf("%d", &item);
    printf("Enter the newitem");
    scanf("%d", &newitem);
    for(i=0; i<n; i++)
    {
        if(a[i] == item)
        {
            pos = i;
            break;
        }
    }
    if(pos != -1)
        a[pos] = newitem;
    else
        printf("The item not found for updation");
}
getch();
```

<1> Write a program to perform various operations in an array.

Operations are :

- 1) Traverse 4) Merge
- 2) insert 5) Search
- 3) Delete 6) update

```
# include <stdio.h>
# include <conio.h>
int a[80], n, i;
void main()
{
    int p;
    void traverse();
    void insert();
    void delete();
    void merge();
    void search();
    void update();
    clrscr();
    printf ("Enter the size of the array :");
    scanf ("%d", &n);
    printf ("\n Enter the elements of the array :");
    for( i=0; i<n; i++)
    {
        scanf ("%d", &a[i]);
    }
    printf ("The array is :\n");
    for( i=0; i<n; i++)
    {
        printf ("%d\t", a[i]);
    }
}
```

```

while (1)
{
    printf ("\n Enter 1 for traverse :\n");
    printf ("\n Enter 2 for insert :\n");
    printf ("\n Enter 3 for delete :\n");
    printf ("\n Enter 4 for merge :\n");
    printf ("\n Enter 5 for search :\n");
    printf ("\n Enter 6 for update :\n");
    printf ("\n Enter 7 for exit :\n");

    printf (" Enter your choice :");
    scanf ("%d", &p);

    switch (p)
    {
        Case 1 :
            traverse();
            break;

        Case 2 :
            insert();
            break;

        Case 3 :
            delete();
            break;

        Case 4 :
            merge();
            break;

        Case 5 :
            search();
            break;

        Case 6 :
            update();
            break;
    }
}

```

Case 7 :

```
    exit();
    break;

default :
    printf ("Wrong choice");
    break;
}
}
}
```

LectureNotes.in

void traverse()

```
{
    printf ("The array is:\n");
    for (i=0; i<n; i++)
    {
        printf ("%d\n", a[i]);
    }
}
```

void delete()

```
{
    int item, k, loc;
    if (n == 0)
    {
        printf ("Underflow\n");
        exit();
    }
    else
    {
        printf ("Enter the position where the item is to be deleted:");
        scanf ("%d", &loc);
        item = a[loc];
        a[loc] = a[n-1];
        printf ("The deleted item is : %d\n", item);
        for (k=loc+1; k<n-1; k++)
        {
            a[k] = a[k+1];
        }
    }
}
```

```

a[k] = a[k+1];
}
printf ("The new array is :");
for (k=0; k<n-1; k++)
    printf ("%d", a[k]);
}

void insert()
{
    int item, k, loc;
    printf ("Enter the item to be inserted :");
    scanf ("%d", &item);
    printf ("Enter the location at which the item is to be inserted :");
    scanf ("%d", &loc);
    for (k=n; k>=loc-1; k--)
    {
        a[k+1] = a[k];
    }
    a[loc-1] = item;
    printf ("The new array is :\n");
    for (k=0; k<n+1; k++)
    {
        printf ("%d", a[k]);
    }
}

void search()
{
    int i=0, pos=0, item;
    printf ("Enter the item :");
    scanf ("%d", &item);
    while ((i < n) && a[i] != item)
    {
}

```

```

    i = i + 1 ;
}

if (a[i] == item)
{
    pos = i ;
    printf ("\n The item is found at position : %d", pos+1);
}

else if (i == n)
{
    printf ("Item is not found in an array");
}
}

void merge()
{
    int m, c[20], b[10], j=0, k=0;
    printf ("Enter the size of new array : ");
    scanf ("%d", &m);

    printf ("Enter the elements of new array : \t");
    for (i=0; i<m; i++)
        scanf ("%d", &b[i]);
    i=0;

    while (i<n)
    {
        c[k] = a[i];
        k++;
        i++;
    }

    while (j<m)
    {
        c[k] = b[j];
        k++;
        j++;
    }

    printf ("The merged array is : ");
}

```

```
for (i=0; i<m+n; i++)
{
    printf ("%d", c[i]);
}

void update()
{
    int newitem, pos;
    printf (" Enter the position at which the item is to be updated:")
    scanf ("%d", &pos);
    printf (" Enter the new item :");
    scanf ("%d", &newitem);
    a [pos-1] = newitem;
    for(i=0; i<n; i++)
    {
        printf ("%d", a[i]);
    }
}
```



Data Structure Using C

Topic:
Two-Dimensional Array

Contributed By:
Mamata Garanayak

Two-Dimensional Array :

- Two-dimensional arrays (alternatively termed as matrices) are the collection of homogeneous elements where the elements are ordered in a number of rows and columns.
- An example of $m \times n$ matrix, where m denotes the number of rows and n denotes the no. of columns is as follows.

a_{00}	a_{01}	a_{02}	...	a_{0n-1}
a_{10}	a_{11}	a_{12}	...	a_{1n-1}
a_{20}	a_{21}	a_{22}	...	a_{2n-1}
.
.
a_{m-10}	a_{m-11}	a_{m-12}	...	a_{m-1n-1}

- The subscripts of any arbitrary element, say (a_{ij}) , represents the i -th row and j -th column.

Memory Representation of a Matrix :

Like one-dimensional array, matrices are also stored in contiguous memory locations. There are two conventions of storing any matrix in memory.

- Row-major Order
- Column-major Order

a. Row-major order :

- In row major order, elements of matrix are stored on a row-by-row basis, that is, all the elements in first row, then in second row and so on.
- For an example : `int a[3][4];`

where a is the name of the array, int represents the type of data to be stored in array elements, 3 represents the max^m no. of rows and 4 represents the max^m no. of columns.

→ The logical representation of the array a of order 3×4 is;

$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$

→ Row-major representation of $a[3][4]$ is

$$\begin{array}{cccc} a[0][0] & \rightarrow & a[0][1] & \rightarrow a[0][2] & \rightarrow a[0][3] \\ \leftarrow & a[1][0] & \rightarrow a[1][1] & \rightarrow a[1][2] & \rightarrow a[1][3] \\ \leftarrow & a[2][0] & \rightarrow a[2][1] & \rightarrow a[2][2] & \rightarrow a[2][3] \end{array}$$

→ The storage representation of two dimensional array in row major order is;

Row 0	{	$a[0][0]$	1	bare(a)
		$a[0][1]$	3	100
		$a[0][2]$	5	102
		$a[0][3]$	2	104
				106
Row 1	{	$a[1][0]$	17	108
		$a[1][1]$	9	110
		$a[1][2]$	10	112
		$a[1][3]$	51	114
Row 2	{	$a[2][0]$	32	116
		$a[2][1]$	19	118
		$a[2][2]$	13	120
		$a[2][3]$	6	122

(storage representation in row-major order (row-by-row))

→ Address calculation of matrix elements in memory in row-major order is;

$$\text{loc}(a[i][j]) = \text{bare}(a) + \omega [N * i + j]$$

where N = Total no. of columns

ω = data type size.

$\text{bare}(a)$ = base address of matrix a .

Example :

$$\begin{aligned}
 \text{loc}(\alpha[2][3]) &= \text{base}(\alpha) + \omega [n(i,:) + (j,:)] \\
 &= 100 + 2 [4(2,:) + (3,:)] \\
 &= 100 + 2 [8 + 3] \\
 &= 100 + 122 \\
 &= 122
 \end{aligned}$$

So, the address of 2nd row, 3rd column is 122.

b. column-major order:

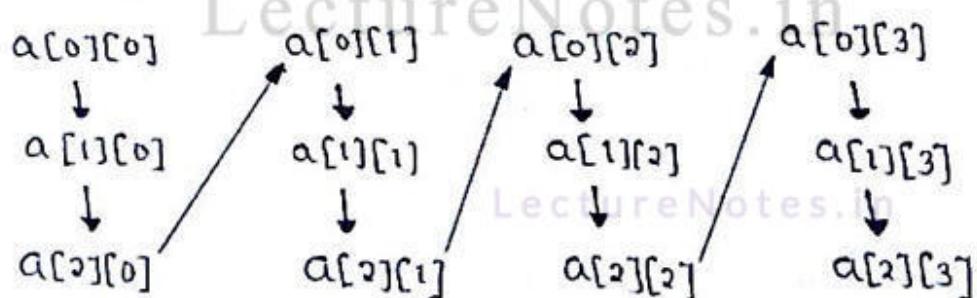
→ In Column-major order, elements are stored column-by-column, that is, all the elements in first column are stored in their order of rows, then in second column, third column and so on.

→ Example : `int a[3][4];`

→ The logical representation of the above array α of order 3×4 is;

$$\begin{bmatrix}
 \alpha[0][0] & \alpha[0][1] & \alpha[0][2] & \alpha[0][3] \\
 \alpha[1][0] & \alpha[1][1] & \alpha[1][2] & \alpha[1][3] \\
 \alpha[2][0] & \alpha[2][1] & \alpha[2][2] & \alpha[2][3]
 \end{bmatrix}$$

→ Column-major representation of $\alpha[3][4]$ is ;



→ The storage representation of two dimensional array in column-major order is ;

			bare(a)
Column 0	$a[0][0]$	7	100
	$a[1][0]$	17	102
	$a[2][0]$	32	104
Column 1	$a[0][1]$	3	106
	$a[1][1]$	09	108
	$a[2][1]$	19	110
Column 2	$a[0][2]$	5	112
	$a[1][2]$	10	114
	$a[2][2]$	13	116
Column 3	$a[0][3]$	2	118
	$a[1][3]$	51	120
	$a[2][3]$	6	122

(storage representation in column-major order (column-by-column))

→ Address calculation of matrix elements in memory in column-major order is :

$$\text{loc}(a[i][j]) = \text{bare}(a) + w[M(j) + i])$$

where M = Total no. of rows

w = data type size

$\text{bare}(a)$ = base address of matrix a .

Example : $\text{loc}(a[2][3]) = \text{bare}(a) + w[M(j) + (i)])$

$$\begin{aligned}
 &= 100 + 2 [3(3-1) + (2-1)] \\
 &= 100 + 2 \{ [3 * 3] + 2 \} \\
 &= 100 + 122 //
 \end{aligned}$$

The address of 2nd row and 3rd column is ~~100 122~~ //
122 //

Sparse Matrix :

- Matrices, which have more zero entries are called as sparse matrices.
- The matrix where all entries above the main diagonal are zero or equivalently, where non zero entries can only occur on or below the main diagonal is called a (lower) triangular matrix.
- The second type matrix, where non-zero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a tridiagonal matrix.

$$\begin{bmatrix} 5 \\ -6 & 9 \\ 1 & 5 & 9 \\ 13 & -92 & -55 & 67 \end{bmatrix}$$

(Triangular - Matrix)

$$\begin{bmatrix} -5 & -14 \\ 52 & 19 & 21 \\ & -61 & -17 & -53 \\ & -31 & 13 \end{bmatrix}$$

(Tridiagonal Matrix)

Representation of sparse Matrix :

- Matrix has presentation in row and column only.
- So it can be presented in two dimensional array.
- Suppose a matrix has many zero entries and we have work only with non zero entries. So it's a wastage of memory.
- Now we want to save the memory space which every zero entry is using. So alternate method is to save only non-zero entries which can be through sparse matrix.
- The 3-tuple method is used for presenting sparse matrix.

3-tuple Method :

- As the name implies, every nonzero entry of sparse matrix represented by three tuples.
- First tuple represents row, second for column and third for values.
- Here all the elements of matrix will come sequentially.

- It can be row major or column major 3-tuple presentation of sparse matrix.
 - First row will represent the number of rows, number of columns and no. of non-zero elements of matrix.
 - After that each row will represent non zero entries of matrix.
- Let us take a matrix of 4×5 .

	0	-4	5	0	1
Matrix $M_{4 \times 5} = 0$	9	0	0	2	0
	0	3	0	0	0
	6	0	0	0	12

(sparse matrix)

- Now we have to represent this matrix in 3-tuple representation of sparse matrix. Here total no. of rows = 4, total number of column = 5 and non-zero entries are 8.

- So sparse matrix will be :

Row	Column	Value
4	5	8
1	2	-4
1	3	5
1	5	1
2	1	9
2	4	2
3	2	3
4	1	6
4	5	12

(row-major 3-tuple presentation)

- Here it's a row major 3-tuple presentation. so elements are coming sequentially on the basis of row.
- Suppose we want to represent column major 3-tuple sparse matrix then it will be as -

<u>Column</u>	<u>Row</u>	<u>value</u>
5	4	8
1	2	9
1	4	6
2	1	-4
2	3	3
3	1	5
4	2	2
5	1	1
5	4	12

LectureNotes.in (column-major 3-tuple presentation)

Algorithm:

Create - sparse (A, SP, n, c, nz, i, j, K)

// A : a two dimensional matrix .

// SP : a two dimensional matrix is to be created as a sparse matrix having 3-no. of Col.

// n : No. of row in the matrix 'A' .

// c : No. of column in the matrix 'A' .

// nz : No. of nonzero elements in matrix 'A' .

// i, j, K : are the index variables for row and column respectively for 'A' and K is the row index for SP .

Step1 : K = 0 , SP[0][0] = n , SP[0][1] = c , SP[0][2] = nz .

Step2 : for (i=0 , i < n ; i++)

 for (j=0 , j < c ; j++)

 (a) if (A[i][j] != 0)

 (i) K++

 (ii) SP[K][0] = i

 (iii) SP[K][1] = j

 (iv) SP[K][2] = A[i][j]

 [End for 1]

 [End for 2]

Step 3 : exit

C - procedure :

```

Create - sparse ( int a[ ][ ] , int sp[ ][ 3 ] , int n , int c , int nz )
{
    int k=0, i, j;
    sp[k][0] = n;
    sp[k][1] = c;
    sp[k][2] = nz;
    for ( i=0; i<n; i++ )
        for ( j=0; j<c; j++ )
            {
                if ( a[i][j] != 0 )
                    {
                        k++;
                        sp[k][0] = i;
                        sp[k][1] = j;
                        sp[k][2] = a[i][j];
                    }
            }
}

```

Displaying the Sparse matrix :

```

C - procedure ( int sp[ ][ 3 ] )
{
    int k;
    for ( k=0; k<=sp[0][2]; k++ )
        printf( "%d %d %d \n", sp[k][0], sp[k][1], sp[k][2] );
}

```

Transpose of sparse Matrix :

→ In this process only thing that is to be done for getting the transpose is that, changing the row-value (r_n) to the column value (c_n) in an existing transpose matrix.

→ Let

$$\text{Matrix} = \begin{bmatrix} 0 & 37 \\ 52 & 0 \\ 0 & 19 \end{bmatrix}_{3 \times 2}$$

<u>rn</u>	<u>cn</u>	<u>nz</u>
3	2	3
0	1	37
1	0	52
2	1	19

- This sparse matrix show that the original matrix having 3 no. of row, 2 no. of column and 3 no of non zero values.
- The non-zero elements are shown in the row 1, 2, & 3 only with the row no. & col. no. in the sparse matrix.
- If we will interchange rn and cn values, we can get our required result.
- The transpose of the above sparse matrix is as follows :

<u>rn</u>	<u>cn</u>	<u>nz</u>
2	3	3
1	0	37
0	1	52
1	2	19

(Transpose of sparse matrix)

C - Procedure :

```
void transpose - Sparse (int SP[3][3], int TSP[3][3])
{
    int i;
    for (i=0; i<=SP[0][2]; i++)
    {
        TSP[i][0] = SP[i][1];
        TSP[i][1] = SP[i][0];
        TSP[i][2] = SP[i][2];
    }
}
```

Assignment : Find the address of $a[3][2]$ where the matrix 'a' having no. of row=4, no of col=3 and base address = 213 . Assuming 'a' having the elements of integer type.

Soln? If an array named as 'a' having no. of col=c, then the address location for $a[i][j]$

$$= a + (i * \text{col} + j) * \text{sizeof}(\text{data type of an array}) .$$

\downarrow
Base address

Now the address for $a[3][2]$

$$\begin{aligned} &= a + (3 * \text{col} + 2) * \text{sizeof}(\text{int}) \\ &= 213 + (3 * 3 + 2) * 2 \\ &= 213 + 22 \\ &= 235 \end{aligned}$$

LectureNotes.in

LectureNotes.in

Assignment: Write a C-procedure to find the original matrix from a given sparse matrix.

```
void (int SP[ ][3], int M[ ][ ] )  
{  
    int i, j, k=0;  
    for (i=0; i<SP[0][0]; i++)  
    {  
        for (j=0; j<SP[0][1]; j++)  
        {  
            if (i == SP[k][0] && j == SP[k][1])  
            {  
                M[i][j] = SP[k][2];  
                k++;  
            }  
            else  
            {  
                M[i][j] = 0;  
            }  
        }  
        printf ("\n");  
    }  
}
```

LectureNotes.in

LectureNotes.in

(2) Write a program to create a sparse matrix.

```
#include <stdio.h>
#include <conio.h>
#define SROW 50
#define MCOL 20
#define MROW 20

void main()
{
    int mat[MROW][MCOL], sparse[SROW][3];
    int i, j, nz=0, mrc, mc, srr, sr;
    clrscr();
    printf ("\nEnter the no. of rows :");
    scanf ("%d", &mrc);
    printf ("\nEnter the no. of columns :");
    scanf ("%d", &mc);
    printf ("Enter the elements of row and columns :");
    for(i=0; i<mrc; i++)
    {
        for(j=0; j<mc; j++)
        {
            scanf ("%d", &mat[i][j]);
        }
    }
    printf ("\n\nThe Entered matrix is :\n\n");
    for ( i=0 ; i<mrc ; i++ )
    {
        for ( j=0 ; j<mc ; j++ )
        {
            printf ("%d\t", mat[i][j]);
        }
        printf ("\n");
    }
    for( i=0 ; i<mrc ; i++ )
    {
        for ( j=0 , j<mc ; j++ )
        {
    }
```

```

if (mat[i][j] != 0)
{
    nz++;
}
}

printf ("\n");
sn = nz + 1;
printf ("\n Sparse Matrix is :\n\n");
sparse[0][0] = mr;
sparse[0][1] = mc;
sparse[0][2] = nz;

s = 1;
for (i=0; i<mr; i++)
{
    for (j=0; j<mc; j++)
    {
        if (mat[i][j] != 0)
        {
            sparse[s][0] = i+1;
            sparse[s][1] = j+1;
            sparse[s][2] = mat[i][j];
            s++;
        }
    }
}

for (i=0; i<sn; i++)
{
    for (j=0; j<3; j++)
    {
        printf ("%d\t", sparse[i][j]);
    }
    printf ("\n");
}

getch();
}

```

<3> Write a program to find the transpose of a sparse matrix.

```
# include <stdio.h>
# include <Conio.h>
# define SROW 50
# define MROW 20
# define MCOL 20
void transpose();
int mat[MROW][MCOL], sparse[SROW][3];
void main()
{
    int i, j, nz=0, mrr, mc, snr, s;
    clrscr();
    printf ("\n Enter the no. of rows :");
    scanf ("%d", &mrr);
    printf ("\n Enter the no. of columns :");
    scanf ("%d", &mc);
    printf ("\n Enter the elements of row and columns :");
    for(i=0; i<mrr; i++)
    {
        for (j=0; j<mc; j++)
        {
            {
                scanf ("%d", &mat[i][j]);
            }
        }
    }
    printf ("\n The Entered matrix is :\n\n");
    for (i=0; i<mrr; i++)
    {
        for (j=0; j<mc; j++)
        {
            {
                printf ("%d\t", mat[i][j]);
            }
        }
        printf ("\n");
    }
}
```

```

for (i=0; i<mnr; i++)
{
    for (j=0; j<mc; j++)
    {
        if (mat[i][j] != 0)
        {
            nz++;
        }
    }
}
printf ("\n");
sr = nz+1;
printf ("\n sparse matrix is :\n\n");
sparse [0][0] = mnr;
sparse [0][1] = mc;
sparse [0][2] = nz;
s=1;
for (i=0; i<mnr; i++)
{
    for (j=0; j<mc; j++)
    {
        if (mat[i][j] != 0)
        {
            sparse [s][0] = i+1;
            sparse [s][1] = j+1;
            sparse [s][2] = mat[i][j];
            s++;
        }
    }
}
for (i=0; i<sr; i++)
{
    for (j=0; j<3; j++)
    {
        printf (" +d\t", sparse[i][j]);
    }
}
printf ("\n");
}

```

```
transpose();
getch();
}

void transpose()
{
    int i, tsp[20][3];
    for (i=0; i<=sparse[0][2]; i++)
    {
        tsp[i][0] = sparse[i][1];
        tsp[i][1] = sparse[i][0];
        tsp[i][2] = sparse[i][2];
    }
    printf("The transpose of a sparse matrix is :\n");
    for (i=0; i<=sparse[0][2]; i++)
    {
        printf("%d %d %d\n", tsp[i][0], tsp[i][1], tsp[i][2]);
    }
}
```

— • —

LectureNotes.in

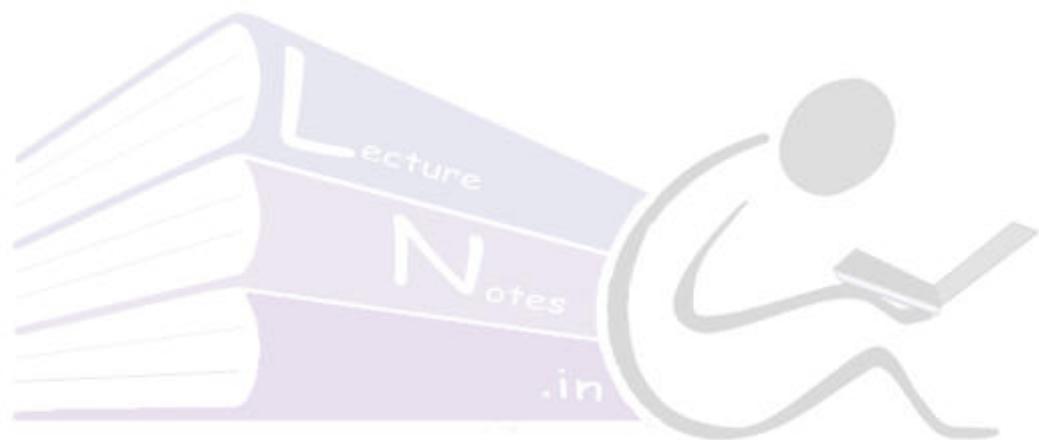
```

while (top >= 0)
{
    printf("%d", stack[top]);
    top--;
}

```

Soln Here our aim is to display the contents of the stack, but not delete. In stack decrementing the top value means deleting the top most element.

In this procedure top has been decremented, which logically incorrect as per traversal PROCESS.



LectureNotes.in

(4) Write a program to perform various operations such as
i. PUSH
ii. POP
iii. Traverse, on the stack.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define SIZE 80
int top = -1, stack[SIZE], i;
void traverse();
void push();
void pop();
void main()
{
    int p, n;
    clrscr();
    printf("Enter the size of a stack:");
    scanf("%d", &n);
    printf("\nEnter the elements of the stack:");
    for(i=0; i<n; i++)
    {
        scanf(".d", &stack[i]);
        top++;
    }
    printf("\nThe elements of the stack are:\n");
    for(i=top; i>=0; i--)
    {
        printf(".d\n", stack[i]);
    }
}
```

```

printf ("\n 1. For Traversal.\n");
printf ("\n 2. For push.\n");
printf ("\n 3. For pop.\n");
printf ("\n 4. For exit.\n");
printf ("\n Enter your choice :");
scanf ("%d", &r);

switch (r)
{
    case 1 :
        traverse();
        break;
    Case 2 :
        push();
        break;
    Case 3 :
        pop();
        break;
    case 4 :
        exit(1);
        break;
    default :
        printf ("Wrong Choice.");
        break;
}
}

void traverse()
{
    if (top == -1)
    {
        printf ("\n stack is empty.");
    }
}

```

```

        exit(1);
    }
else {
    for( i=top ; i>=0 ; i-- )
        printf( "\n%d\\t", stack[i] );
}
}

void push()
{
    int item;
    printf( "\nEnter the new item : " );
    scanf( "%d", &item );
    if ( top == SIZE - 1 )
    {
        printf( "Overflow" );
        exit(1);
    }
    else {
        top = top + 1 ;
        stack[top] = item ;
        for( i=top ; i>=0 ; i-- )
            printf( "\n%d\\t", stack[i] );
    }
}

void pop()
{
    if ( top == -1 )
    {
        printf( "Underflow" );
        exit(1);
    }
    else {
        int ditem ;
        ditem = stack[top] ;
        printf( "\nThe deleted item is : %d\\n", ditem ) ;
        top = top - 1 ;
        for( i=top ; i>=0 ; i-- )
            printf( "\n%d\\t", stack[i] );
    }
}

```



Data Structure Using C

Topic:
Stack

Contributed By:
Mamata Garanayak

Lesson Number : 5Stack :

- It is a linear data structure in which all insertion and deletions are made at one end called top of the stack.
- It follows the principles LIFO (Last in First Out) i.e. the element that will be inserted last can be taken as the 1st element.
- Example : (i) plates placed on the counter of a cafeteria.
 (ii) Disks on a pag.
 (iii) Books on a table one after another
 (iv) postponed decisions.

Terminology :

- PUSH : Inserting the elements at the top.
- POP : Deleting the elements from the top.
- Peep : Extracting the information from the required position of a stack.
- Update : Modifying the old information by new one.
- Top : It is an integer variable which holds the index of the top most element / information in the stack if it is an array implementation of stack. Otherwise, it is a pointer variable which holds the address of the top most element in the stack. Initial value for top = -1 for array implementation and top = NULL for linked representation.

NOTE : Top is also known as TOS (Top of the stack).

SIZE : If it is an array implementation of stack, then the size refers to the maximum no. of element that array can store.

Representing | Implementing the Stack :

- The stack can be represented
 - By an array / static Representation
 - By a link list / Dynamic implementation.

Array implementation of stack:

- Here stack is represented through an array.
- Consider the array named "stack" having size = 7 below.

Stack

0	1	2	3	4	5	6

Here top = -1

Let stack is empty.

[Horizontal Representation of stack]

	6
	5
	4
	3
	2
	1
	0

top = -1
stack is empty.

[vertical Representation]

- If stack is empty, then top = -1. In this situation if any one want to delete any item from the stack, no item will be found, which is called an underflow situation.
- If the stack is full, then top will have the last index of the array.

21	6
08	5
19	4
13	3
67	2
55	1
23	0

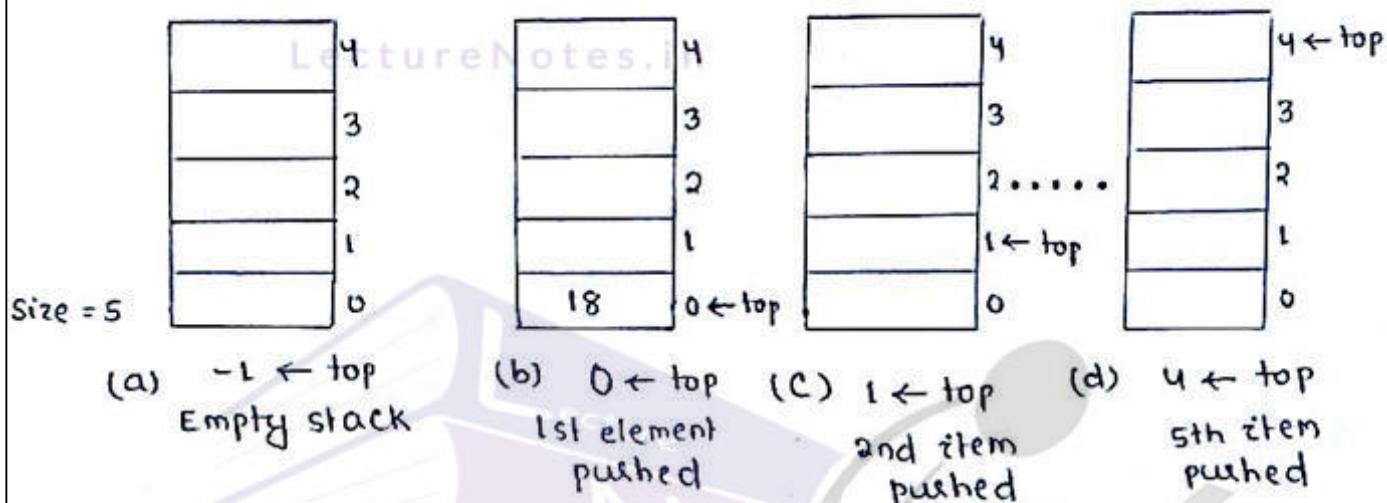
Stack

- Here no new element can be inserted or pushed into the stack. It is called as overflow situation.
- When top = size - 1 occurs we can say it overflow in the stack.

PUSH Operation :

The following steps should be followed at the time of pushing any new element into the stack.

- (i) checking of overflow.
- (ii) Incrementation of the top value i.e. $\text{top}++$.
- (iii) putting the new item/element in the top position i.e. $\text{stack}[\text{top}] = \text{item}$.



→ Here after pushing 5th element into the array, if we want to push another item, then we will encounter the overflow situation because $\text{top} = \text{size}-1$.

Algorithm :

PUSH (STACK , size , top , item)

// STACK : It is a linear array
 // size : Size of the stack .
 // top : It is an integer variable , which holds the index of last element in the stack .

// item : The new element that is to be pushed into the stack

Step 1 : if (top == size - 1)
 (a) print overflow
 otherwise goto step 2 .

Step 2: $\text{top} = \text{top} + 1$

Step 3: $\text{STACK}[\text{top}] = \text{item}$

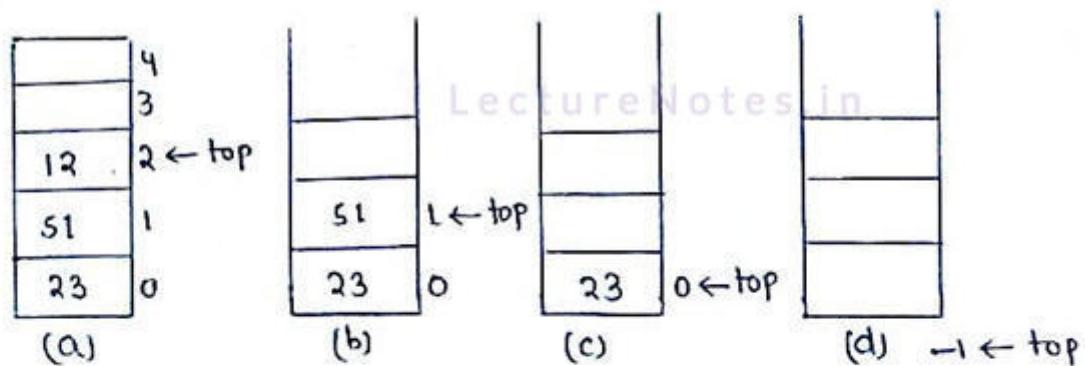
Step 4: exit

C - procedure :

```
void push ( int stack[], int item )
// Here "top" is declared as global variable
{
    if ( top == size - 1 ) .in
        {
            printf( "overflow" );
            exit();
        }
    else
        {
            top = top + 1 ;
            stack[ top ] = item ;
        }
}
```

POP Operation :

- Deleting the topmost element in the stack is called as pop operation.
- Popping the element is nothing but decrementing the top value.



- Consider the above figure , where initially only 3 elements was there in the stack and one by one element has been deleted in the subsequent steps a. b. c.

At last figure (d) indicates that there exist no elements in the stack. so $\text{top} = -1$.

→ At the time of popping operation the following steps is to be followed .

- (i) checking the underflow condition .
- (ii) Decrementing the top value by 1 .

Algorithm :

pop (stack , top . Ditem)

// stack : It is a linear array .

// top : It is an integer variable , which holds the index of last element in the stack .

// Ditem : Before deleting or decrementing the top value the element will be stored in Ditem .

Step 1 : if (top == -1)

 printf "Underflow" otherwise
 goto next step .

Step 2 : Ditem = stack [top]

Step 3 : top = top - 1

Step 4 : print the deleted item is Ditem .

Step 5 : exit .

C - Procedure :

```
void pop ( int stack[1] )
{
    int Ditem ;
    if ( top == -1 )
    {
        printf ("Underflow") ;
        exit () ;
    }
    else
    {
        Ditem = stack [ top ] ;
        top = top - 1 ;
        printf ("The deleted item = %d ", Ditem ) ;
    }
}
```

Peep Operation:

- Extracting the required element from a given position of a stack is called peep operation
- For checking whether the given position from where the element is to be extracted, is valid one or not, formula is $\text{top} - \text{pos} + 1$.
- If $(\text{top} - \text{pos} + 1 < 0)$ then it is a invalid position.

Example :

27	3 \leftarrow top
55	2 pos = 3
23	1 so $\text{top} - \text{pos} + 1$
12	0 = $3 - 3 + 1$ = 1 > 0 so position is valid.

If $\text{pos} = 5$
 $\text{top} = 3$ then $\text{top} - \text{pos} + 1$
 $= 3 - 5 + 1 = -1 < 0$ i.e. position is invalid.

If $\text{pos} = 4$ then $\text{top} - \text{pos} + 1$
 $\text{top} = 3$ $= 3 - 4 + 1 = 0$ is a valid position.

C - Procedure:

```

void peep ( int stack[], int pos )
{
    int item;
    if ( top - pos + 1 < 0 )
        printf (" Invalid position ");
    else
    {
        item = stack [ pos - 1 ];
        printf (" The peeked item = %d ", item );
    }
}

```

Algorithm :

PEEP (stack , pos , item)

// stack : Linear array .

// pos : position from which element is to be peeked .

// item : variable which will hold peeked element .

Step 1 : if (top - pos + 1 < 0)

a) print " invalid position "

LectureNotes.in

b) otherwise goto step 2 .

Step 2 : a) item = stack [pos - 1]

↳ Assign stack [pos - 1] to item .

b) print " the value of peeked item . "

Step 3 : Exit .

Traversal or Displaying operation

C - procedure :

```

void traverse ( int stack[])
{
    int i;
    if ( top == -1 )
        printf (" No element is there in stack");
    else {
        for ( i = top ; i >= 0 ; i-- )
            printf ("%d\n", stack[i]);
    }
}

```

Overflow checking By a Function:

```

int isfull ( int top)
{
    if ( top == size - 1 )
        return 1;
    else
        return 0;
}

```

Underflow checking By a Function:

```

int isempty ( int top)
{
    if ( top == -1 )
        return 1;
    else
        return 0;
}

```

Assignment: write the C-procedure for update operation in a stack.

Sol? void update (int stack[], int pos, int item)

```
{  
    if ( top - pos + 1 < 0 )  
        printf (" Invalid position");
```

```
else  
    stack [ pos - 1 ] = item ;  
}
```

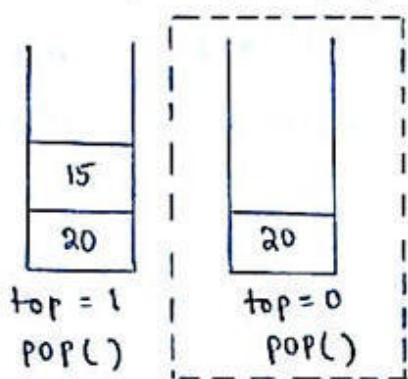
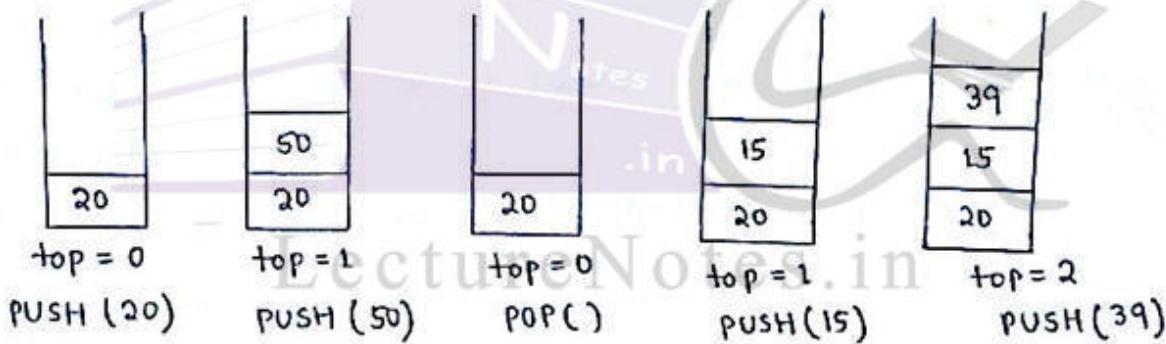
LectureNotes.in

Assignment: Consider the following operations in a stack

1. PUSH (20)
2. PUSH (50)
3. POP()
4. PUSH (15)
5. PUSH (39)
6. POP()
7. POP()

After these operations find out the elements in the stack and if top = -1 initially then what will be the top value?

Sol?



LectureNotes.in
→ Final Stack
top = 0 here.

Assignment: Consider the following display procedure. Find the wrong with the code or the logic in the display operation?

```
void display ( int stack[], int top )  
{
```



Data Structure Using C

Topic:
Application Of Stack

Contributed By:
Mamata Garanayak

Lesson Number : 6Application Of Stack :

A stack is a LIFO structure or it is an appropriate data structure for application in which information must be saved and later retrieved in reverse order.

Some of the application of stack are ;

1) Arithmetic expression

- a. infix notation
- b. prefix notation or polish notation
- c. postfix notation or reverse polish notation.
- d. Evaluation of postfix expression.

2) Conversion of infix to postfix expression.

3) Conversion of infix to prefix expression.

Infix Notation :

- The expression, which are in normal format is known as infix notation.
- In this format type, the operator lies between the operands.
- Syntax : $\boxed{\text{<operand> <operator> <operand>}}$
- Example : $A + B$, $C - D$, $A * B$, G / H etc.
- This is called infix notation because the operator comes in between the operands.

Prefix Notation or Polish Notation :

- In prefix notation or polish notation, the operators comes first followed by the operands.
- Syntax : $\boxed{\text{<operator> <operand> <operand>}}$
- Example : $+ AB$, $- CD$, $* AB$ etc.

Postfix Notation or Reverse Polish Notation:

- In postfix notation or reverse post polish notation, the operands comes first followed by the operator.
- Syntax: $\boxed{\langle \text{operand} \rangle \langle \text{operand} \rangle \langle \text{operator} \rangle}$
- Example: AB+, CD-, GHI etc.

Conversion of Infix \rightarrow postfix:

i) ~~LectureNotes.in~~ parenthesize the expression starting from left to right as per the precedence of operators.

$$\text{Example: } A + B * C$$

$$\Rightarrow A + (B * C)$$

ii) ~~LectureNotes.in~~ The parenthesize expressions will be replaced by a temporary variable.

$$\begin{aligned} &A + (B * C) \\ &\Rightarrow (A + T) \end{aligned}$$

iii) ~~LectureNotes.in~~ The step (i) and (ii) would be repeated till the total expression converted to an expression having two operands and one operator.

iv) Now back track and shift the operator to the right parenthesis.

Example: Convert the expression $A * B - C / D + E$ to postfix expression.

$$\begin{aligned} \text{Ans: } &A * B - C / D + E & [\because T_1 \leftarrow A * B] \\ &= (A * B) - (C / D) + E & T_2 \leftarrow C / D \\ &= (T_1 - T_2) + E & T_3 \leftarrow T_1 - T_2 \\ &= T_3 + E \\ &= T_3 E + \\ &= (T_1 - T_2) E + \\ &= T_1 T_2 - E + \end{aligned}$$

$$\begin{aligned}
 &= T_1 (\underline{C/D}) - E + \\
 &= T_1 CD / - E + \\
 &= (A * B) CD / - E + \\
 &= AB * CD / - E + \quad (\text{Ans})
 \end{aligned}$$

Example : Convert the expression $(A+B) * C/D + E \cap F/G$ into postfix expression.

$$\begin{aligned}
 \text{Soln: } & (A+B) * C/D + E \cap F/G \\
 &= [AB+] * C/D + E \cap F/G \\
 &= T_1 * (C/D) + (E \cap F)/G \\
 &= T_1 * T_2 + T_3/G \\
 &= (T_1 * T_2) + (T_3/G) \\
 &= [T_1 T_2 *] + [T_3 G/] \\
 &= T_1 T_2 * T_3 G/ + \\
 &= AB + (\underline{C/D}) * E \cap F G/ + \\
 &= AB + CD/ * EF \cap G/ + \\
 &= AB + CD/ * EF \cap G/ + \quad (\text{Ans})
 \end{aligned}$$

$$(A+B) * C/D + E \cap F/G$$

$$\begin{aligned}
 &= AB+ * C/D + (E \cap F)/G \\
 &= AB+ * (C/D) + EF \cap /G \\
 &= AB+ * CD/ + (EF \cap /G) \\
 &= AB+ * CD/ + EF \cap G/ \\
 &= AB+ CD/ * + EF \cap G/ \\
 &= AB+ CD/ * EF \cap G/ +
 \end{aligned}$$

(Ans)

Conversion of Infix to prefix :

- i) parenthesize the expression starting from left to right as per the precedence of operators.
- ii) The parenthesized expressions will be replaced by a temporary variable.
- iii) The step *(i)* and *(ii)* would be repeated till total expression is converted to a expression having 2- operands and one operator.
- iv) Now back track and shift the operator to the left parenthesis.

Example : Convert the expression $A * B - C / D + E$ to polish / prefix notation.

Soln? $A * B - C / D + E$

$$= \underbrace{(A * B)}_{\text{Step i)}} - \underbrace{(C / D)}_{\text{Step ii)}} + E$$

$$= [* AB] - [/ CD] + E$$

$$= \underbrace{(- T_1 T_2)}_{\text{Step iii)}} + E$$

$$= [- T_1 T_2] + E$$

$$= \underbrace{(+ T_3 + E)}_{\text{Step iv)}} \bullet$$

$$= + - T_1 T_2 E$$

$$= + - * AB / CDE \quad (\text{Ans})$$

Example : Convert the expression $(A+B) | C * D - E$ into prefix notation.

Soln? $(A+B) | C * D - E$

$$= [+ AB] | C * D - E$$

$$= (T_1 | C) * D - E$$

$$= [/ T_1 C] * D - E$$

$$= T_2 * D - E$$

$$= [* T_2 D] - E$$

$$\begin{aligned}
 &= (T_3 - E) \\
 &= - T_3 E \\
 &= - * T_2 D E \\
 &= - * / T_1 C D E \\
 &= - * / + A B C D E \quad (\text{Ans})
 \end{aligned}$$

Evaluation of postfix expression :

Suppose P is an arithmetic expression written in postfix notation.
 The following algorithm, which uses a stack to hold operands,
 evaluates P .

Algorithm: /* This algorithm finds the value of an arithmetic expression P written in postfix notation */

1. Add a right parenthesis ")" at the end of P . [This acts as a sentinel].
2. Scan P from left to right and repeat step 3 and step 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator \otimes is encountered, then :
 - (a) Remove the two top elements of STACK, where A is the top element and B is the next ^{to} top element.
 - (b) Evaluate $B \otimes A$.
 - (c) place the result of (b) back on STACK.
- [End of if structure]
- [End of step 2 loop]
5. Set VALUE equal to the top element on the STACK.
6. Exit.

O/P Evaluate - postfix (stack , top , P)
// stack : It is a linear array .
// top : Top of the stack , where top = -1 .
// P : The given postfix expression . It is also a character array .

Step 1 : i = 0

Step 2 : while (p[i] != NULL)

(a) if (p[i] == operand)

[Push operand to the stack]

(i) top = top + 1

(ii) stack [top] = p[i]

[End if]

(b) if (p[i] == operator \otimes)

[Pop 2 top most operands]

(i) A = stack [top]

(ii) top = top - 1

(iii) B = stack [top].

[Evaluate B \otimes A and put the result into the stack top].

(iv) stack [top] = B \otimes A

[End if]

(c) i = i + 1

[End while]

Step 3 : [Display the Result]

print the stack [top].

Step 4 : exit .

C - Procedure :

```
void Eval-postfix( int stack[], char p[] )
{
    int i, A, B ;
    for ( i= 0 ; p[i] != '\0' ; i++ )
    {
        if ( isalpha( p[i] ) )
        {
            printf (" Enter the value of %c", p[i] );
            scanf ("%d", &stack[top] );
        }
        else
        {
            switch ( p[i] )
            {
                case '/':
                    A = stack[top];
                    top--;
                    B = stack[top];
                    stack[top] = B/A;
                    break;
                case '*':
                    A = stack[top];
                    top--;
                    B = stack[top];
                    stack[top] = A*B;
                    break;
                case '+':
                    A = stack[top];
                    top--;
                    B = stack[top];
                    stack[top] = A+B ;
                    break;
            }
        }
    }
}
```

Case '-' :

```
A = stack[top] ;  
top-- ;  
B = stack[top] ;  
stack[top] = B - A ;  
break ;
```

Case '^' :

```
A = stack[top] ;  
top-- ;  
B = stack[top] ;  
stack[top] = pow(B, A) ;  
break ;
```

```
}  
}  
printf ("Result = %d \n", stack[top]);  
}
```

Example : Evaluate the postfix notation

P: 5, 6, 2, +, *, 12, 4, /, -

Sol?

Symbol Scanned	STACK
(1) 5	5
(2) 6	5, 6
(3) 2	5, 6, 2
(4) +	5, 8
(5) *	40
(6) 12	40, 12
(7) 4	40, 12, 4
(8) /	40, 3
(9) -	37
(10))	

So VALUE = 37

Assignment: Convert the following infix expression to postfix expression.

$$(i) X \& Y || !(A > B)$$

$$(ii) X \wedge Y \wedge Z$$

$$(iii) a + (b + c * d + e) + f / g$$

$$\text{Soln: } (i) X \& Y || !(A > B)$$

$$= X \& Y || ! [AB >]$$

$$= [XY \&] || [!AB >]$$

$$= XY \& !AB > || (A \underline{\wedge})$$

$$(ii) X \wedge Y \wedge Z$$

$$= (X \wedge Y) \wedge Z$$

$$= [XY \wedge] \wedge Z$$

$$= XY \wedge Z \wedge (A \underline{\wedge})$$

$$(iii) a + (b + c * d + e) + f / g$$

$$= a + (b + [cd *] + e) + [fg /]$$

$$= a + ([bcd *+] + e) + [fg /]$$

$$= a + ([bcd *+e+] + [fg /])$$

$$= [abcd *+e++]+[fg /]$$

$$= abcd *+e++fg / + (A \underline{\wedge})$$

Assignment: Evaluate the postfix notation

$$P: 12, 7, 3, -, /, 2, 1, 5, +, *, +,)$$

Ans:

Symbol	STACK
(1) 12	12
(2) 7	12, 7
(3) 3	12, 7, 3
(4) -	12, 4
(5) /	3

Symbol	STACK
(6) 2	3, 2
(7) 1	3, 2, 1
(8) 5	3, 2, 1, 5
(9) +	3, 2, 6
(10) *	3, 12
(11) +	15
(12))	15

VALUE = 15 //

Assignment : Evaluate the expression

9, 8, 7, *, +

Soln. 9, 8, 7, *, +,)

Symbol	STACK
(1) 9	9
(2) 8	9, 8
(3) 7	9, 8, 7
(4) *	9, 56
(5) +	65
(6))	

VALUE = 65 //

Assignment : Evaluate the following postfix expression .

4.5, 4, 2, ^, +, *, 2, 2, ^, 9, 3, /, *, -

Soln. 4.5, 4, 2, ^, +, *, 2, 2, ^, 9, 3, /, *, -, -)

symbol scanned	operator in stack
(1) 4	4
(2) 5	4, 5
(3) 4	4, 5, 4
(4) 2	4, 5, 4, 2
(5) ^	4, 5, 16
(6) +	4, 21
(7) *	84
(8) 2	84, 2
(9) 2	84, 2, 2
(10) ^	84, 4
(11) 9	84, 4, 9
(12) 3	84, 4, 9, 3
(13) /	84, 4, 3
(14) *	84, 12
(15) -	72
(16))	

VALUE = 72 ,



Data Structure Using C

Topic:

Evaluation Of Prefix Expression

Contributed By:

Mamata Garanayak

Lesson Number : 7

Evaluation of prefix Expression :

Suppose P is an arithmetic expression written in prefix notation. The following algorithm, which uses a stack to hold operands, evaluates P .

Algorithm : /* This algorithm finds the value of an arithmetic expression P written in postfix or prefix notation */

Step 1: Reverse the prefix notation.

Step 2: Add a right parenthesis ")" at the end of P . [This acts as a sentinel.]

Step 3: Scan P from left to right and repeat step 4 to step 5 for each element of P until the sentinel ")" is encountered.

Step 4: If an operand is encountered, put it on stack.

Step 5: If an operator \otimes is encountered, then :

(a) Remove the two top elements of stack, where A is the top element and B is the next-to-top element.

(b) Evaluate $A \otimes B$.

(c) place the result of (b) back on stack.

[End of if structure]

[End of step 2 loop]

Step 6: Set VALUE equal to the top element on the STACK.

Step 7: Exit.

C - procedure :

```
void Eval-prefix ( int stack[], char P[] )
{
    int i, A, B;
    strrev(P);
    for(i=0; P[i] != '\0'; i++)
    {
        if (P[i] >='0' & P[i] <='9')
            stack[i] = P[i] - '0';
        else
            stack[i] = P[i];
    }
}
```

```

if (isalpha(p[i]))
{
    top++;
    printf("Enter the value of %c", p[i]);
    scanf("%d", &stack[top]);
}
else {
    switch(p[i])
    {
        case '/':
            A = stack[top];
            top--;
            B = stack[top];
            stack[top] = A/B;
            break;
        case '*':
            A = stack[top];
            top--;
            B = stack[top];
            stack[top] = A*B;
            break;
        case '+':
            A = stack[top];
            top--;
            B = stack[top];
            stack[top] = A+B;
            break;
        case '-':
            A = stack[top];
            top--;
            B = stack[top];
            stack[top] = A-B;
            break;
        case '^':
            A = stack[top];
            top--;
            B = stack[top];
            stack[top] = pow(A,B);
            break;
    }
}

printf("Result = %d\n", stack[top]);
}

```

Example: Evaluate the prefix notation

P : +, -, *, 2, 2, /, 16, 8, 5

Soln: Reverse of prefix notation is

P : 5, 8, 16, /, 2, 2, *, -, +,)

Symbol Scanned	STACK
(1) 5	5
(2) 8	5, 8
(3) 16	5, 8, 16
(4) /	5, 2
(5) 2	5, 2, 2
(6) 2	5, 2, 2, 2
(7) *	5, 2, 4
(8) -	5, 2
(9) +	7
(10))	.

VALUE = 7.

Conversion of postfix expression to infix expression :

* Apply the algorithm Evaluation of postfix for converting any postfix \rightarrow infix.

Example : AB + C * DEF - / -)

<u>Reading of postfix</u>	<u>stack top</u>	<u>Expression</u>
(1) A	A	A
(2) B	B	B A
(3) +	(A+B)	(A+B)
(4) C	C	C (A+B)

(5)	*	$((A+B)*C)$	$((A+B)*C)$
(6)	D	D	D
(7)	E	E	E D $(A+B)*C$
(8)	F	F	F E D $(A+B)*C$
(9)	-	$(E-F)$	$E-F$ D $(A+B)*C$
(10)	/	$(D/(E-F))$	$(D/(E-F))$ $(A+B)*C$
(11)	-	$((A+B)*C) - (D/(E-F))$	$((A+B)*C) - (D/(E-F))$
(12))		

So the infix expression is :

$$((A+B)*C) - (D/(E-F)) //$$

Example : Convert the following postfix expression to infix expression.

$$AB - DE + F * /$$

<u>Reading of postfix</u>	<u>stack top</u>	<u>Expression</u>
(1) A	A	A
(2) B	B	B A

$$(3) - \quad (A-B) \quad \boxed{(A-B)}$$

$$(4) \quad D \quad D \quad \boxed{D \\ (A-B)}$$

$$(5) \quad E \quad E \quad \boxed{E \\ D \\ (A-B)}$$

$$(6) \quad + \quad (D+E) \quad \boxed{(D+E) \\ (A-B)}$$

$$(7) \quad F \quad F \quad \boxed{F \\ (D+E) \\ (A-B)}$$

$$(8) \quad * \quad ((D+E)*F) \quad \boxed{((D+E)*F) \\ (A-B)}$$

$$(9) \quad / \quad (A-B)/((D+E)*F) \quad \boxed{(A-B)/((D+E)*F)}$$

$$(10) \quad) \quad \boxed{(A-B)/((D+E)*F)}$$

So the infix expression is : $(A-B)/((D+E)*F)$ //

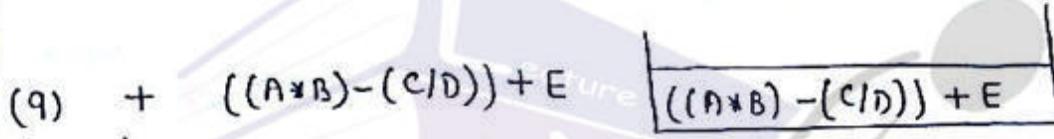
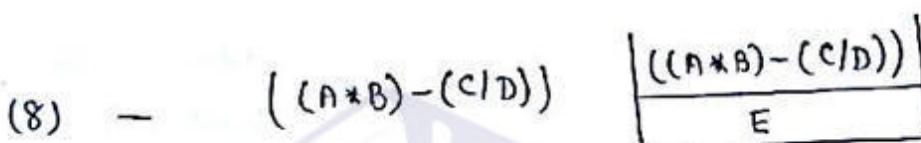
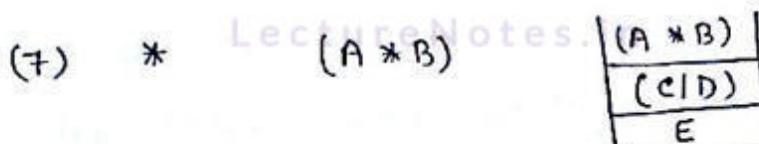
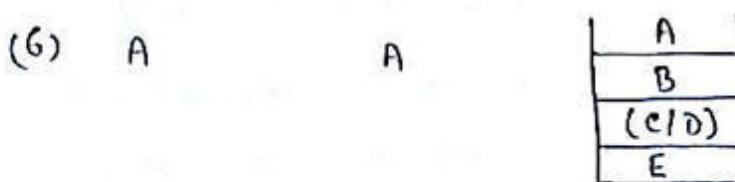
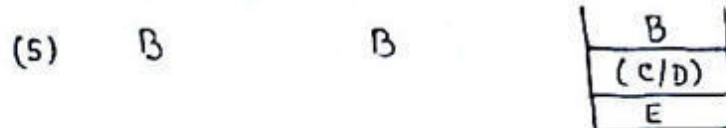
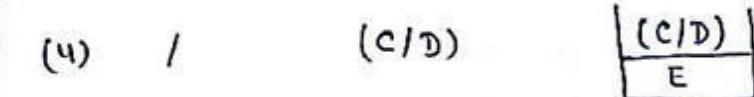
Conversion of Prefix expression to Infix expression :

* Apply the algorithm Evaluation of prefix expression for converting any prefix to infix.

Example : $+ - * A B / C D E$

The reverse of the prefix expression is : $E D C / B A * - +)$

<u>Reading of prefix</u>	<u>Stack top</u>	<u>expression</u>
(1) E	E	\boxed{E}
(2) D	D	$\boxed{D \\ E}$
(3) C	C	$\boxed{C \\ D \\ E}$



(10))
The infix expression is : $((A*B)-(C/D))+E$ //

Conversion of infix to postfix expression :

- An infix expression consists of operand, operators, right parenthesis and left parenthesis.
- In this conversion process, we need to scan one by one characters in the infix expression .
 - if operand will be found , then this one will be added to postfix expression .
 - if operator is found , then it will push into the stack after checking the precedence with stack after the operator already there in stack .
 - if 'C' is found , it will be pushed into the stack .

- if ')' is found, then the element in the stack will be popped out and will be added to postfix expression till get an 'C'.

Algorithm :

Infix - Postfix (Q, P, stack, top)

- // Q : The given infix expression which is a character array.
- // P : gt is a character array, initially blank.
- // stack : gt is a linear array i.e. character type.
- // top : Index variable of integer type.

Step1 : Add ')' at the end of the 'Q' and 'C' to the top of the stack.

Step2 : Scan the infix expression 'Q' from left to right and repeat the steps from 3 → 6 till the stack is empty.

Step3 : if an operand is found add it to 'P'.

Step4 : if '(' is encountered push it into the stack.

Step5 : if operator \otimes is found in Q then

(a) if (precedency of $\otimes \geq$ precedency \otimes)

repeatedly pop the operator \otimes from stack and add it to 'P'.

(b) Add \otimes into the stack.

[End if]

Step6 : if ')' is encountered then :

(a) pop the operators from the stack and add it to 'P' one by one until a '(' is encountered.

(b) Remove the left parenthesis from the stack, don't add it to 'P'.

[End if]

Step7 : print 'P'. Step8 : Exit .

Example : $A + (B * C - (D E \uparrow F) * G) * H$

Soln? $A + (B * C - (D E \uparrow F) * G) * H$

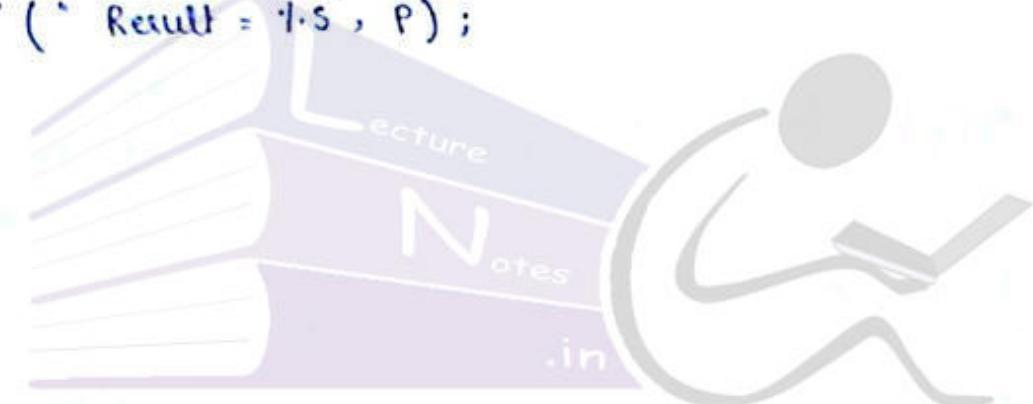
$$= A + (B * C - (D E \uparrow F) * G) * H$$

Symbol Scanned	STACK	Expression (P)
(1) A	(A
(2) +	(+	A
(3) ((+ (A
(4) B	(. + (AB
(5) *	(+ (*	AB
(6) C	(+ (*	ABC
(7) -	(+ (-	ABC *
(8) ((+ (- (ABC *
(9) D	(+ (- (ABC * D
(10) /	(+ (- (/	ABC * D
(11) E	(+ (- (ABC * DE
(12) ↑	(+ (- (↑	ABC * DE
(13) F	(+ (- (↑	ABC * DEF
(14))	(+ (-	ABC * DEF ↑
(15) *	C + C - *	ABC * DEF ↑
(16) G	(+ (- *	ABC * DEF ↑ G
(17))	(+	ABC * DEF ↑ G * -
(18) *	(+ *	ABC * DEF ↑ G * -
(19) H	(+ *	ABC * DEF ↑ G * - H
(20))		ABC * DEF ↑ G * - H * +

C - procedure :

```
void infix-postfix ( char stack[ ] , char in[ ] )
{
    int i=0, j, l;
    top++;
    stack[top] = '(';
    l = strlen ( in );
    in[l] = ')';
    j=0;
    while ( top != -1 )
    {
        if ( isalpha ( in[i] ) )
        {
            p[j] = in[i];
            j++;
        }
        else if ( in[i] == '(' )
        {
            top++;
            stack [top] = in[i];
        }
        else if ( in[i] == ')' )
        {
            while ( stack [top] != '(' )
            {
                p[j] = stack [top];
                top--;
                j++;
            }
            top--;
        }
        else
        {
            if ( precedence ( in[i] ) <= precedence ( stack [top] ) )
```

```
{  
    p[j] = stack[top];  
    j++;  
    top--;  
}  
top++;  
stack[top] = in[i];  
}  
++ ; LectureNotes.in  
}  
p[j] = '\0';  
printf (" Result = %s", p);  
}
```



LectureNotes.in

LectureNotes.in

```

/* program to convert infix expression to postfix */

#include <stdio.h>
#include <conio.h>
char stack[30];
int top = -1;
int precedence (char c) → it will return the precedence of an operator
{
    int a;
    switch (c)
    {
        Case ')':
            a = 0;
            break;
        Case '(':
            a = 1;
            break;
        Case '+':
        Case '-':
            a = 2;
            break;
        Case '/':
        Case '*':
            a = 3;
            break;
        Case '^':
            a = 4;
            break;
    }
    return a;
}

void main()
{
    char in[50], p[50];
    int i = 0, l, j;
    clrscr(); // printf("Enter an infix notation :\n");
               // scanf ("%s", in);
}

```

```

printf("Enter an infix notation :");
scanf("%s", in);
top++;
stack[top] = '('; → pushing '(' to the top of the stack (step 1)
l = strlen(in);
in[l] = ')'; → inserting ')' at the end of infix (step 1).
j = 0;
while (top != -1) → start scanning infix expression till the
                     stack is empty (step 2).
{
    if (isalpha(in[i])) → checking for operand (step 3).
    {
        p[j] = in[i]; → Adding the operand to 'p'.
        j++;
    } → [End if].
    else if (in[i] == '(') → checking for '(' (step 4).
    {
        top++;
        stack[top] = in[i]; → pushing '(' to the stack.
    } [End if]
    else if (in[i] == ')') → checking for ')' (step 5).
    {
        while (stack[top] != '(') → Till get a '(' adding all
                                     (a) the operators from stack (by
                                         popping) to 'p'.
        {
            p[j] = stack[top];
            top--;
            j++;
        }
        top--; → (b) when '(' in stack pop it, don't add it to 'p'
    } → [End if].
    else → for checking the operators (step 6).
    {

```

```

if ( precedence ( in[i] ) <= precedence ( stack [top] ) ) → comparison
{
    p[j] = stack [top];
    top-- ; → (a) Adding the operator from stack to 'p'
}
top ++; → (b) pushing the operator in infix to stack.
stack [top] = in[i];
} → [End if]
i++; → incrementing the index for infix expression to scan
next character.
}
p[i] = '\0';
printf ( " \n The postfix expression : " ); → printing the postfix
printf ( "%s", p );
getch();
}

```

Assignment: Evaluate the following infix expression to postfix.

$$A * (B + C \wedge D) - E \wedge F * (G / H)$$

Soln: $A * (B + C \wedge D) - E \wedge F * (G / H))$

Symbol Scanned	Stack	Expression P
(1) A	(A
(2) *	(*	A
(3) ((* (A
(4) B	(* (B	AB
(5) +	(* (+	AB
(6) C	(* (+ C	ABC
(7) \wedge	(* (+ \wedge	ABC
(8) D	(* (+ \wedge D	ABCD
(9))	(*)	ABCD \wedge +
(10) -	(-	ABCD \wedge + *
(11) E	(-	ABCD \wedge + * E
(12) \wedge	(- \wedge	ABCD \wedge + * E
(13) F	(- \wedge F	ABCD \wedge + * EF
(14) *	(- \wedge F *	ABCD \wedge + * EF \wedge
(15) ((- \wedge F (ABCD \wedge + * EF \wedge
(16) G	(- \wedge F (G	ABCD \wedge + * EF \wedge G
(17) /	(- \wedge F (/	ABCD \wedge + * EF \wedge G /
(18) H	(- \wedge F (/ H	ABCD \wedge + * EF \wedge G H
(19))	(- \wedge	ABCD \wedge + * EF \wedge G H /
(20))		ABCD \wedge + * EF \wedge G H / * -

So now the postfix expression is -

$$ABCD \wedge + * EF \wedge G H / * - //$$



Data Structure Using C

Topic:

Conversion Of Infix To Prefix

Contributed By:

Mamata Garanayak

Conversion of Infix to prefix:

It follows the same process that is there in postfix but with a little bit of modification.

Algorithm:

Infix-prefix (P, Q, STACK, top)

// P : The prefix expression , which is a character array.

// Q : The infix expression

// STACK : It is a linear array i.e character type

// top : index variable of integer type

Step1 : Reverse the infix expression , store it in "RQ".

Step2 : change all '(' to ')' and ')' to '(' in "RQ".

Step3 : Insert ')' to the top of the stack and ')' at the end of "RQ".

Step4 : Scan "RQ" from left to right and repeat the following steps from 5 → 8 till the stack is empty.

Step5 : If an operand is encountered add it to 'P'.

Step6 : If ')' is found push it to the stack .

Step7 : If operator \otimes is encountered then

(a) Compare \otimes with \otimes

[$\otimes \rightarrow$ operator in stack]

 while (precedence of \otimes > precedence of \otimes)

 pop the operator from stack and add it 'P'
 repeatedly .

(b) Add \otimes to the stack .

Step8 : If ')' is encountered

(a) Pop all the operators from the stack and
add to 'P' till ')' is found in stack .

tell 'C' is found in stack.

(b) Remove the 'C' from stack, don't add it to 'P'.

Step 9: Reverse 'P' and print. It is the required prefix expression.

Step 10: Exit.

C - Procedure:

```
void infix-prefix ( char stack[], char in[] )
{
    int i=0, j, l;
    strrev(in);
    for (j=0; in[j] != '\0'; )
    {
        if (in[j] == ')')
        {
            in[j] = '(';
            j++;
        }
        else if (in[j] == '(')
        {
            in[j] = ')';
            j++;
        }
        else
            j++;
    }
    top++;
    stack[top] = '(';
    l = strlen(in);
    in[l] = ')';
    j=0;
    while (top != -1)
    {
```

```

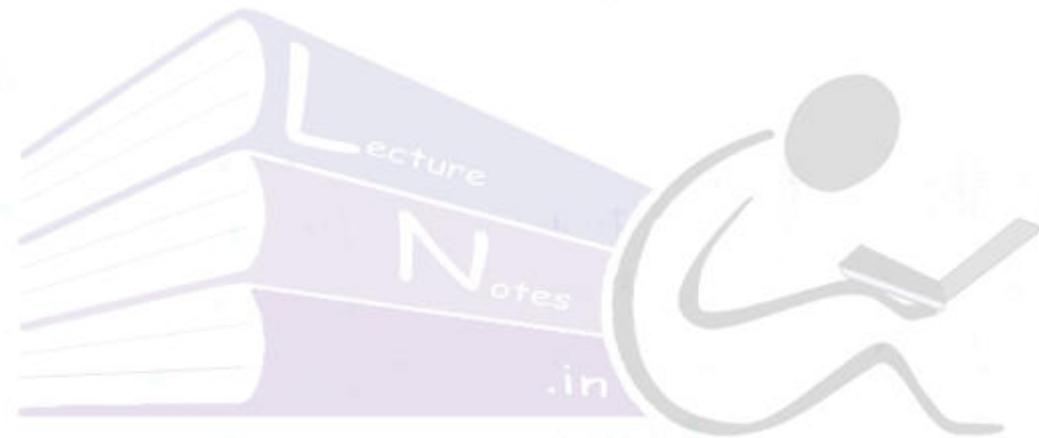
if ( isalpha (in[i]) )
{
    p[j] = in[i];
    j++;
}
else if ( in[i] == '(' )
{
    top++;
    stack[top] = in[i];
}
else if ( in[i] == ')' )
{
    while ( stack[top] != '(' )
    {
        p[i] = stack[top];
        top--;
        j++;
    }
    top--;
}
else
{
    while ( precedence (in[i]) < precedence (stack[top]) )
    {
        p[j] = stack[top];
        j++;
        top--;
    }
    top++;
    stack[top] = in[i];
}
i++;
}
p[j] = '\0';
printf (" Result = %s", strrev(p));
}

```

Now the prefix expression is ;

+ A * - * BC * / D ↑ EFGH ..

LectureNotes.in



LectureNotes.in

LectureNotes.in

Example: Convert the following infix expression into prefix expression.

$$A + (B * C - (D \uparrow E \uparrow F) * G) * H$$

Sol? $A + (B * C - (D \uparrow E \uparrow F) * G) * H$

$$\begin{aligned} &= H *) G *) F \uparrow E | D (- C * B (+ A \\ &= H * (G * (F \uparrow E | D) - C * B) + A) \end{aligned}$$

Symbol scanned	stack	expression p
(1) H	(H
(2) *	(*	H
(3) ((* (H
(4) G	(* (HG
(5) *	(* (*	HG
(6) ((* (*	HG
(7) F	(* (*	HGF
(8) ↑	(* (*	HGF
(9) E	(* (*	HGF
(10) /	(* (*	HGF E ↑
(11) D	(* (*	HGF E ↑ D
(12))	(* (*	HGF E ↑ D /
(13) -	(* (-	HGF E ↑ D / *
(14) C	(* (-	HGF E ↑ D / * C
(15) *	(* (- *	HGF E ↑ D / * C
(16) B	(* (- *	HGF E ↑ D / * C B
(17))	(*	HGF E ↑ D / * C B * -
(18) +	(+	HGF E ↑ D / * C B * - *
(19) A	(+	HGF E ↑ D / * C B * - * A
(20))	(*	HGF E ↑ D / * C B * - * A +

Assignment: Convert the following infix expression to prefix expression.

$$(A+B*C*(M*N\wedge P+T))-G+H$$

Soln. $(A+B*C*(M*N\wedge P+T))-G+H$
 $= H+G- (T+P\wedge N*M(*C*B+A))$
 $= H+G- (T+P\wedge N*M)*C*B+A)$

Symbol Scanned	Stack	Expression P.
(1) H	(H
(2) +	(+	H
(3) G	(+	HG
(4) -	(+ -	HG
(5) ((+ - (HG
(6) T	(+ - (HGT
(7) +	(+ - (+	HGT
(8) P	(+ - (+	HGTP
(9) \wedge	(+ - (+ \wedge	HGTP
(10) N	(+ - (+ \wedge	HGTPN
(11) *	(+ - (+ *	HGTPN \wedge
(12) M	(+ - (+ *	HGTPN \wedge M
(13))	(+ -	HGTPN \wedge M * +
(14) *	(+ - *	HGTPN \wedge M * +
(15) C	(+ - *	HGTPN \wedge M * + C
(16) *	(+ - * *	HGTPN \wedge M * + C
(17) B	(+ - * *	HGTPN \wedge M * + CB
(18) +	(+ - +	HGTPN \wedge M * + CB * *
(19) A	(+ - +	HGTPN \wedge M * + CB * * A
(20))		HGTPN \wedge M * + CB * * A + - +
(21))		HGTPN \wedge M * + CB * * A + - +

Result = + - + A * * BC + * M \wedge N P T G H //

```
/* program to convert infix expression to prefix */
```

```
#include <stdio.h>
#include <conio.h>
char stack[30];
int top = -1;

int precedence (char c)
{
    int a;
    switch (c)
    {
        case ')':
        case '(':
            a = 0;
            break;
        case '+':
            a = 1;
            break;
        case '-':
            a = 2;
            break;
        case '*':
            a = 3;
            break;
        case '/':
            a = 4;
            break;
    }
    return a;
}

void main()
{
    char in[50], p[50];
    int i = 0, l, j;
    clrscr();
    printf (" Enter an infix notation : ");
    scanf ("%s", in);
    strrev (in);
```

```

for ( j=0; in[j]!='\0' ; )
{
    if ( in[j] == ')' )
    {
        in[j] = '(';
        j++;
    }
    else if ( in[j] == '(' )
    {
        in[j] = ')';
        j++;
    }
    else
        j++;
}
top++;
stack[top] = '(';
l = strlen(in);
in[l] = ')';
j=0;
while ( top != -1 )
{
    if ( isalpha(in[i])) LectureNotes.in
    {
        p[i] = in[i];
        j++;
    }
    else if ( in[i] == '(' )
    {
        top++;
        stack[top] = in[i];
    }
    else if ( in[i] == ')' )
    {
        while ( stack[top] != '(' )
        {
    }
}

```

```
p[j] = stack[top];  
top--;  
j++;  
}  
top--;  
}  
else  
{  
    while (precedence(in[i]) < precedence(stack[top]))  
    {  
        p[j] = stack[top];  
        j++;  
        top--;  
    }  
    top++;  
    stack[top] = in[i];  
}  
i++;  
}  
p[j] = '\0';  
printf("\n The prefix expression :");  
printf(" .s", strrev(p));  
getch();  
}
```

QUEUE :

" Queue is a linear data structure in which the deletions can take place only in one end called front and insertions can take place only at other end called rear."

It follows the principle FIFO (First In First Out).

Representation of Queue :

- Array representation
- Link list representation

• Array representation :

→ In an array representation front will be an index variable which will hold the index of the element which will be deleted.

Initial value of front = -1.

→ Here, rear is also an index variable which will hold the index of last element in the queue.

Initial value of rear = -1.

• Link list representation :

→ In a link list representation, front will be a pointer variable which will hold the address of the element or the node i.e. to be deleted.

Initial value of front = NULL.

→ Here, rear is also a pointer variable which will hold the address of last element or the node.

Initial value of rear = NULL.

Terminology :

Front : It is a variable which holds the index of the element that is to be deleted in case of an array representation and it is a pointer variable which holds the address

in case of link list representation.

of the element or node that is to be deleted. initial value of front = -1 in an array representation and initial value of front = NULL in link list representation.

Rear: It is a variable which holds the index of the last element that has been inserted in case of an array representation and it is a pointer variable which holds the address of the last element that has been inserted in case of a link list representation. Initial value of front = rear = -1 in an array representation and initial value of rear = NULL in link list representation.

Size: The size of the queue is the total number of elements that a queue can hold.

Overflow: if ($\text{rear} == \text{size} - 1$) condition is true, then it is called overflow.

Underflow: if ($\text{front} == -1$) condition is true, then it is called underflow.

Example: Suppose a queue has 15-elements but having size 30. But at present there are only 3-elements in the queue shown in figure below.

x	x	...	x	✓	✓	✓		
0	1		11	12	13	14		28	29

x : Elements deleted

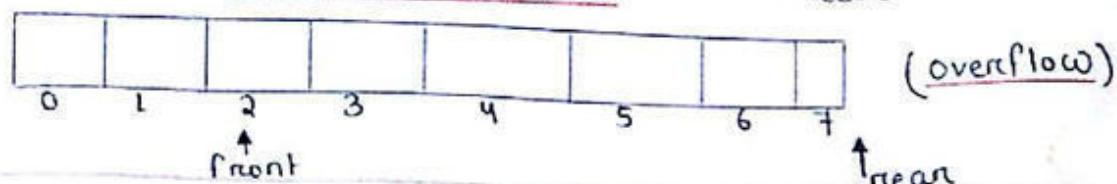
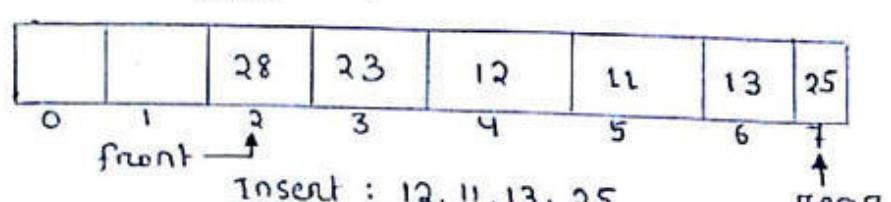
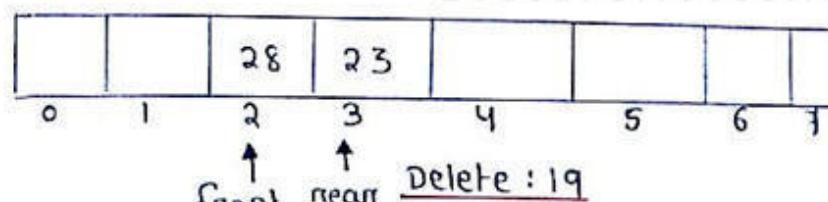
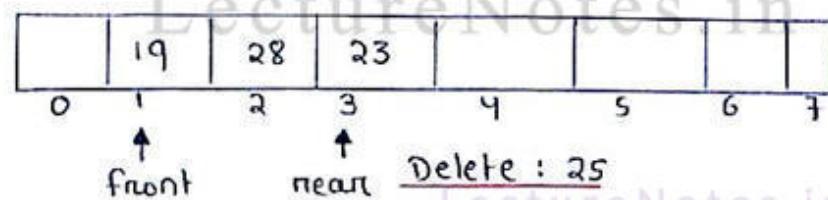
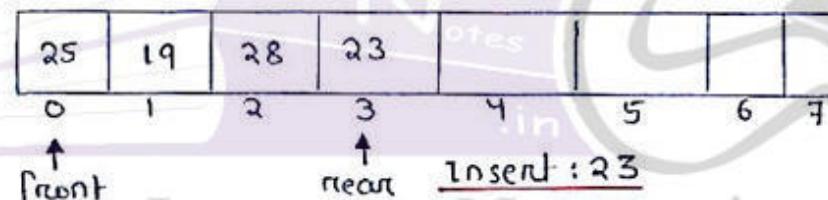
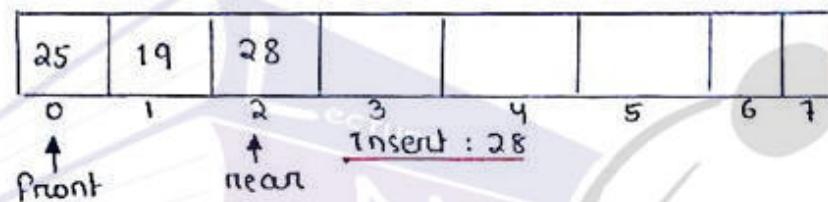
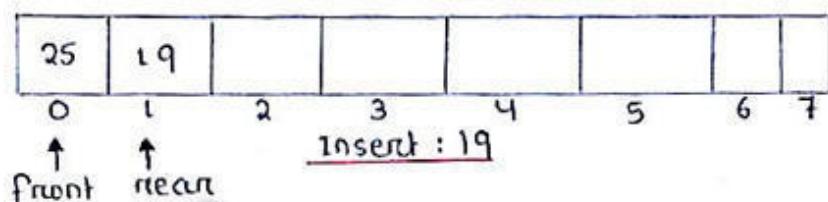
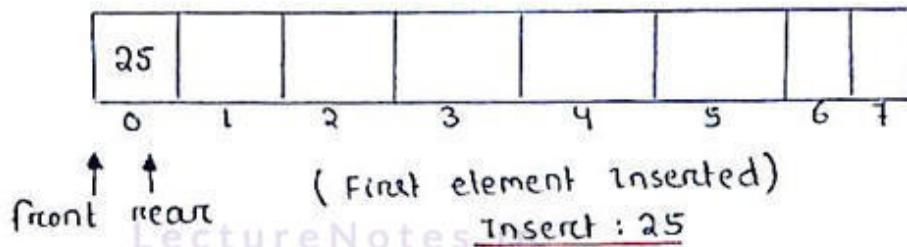
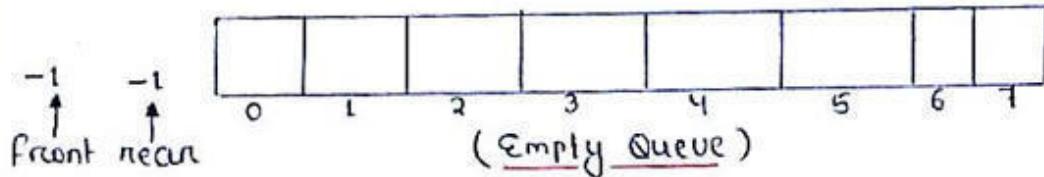
✓ : At present, elements are in position 12, 13, 14 index.

Now front = 12

rear = 14

Some Insertion and Deletion Operations :

Size = 8





Data Structure Using C

Topic:

Various Operations In Queue

Contributed By:

Mamata Garanayak

Lesson Number : 9

Various Operations in Queue :

- Insertion ~~overflay~~
- Deletion
- Traversal / Display
- update

Insertion Operation :

- In the insertion operation, the new element is inserted into the queue in the rear end.
- It follows the following steps.

- i) checking the overflow.
- ii) If queue is empty then front will be set to 0.
- iii) Increment the rear and put the item in rear position.

Algorithm:

PROCEDURE Insertion (Q, item, size, rear, front)

// Q : Linear array or it is a queue represented by linear array.
// item : variable for holding the data element to be inserted.
// size : Maximum no. of elements that Q can hold.
// rear : index variable
// front : index variable

Step 1 : if rear = size - 1
 i) print it is overflow condition.
 ii) otherwise goto step 2.

Step 2 : if rear = -1
 i) front = front + 1 or front = 0
 ii) rear = rear + 1
 iii) Q [rear] = item

Step 3 : Exit

C - Procedure :

```
void insertion ( int Q[], int item )
// rear has been declared as global variable .
// front has been declared as global variable .
// size is a symbolic constant that has been defined .

{
    if ( rear == size - 1 )
        { printf (" overflow condition ");
            exit();
        }
    else
    {
        if ( rear == -1 )
            front = 0;
        rear++;
        Q[rear] = item;
    }
}
```

Deletion : Deletion is nothing but incrementing the front value. It follows the following steps.

- i) underflow checking .
- ii) checking for single element .
- iii) Incrementing front .

Algorithm:

```
PROCEDURE delete ( Q, item, front, rear )
// Q : Queue represented by linear array .
// item : item to be deleted .
// front : index variable .
// rear : index variable .

Step1 : if front = -1
    i) print it is underflow condition .
    ii) otherwise goto step2 .

Step2 : else if front = rear and front ≠ -1
```

- i) print the deleted item is $Q[front]$.
- ii) $front = -1$
- iii) $rear = -1$
- iv) otherwise goto step 3.

Step 3 : print the deleted item is $Q[front]$.

$front = front + 1$

Step 4 : Exit.

C - Procedure :

```

void delete ( int Q[])
{
    int item;
    if ( front == -1 )
    {
        printf (" Underflow");
        return ;
    }
    else if ( front == rear && front != -1 )      ↳ Optional
    {
        item = Q[front];
        front = -1;
        rear = -1;
        printf (" The deleted item is = %d", item);
    }
    else
    {
        item = Q[front];
        front++;
        printf (" The deleted item = %d", item);
    }
}

```

Traversal / Display :

In traversal method or display method, visiting of elements starts from front end to till rear end, by which we can say it follows FIFO principles.

Algorithm :

```
PROCEDURE display ( Q, rear, front )
// Q: It is a linear queue (existing one).
// rear : index variable.
// front : index variable.

Step 1: if front = -1
    i) print underflow condition
    ii) otherwise goto step 2.

Step 2: for i = front to rear
    print Q[i].
    [End for]
    [End if structure]

Step 3: Exit
```

C - Procedure :

```
void display ( int Q[])
// front and rear are declared as global variable.

{
    int i;
    if ( front == -1 )
        printf (" Queue is empty ");
    else
    {
        for ( i = front ; i <= rear ; i++ )
            printf ("%d\t", Q[i]);
    }
}
```

Update : Updating means modifying the existing data in the queue.

Algorithm :

```
PROCEDURE update (Q, rear, front, pos, item)
// Q : linear queue
// rear : index variable
// front : index variable
// pos : position at which item will be updated.
// item : updated item.      step1 : pos = pos-1
step1: if ( pos > = front ) and (pos < = rear )
        i) Q [ pos + 1 ] = item .
        ii) otherwise goto step2.
step2: print invalid position.
```

C - procedure :

```
void update (int Q[], int pos, int item)
{
    pos = pos - 1;
    if ( pos > = front && pos < = rear )
        Q [ pos + 1 ] = item ;
    else
        printf (" invalid position ");
}
```

Assignment: What are the limitations of a queue?

Sol? Consider the queue as given below.

x	x	x	v	v	v	v
0	1	2	3	4	5	6
			↑ front			↑ rear

Here, 3-elements have been deleted. Now 4-elements remain in the queue. rear is pointing to the last element.

LectureNotes.in

Size of the queue = 7

Now if we will go for inserting an element in the queue then overflow will arise, because here rear = size - 1.

But there exist free memory locations i.e. in the index position 0, 1, 2. Even having free memory locations, no element can be inserted in the queue.

This is the limitation having in a linear queue.

Assignment: what are the applications of a queue?

Sol? The applications of queue are :

- <1> Round robin technique for process scheduling is implemented using queues.
- <2> All types of customer services.
- <3> Printing jobs in a printer (shared one) connected to the LAN (Local Area Network).



Data Structure Using C

Topic:
Circular Queue

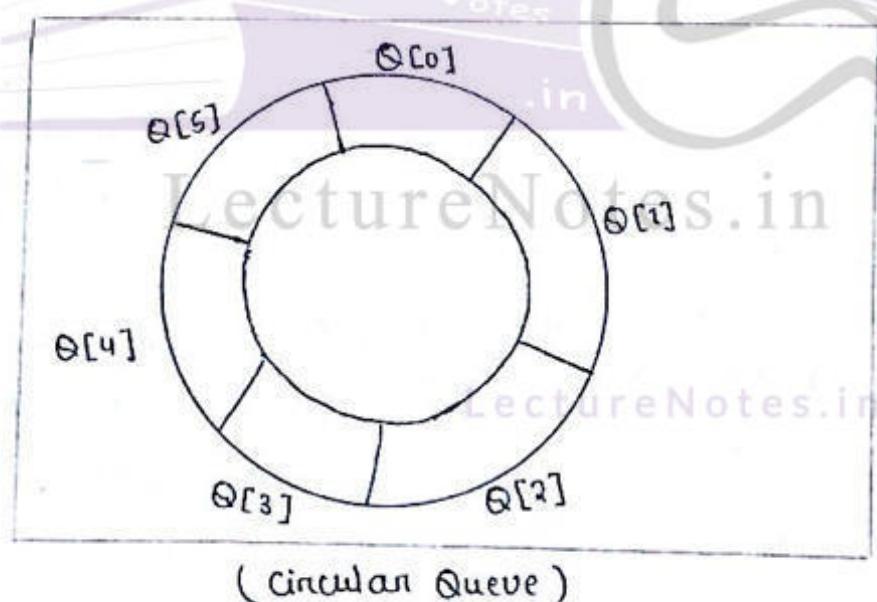
Contributed By:
Mamata Garanayak

Lesson Number: 10

Circular Queue :

" A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location of the queue is full."

- In other words if we have a queue Q of say ' n ' elements, then after inserting an element at last (i.e. in the n -th place) location of the array then the next element will be inserted at very first location (i.e. location with subscript 0) of the array .
- It is possible to insert new elements, if and only if those locations (slots) are empty.
- We can say that a circular queue is one in which the 1st element comes just after the last element . It can be viewed as a mesh or loop of wire , in which the two ends of the wire are connected together . (logically).



Advantages :

- A circular queue overcomes the problem of unutilized space in linear queue implemented as array .

→ Like a linear queue, circular queue is also having front and rear end for deleting and inserting the elements.

→ For getting next rear value, the formula is :

$$\text{rear} = (\text{rear} + 1) \% \text{size}$$

→ For getting next front value, the formula is :

$$\text{front} = (\text{front} + 1) \% \text{size}$$

LectureNotes.in

overflow Condition :

if $\text{front} == (\text{rear} + 1) \% \text{size}$ then overflow condition occur.

Underflow Condition :

if $\text{front} == -1$ then underflow condition occur.

Operations in Circular Queue :

- Insertion
- Deletion
- Traverse (Display)

Insertion :

LectureNotes.in

Inputting the element in the rear end is called insertion. It follows the following steps at the time of insertion.

- i> overflow checking
- ii> checking the 1st element insertion i.e. is it the first element that is going to insert.
- iii> Find the next rear position.
- iv> Input the item or element in the rear position.

Algorithm :

CQ - insert (CQ , front , rear , size , item)
// CQ : It is a linear array , (Circular queue)
// front : Initial value = -1 , it is a index variable .
// rear : Initial value = -1 , it is a index variable .
// size : Size of CQ , i.e. maximum size .
// item : The item that is to be inserted .

Step 1 : If (front == (rear + 1) / size)
 a) print overflow
 b) exit
[End if]

Step 2 : if (rear == -1)
 a) front = 0
[End if]

Step 3 : rear = (rear + 1) % size .

Step 4 : CQ [rear] = item .

Step 5 : exit .

C - Procedure :

void CQ - insert (int CQ[] , int item)

// Here front and rear have been declared as global variable ,
with initial value front = -1 , rear = -1 .

```
{  
    if ( front == ( rear + 1 ) % size )  
    {  
        printf (" overflow " );  
        exit();  
    }  
    else  
    {  
        if ( rear == -1 )  
            front = 0 ;  
        rear = ( rear + 1 ) % size ;  
    }  
}
```

```
CQ [ rear ] = item ;  
}  
}
```

Deletion: Deletion means deleting the element in the front end.
It follows the following steps.

i) Checking for underflow (i.e. front == -1)

ii) Checking if there exist only one element. If it is then
reinitialize front = rear = -1.

iii) Otherwise increment the front value.

i.e. front = (front + 1) % size.

Here before going for next front value, the element that is to be deleted should be stored in a variable.

Algorithm:

PROCEDURE - delete (CQ, rear, front, size, Ditem)

// CQ : circular queue

// rear : index variable

// front : index variable

// size : It is the maximum no. of data element that circular queue (CQ) can hold.

Step 1: [Checking of underflow]

if (front == -1) then print Underflow

else if front == rear then

else store the CQ[front] in another variable Ditem

if (front == rear) then

a) front = rear = -1

b) else front = (front + 1) % size.

print the deleted item is Ditem.

[End of if structure]

Step 2: exit.

C - procedure :

```
void CQ-delete ( int CQ[])
// Here front and rear are taken as global variable with
initial value = -1 .
{
    int Ditem ;
    if ( front == -1 )
    {
        printf (" Underflow ");
        exit();
    }
    else
    {
        Ditem = CQ[front] ;
        if ( front == rear )
            front = rear = -1 ;
        else
            front = (front + 1) % Size ;
        printf (" The deleted item = %d ", Ditem) ;
    }
}
```

Traversal : (Display)

Traversal means visiting each element exactly once.

Algorithm :

LectureNotes.in

```
PROCEDURE Display ( CQ, rear, front, i, size )
```

// CQ : circular queue
// rear : index variable
// front : index variable
// i : loop control variable or counter variable
// size : max^m no. of elements that a queue can hold .

Step1 : if front = -1 then print underflow .

Step2 : if (rear > front)

i. for (i = front ; i != (rear+1) ; i = (i+1)%size)
 print cQ[i] .

Step3 : if (front > rear)

i. for (i = front ; i <= size-1 ; i++)
 print cQ[i] .

ii. for (i = 0 ; i <= rear ; i++)
 print cQ[i] .

Step4 : Exit .

C - Procedure : void display (int cQ[])

```
{  
    int i ;  
    if ( front == -1 )  
        printf (" Underflow : Queue is empty " ) ;  
    if ( rear > front )  
    {  
        for ( i = front ; i != (rear+1) ; i = (i+1)%size )  
            printf (" .dlt ", cQ[i]) ;  
    }  
    if ( front > rear )  
    {  
        for ( i = front ; i <= size-1 ; i++ )  
            printf (" .dlt ", cQ[i]) ;  
        for ( i = 0 ; i <= rear ; i++ )  
            printf (" .dlt ", cQ[i]) ;  
    }  
}
```

Limitation of Linear Queue and circular Queue :

Limitation of linear queue and circular queue is that deletion can takes place only in front end and insertion can take place only in rear end . so we go for Double ended queue or DE - Queue .



Data Structure Using C

Topic:

Double Ended Queue/De Queue

Contributed By:

Mamata Garanayak

Lesson Number: 11

Double Ended Queue / DE Queue :

" Double ended queue or DE-queue is the type of queue where insertion and deletion operations are performed from both ends." That is we can insert elements from the rear end or from the front end, as well as we can delete elements from the rear end or from the front end. Hence it is called as double ended queue or d-queue.

Types of DE - Queue :

There are 2-types of dqueue. These 2-types are due to the restrictions put to perform either the insertions or deletions only at one end. They are ;

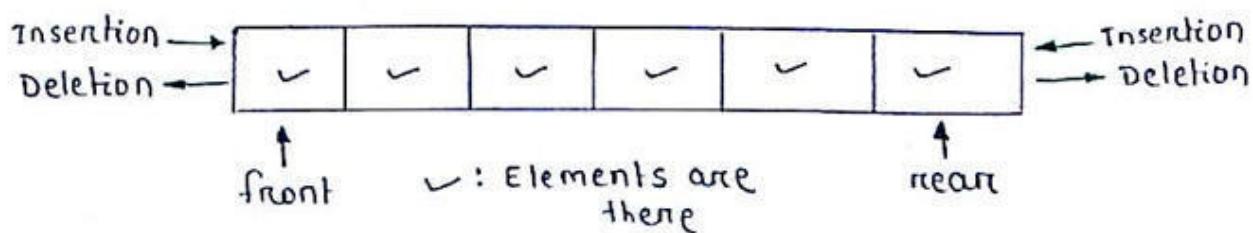
- 1) i/p restricted d-queue .
- 2) o/p restricted d-queue .

1) i/p restricted d-queue :

In input restricted d-queue , insertion can be done only in near end whereas deletion can be done in both near end and front end.

2) Output restricted d-queue :

In output restricted d-queue , insertion can be done in both end (near end and front end) whereas deletion is only possible in front end .



(Double-ended Queue)

Operations on DE-Queue :

Since both insertion and deletion are performed from either end, it is necessary to design algorithm to perform the following four operations. These are;

- (i) Insertion at rear end.
- (ii) Deletion at front end.
- (iii) Deletion at rear end.
- (iv) Insertion at front end.

→ For the IIP restricted d-queue only the operations specified in (i), (ii) & (iii) are valid.

→ For the OIP restricted d-queue only the operations specified in (i), (ii) & (iv) are valid.

Out of these 4-operations (i) & (ii) are already have been discussed in linear queue.

Deletion at rear end :

In this operation, rear should be decremented. The following steps should be followed in deletion at rear end.

- i) Underflow checking (i.e. if $\text{rear} = -1$)
- ii) Checking for only single element.
i.e. if ($\text{front} == \text{rear}$) then there exist only single element, after deleting the element $\text{front} = \text{rear} = -1$.
- iii) Otherwise decrement the rear.
i.e. $\text{rear} = \text{rear} - 1$.

Algorithm :

```
procedure delete (DQ, front, rear, size, Ditem)
```

// DQ : Double ended queue.

// front : index variable

// rear : index variable

// size : Maximum no. of data element that the d-queue can hold
// Ditem : variable which stores the deleted item.

Step1 : [checking of underflow condition]

if rear = -1 then print underflow.

Step2: Else if store the DQ[rear] in the variable Ditem.

if front = rear then

a) front = rear = -1 then

b) else rear = rear - 1 .

print the deleted item is Ditem.

[End if structure]

Step3 : Exit .

C - Procedure :

void Delete-rear (int DQ[])

// Here rear and front have been declared as global variable
with front = rear = -1 .

```
{ int Ditem ;  
if ( rear == -1 )  
    printf (" Underflow " );  
else { Ditem = DQ[rear] ;  
    if ( front == rear )  
        front = rear = -1 ;  
    else  
        rear = rear - 1 ;  
    printf (" The deleted item = %d ", Ditem ) ;  
}
```

Insertion at front end :

In this operation front value decremented and in front position the item or element is inserted.

The following steps is to be followed for inserting an element at the front end.

i) checking the overflow . (i.e. if front == 0) .

ii) if (front == -1) then item can't be inserted at front end.

iii) Decrement the front (i.e. front = front - 1)

iv) put the item in front position .

Algorithm :

PROCEDURE insert-front (DQ, front, rear, item) .

// DQ : Double ended queue .

// front : index variable

// rear : index variable .

// item : Data element to be insert .

Step1: if front = 0 then print overflow condition .

Step2: if (front == -1) then item can't be inserted at front .

Step3: else decrement the front i.e .

front = front - 1 .

put the item in front position i.e .

DQ[front] = item

Step4: Exit .

C - procedure :

```
void insert-front ( int DQ[], int item)
```

```
{
```

```
if ( front == 0 )
```

```
printf ("overflow");
```

```

else if ( front == -1 )
    printf( " item cannot be put in front end" );
else
{
    front = front - 1 ;
    DQ[front] = item ;
}

```

Traversal | Display: Visiting each element exactly once is called as traversal.

Algorithm:

```

procedure display ( DQ, rear, front, i )
// DQ : Double ended queue
// rear : Index variable
// front : index variable .
// i : Counter variable
Step1: if ( front == -1 ) then print queue is empty .
Step2: else for i = front to rear
        print DQ[i] .
Step3: Exit .

```

C- procedure : void dq-display (int DQ[])

```

{
    int i ;
    if ( front == -1 )
        printf( " Queue is empty " );
    else
    {
        for ( i = front ; i <= rear ; i ++ )
            printf( " -d\lt ", DQ[i] );
    }
}

```

Priority Queue :

To represent a priority queue 2-linear arrays (on A 2-D array) are required. First array will keep the data or information and ^{Second} one will keep the priority.

25	23	12	17	08	09	98	99	⇒ Information
1	1	1	2	2	3	3	3	⇒ priority

(priority Queue)

- In priority queue, the informations are processed as per the priority. The lower priority informations will be processed first then the higher priority.
- If we want to put an information '32' having priority '2' then it will be inserted after "08" as per the queue principle so shifting the other data towards right should be done.

25	23	12	17	08	32	09	98	99
1	1	1	2	2	2	3	3	3

↑
(new data inserted)

- This is a time consuming process to shift all the elements towards right. so we can divide all over information to individual priority queue.

Example : Telephone Directory

25	23	12
1	1	1

17	08	32
2	2	2

09	98	99	x
3	3	3	x

(individual priority queue)

Types of priority Queue :

There exist 2 types of priority queue .

- 1) Ascending priority queue
- 2) Descending priority queue .

1) Ascending priority queue :

In ascending priority queue , the informations get stored as per the ascending order of the priority .

Example :

25	23	12	17	08	32	09	98
1	1	1	2	2	2	3	3

2) Descending priority queue :

In descending priority queue , the informations get stored as per the descending order of the priority .

Example :

98	09	32	08	17	12	23	25
3	3	2	2	2	1	1	1



Data Structure Using C

Topic:
Link List

Contributed By:
Mamata Garanayak

Lesson Number: 12

LINK LIST :

Definition: The link list is a linear data structure consists of nodes which are created dynamically.

→ Each nodes of the link list are having 2 parts; one is info part and another is link part

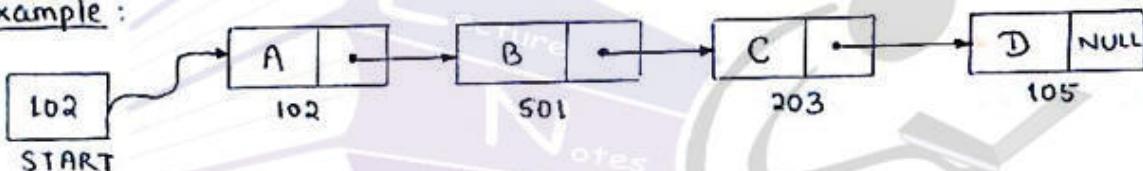
INFO: It contains information on data element

Link: It is a pointer variable which holds the address of the next node.

→ In the link list there exist a special pointer called as START, which holds the address of the first node.

Initial value of start = NULL

Example:



→ There exist 4-types of link list

1. Single link list
2. Double link list
3. Circular link list
4. Header link list

Advantage:

→ In link list, nodes are created dynamically, so the list can grow and shrink during the execution of a program.

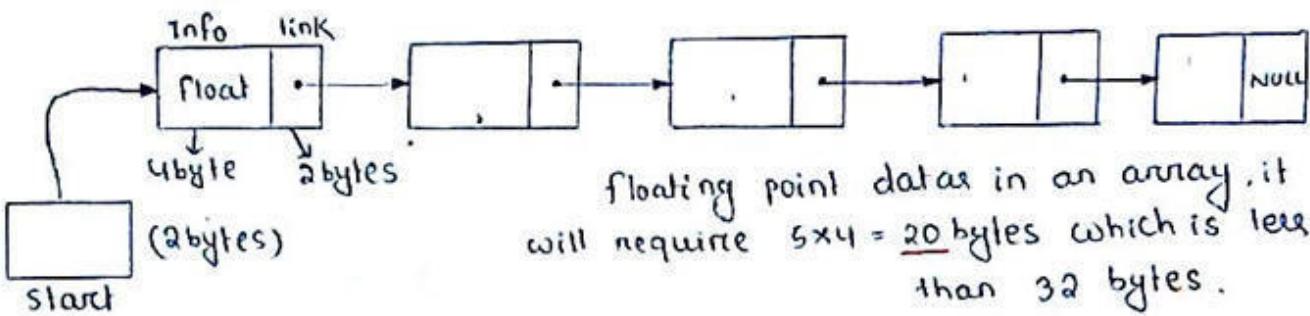
→ Insertion and deletion are easier and efficient.

Disadvantage:

For storing a given information, it require the space for the information and the space for link part as well.

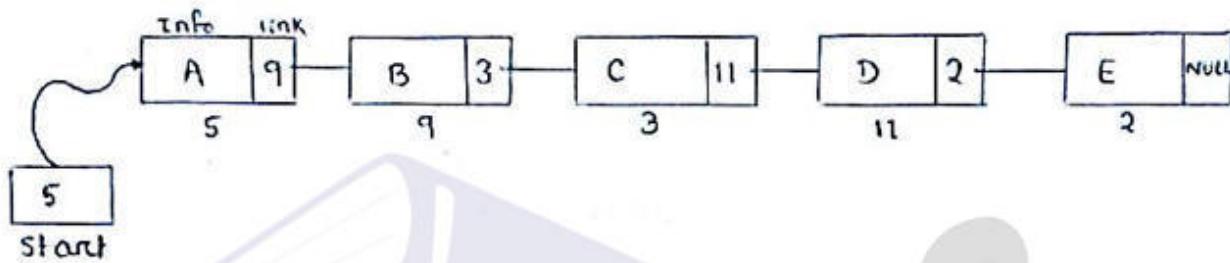
For example if we want to store 5 floating point kind information in a link list, then the total no. of bytes we required

is $5 \times 6 + 2 = 32$ bytes, whereas if we store the same 5 floating point data in an array, it will require $5 \times 4 = 20$ bytes which is less than 32 bytes.



Memory representation of link list :

Consider the link list as follows

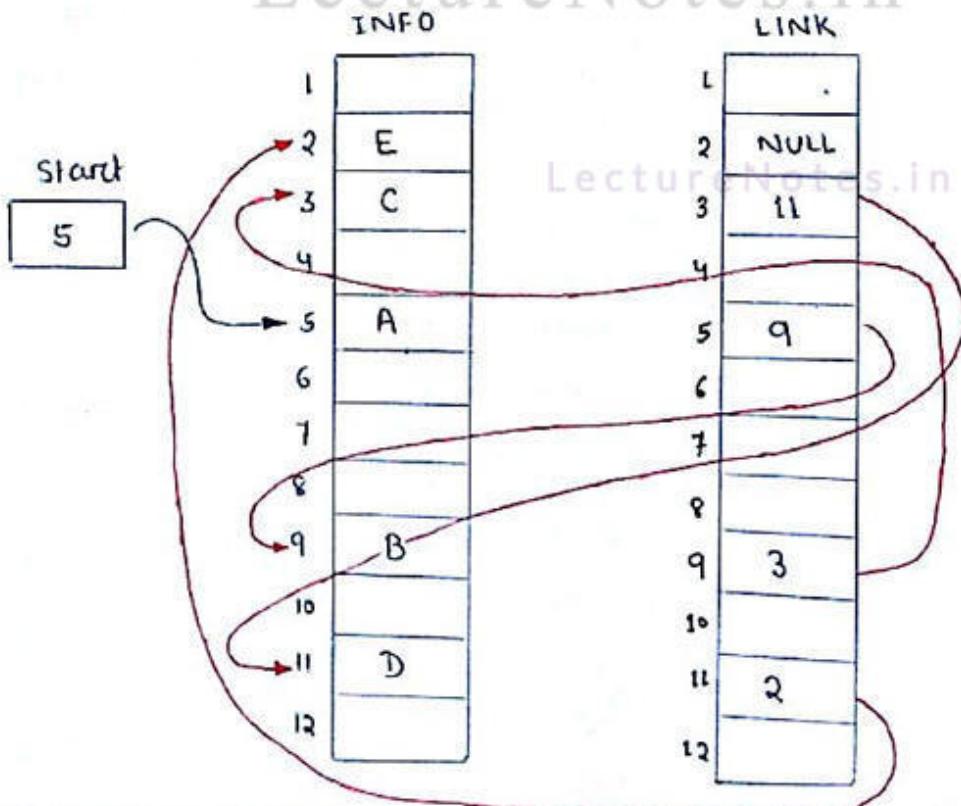


Here 5, 9, 3, 11, 2 are the memory locations of nodes.

→ For representing the above link list memory, 2 linear arrays are required

- i) info
- ii) link

(Memory representation of link list)



Declaration of a node :

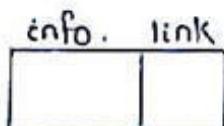
Struct Node

{

int info; → 2 bytes

Struct Node *link; → 2 bytes

}; #include, #include = '40' G



⇒ Node may have more than one information part .

Example : struct Node

Notes.in

{

int age;

char name[20];

float sal;

struct Node *link;

};

age	name	sal	link

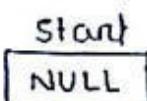
Operations on linked list :

The basic operations on link list are as follows .

1. Creation : Creation of link list .
2. Insertion At the beginning of a link list .
At the end of a link list .
At the specified position .
3. Deletion Beginning of a link list .
End of a link list .
specified position in the list .
4. Traversing : visiting each node exactly once , it may be for displaying , counting , searching .
5. Concatenation : It is the processing of appending (joining) the second list to the end of the list consisting of 'm' nodes . When we concatenate 2-lists , the total list has 'n' nodes , then the concatenated list will be having $(m+n)$ nodes .

[SINGLE LINK LIST]

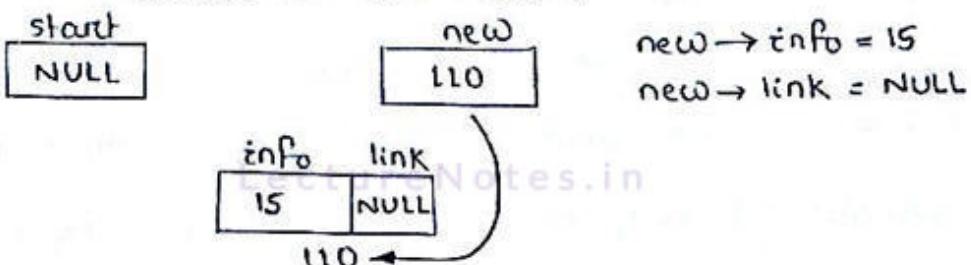
Creation of link list :



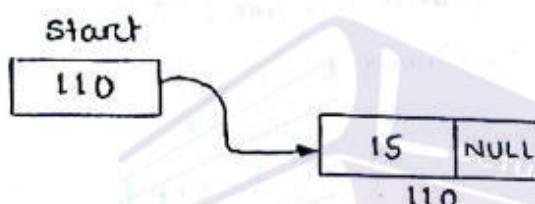
[new is a pointer
start is also a pointer]

Inserting 1st node :

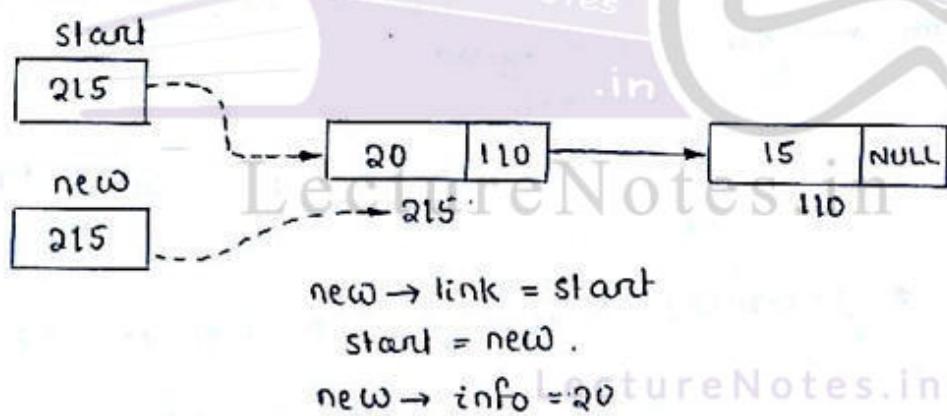
Create a new node .



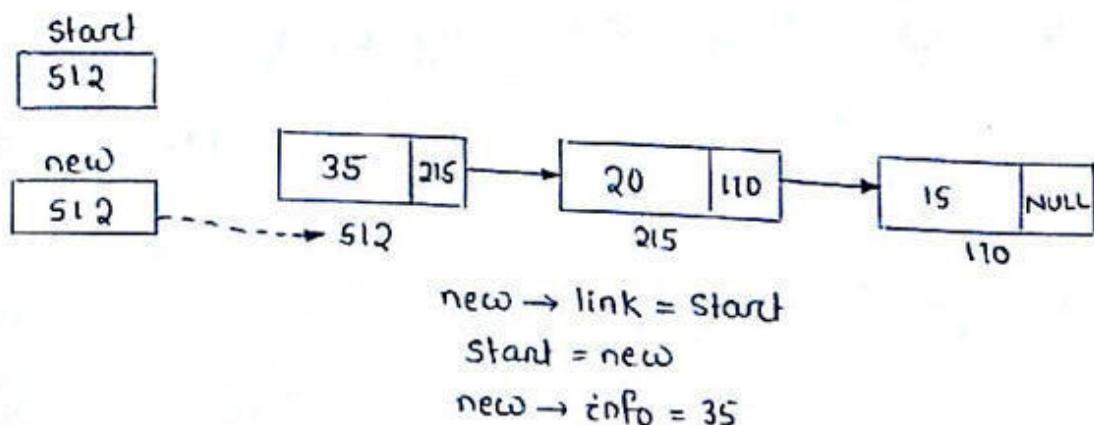
start = new



Insert 2nd node :



Insert 3rd node :



C - Procedure :

```
void create()
{
    int ch;
    do {
        new = (struct node *) malloc (sizeof (struct node));
        if (new == NULL)
            {
                printf ("Memory is not available");
                exit();
            }
        printf ("Enter the item into the node :");
        scanf ("%d", &new->info);
        new->link = NULL;
        new->link = start;
        start = new;
        printf ("Enter your choice\n");
        printf ("1. For Continuing\n");
        printf ("0. For Stop Creating");
        scanf ("%d", &ch);
    } while (ch != 0);
}
```

C - procedure :

```
void create()
{
    int ch;
    do {
        new = (struct node *) malloc (sizeof (struct node));
        if (new == NULL)
        {
            printf ("Memory is not available");
            exit();
        }
        printf ("Enter the item into the node :");
        scanf ("%d", &new->info);
        new->link = NULL;
        new->link = start;
        start = new;
    }
    printf ("Enter your choice\n");
    printf ("1. For Continuing\n");
    printf ("0. For Stop creating");
    scanf ("%d", &ch);
} while (ch != 0);
}
```

```

/* write a program for the creation of node having one
information part */

#include <stdio.h>
#include <conio.h>
#include <alloc.h>

struct node
{
    int info;
    struct node *link;
};

struct node * start = NULL;
struct node * nw;

void main()
{
    int i, n;
    clrscr();
    printf (" Enter the no. of node want to create under link list : ");
    scanf ("%d", &n);

    void create_list(); // Function prototype .
    for( i=0 ; i<n ; i++ )
        create_list(); // Function call .

    getch();
}

void create_list()
{
    nw = ( struct node * ) malloc ( sizeof ( struct node ) );
    if ( nw == NULL )           ↳ // memory allocation to a node
    {
        printf (" Memory is not available ");
        exit();
    }
}

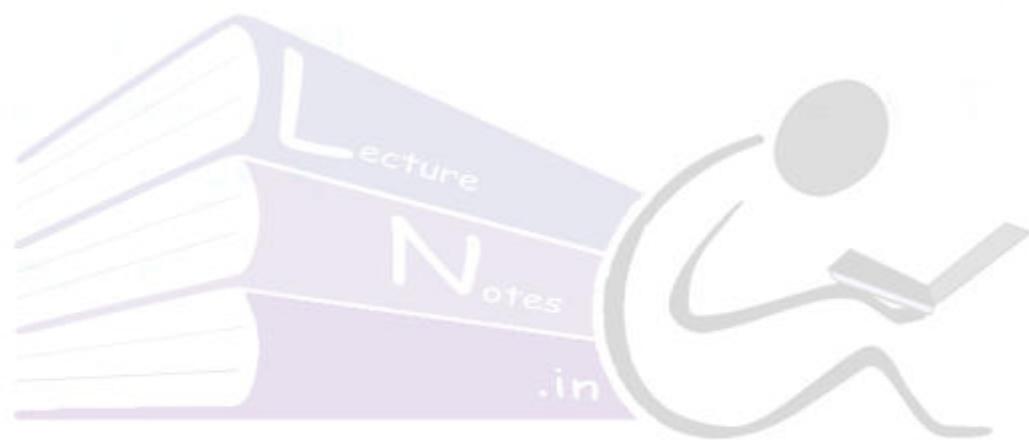
```

```
printf (" Enter the information part : ");
scanf (" %d", &new->info);

new->link = NULL;
new->link = start;
start = new;
}
```

Output :

LectureNotes.in



LectureNotes.in

LectureNotes.in

/* write a program for the creation of node having more than one information part */

```
#include <stdio.h>
#include <Conio.h>
#include <alloc.h>

struct node
{
    int age;
    int sal;
    char name[15];
    struct node *link;
};

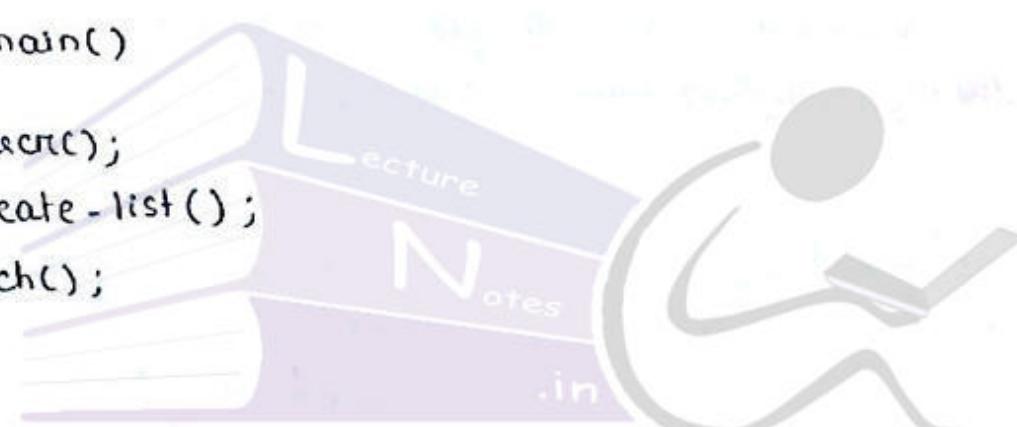
struct node *start = NULL;
struct node *nw;

/* Function Definition */
void create_list()
{
    int ch;
    int c;
    do
    {
        nw = (struct node *) malloc ( sizeof (struct node));
        if (nw == NULL)
        {
            printf ("Memory is not available");
            exit();
        }
        printf (" Enter the information part :\n");
        printf (" enter the age , salary : ");
        scanf ("%d %d", &nw->age, &nw->sal);
        printf (" Enter the name ");
        scanf ("%s", nw->name);
        nw->link = NULL;
    } while (ch != 'N' && ch != 'n');
```

```
new->link = start;
start = new;
++c;
printf ("Enter your choice\n");
printf ("1. For creation of another node\n");
printf ("2. Enter '0' to stop the creation of node\n");
scanf ("%d", &ch);
} while (ch != 0);
printf ("The no. of node that has been created is %d", c);
}

void main()
{
    clear();
    Create-list();
    getch();
}

output :
```



LectureNotes.in



Data Structure Using C

Topic:

Inserting A Node As A 1st Node

Contributed By:

Mamata Garanayak

Lesson Number: 13

Inserting a node as a 1st node;

Inserting a node as a 1st node in the link list follows the following steps.

(i) creating a node .

(ii) checking for overflow .

(iii) putting the information in the information part of the newly created node.

$nw \rightarrow info = item;$

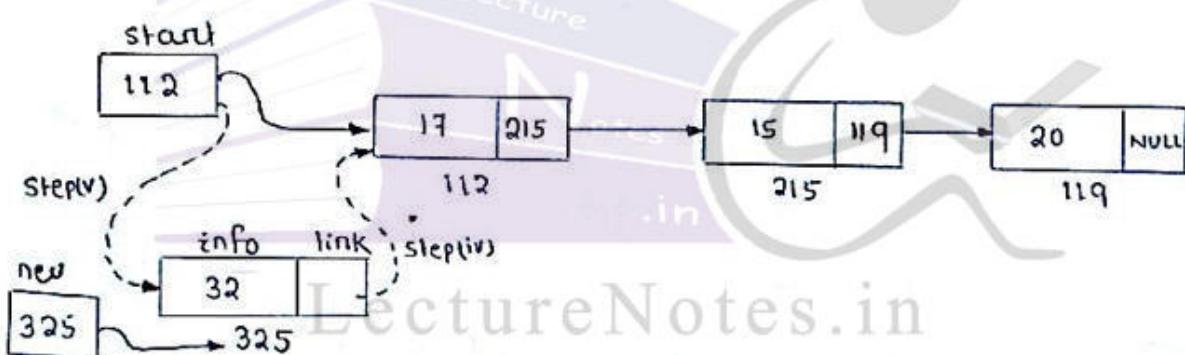
$scanf("1.d", &nw \rightarrow info);$

and assigning NULL to link part of the newly created node.

$nw \rightarrow link = NULL.$

(iv) Assign the address of the 1st node to $nw \rightarrow link$.

(v) Assign the address of "nw" to start pointer i.e. $start = nw$.



Algorithm :

PROCEDURE insert-node-as-a-first-node (List, Link, info, nw, item)

// List : Given single link list

// Link : Link part of the node

// info : Information part of the node

// nw : Newly created node

// item : item to be inserted at 1st position of a link list.

Step 1 : [Checking for overflow]

if ($nw == NULL$)

a. print overflow .

b. Goto step 4

[End if]

Step 2 : $\text{new} \rightarrow \text{info} = \text{item}$.
 $\text{new} \rightarrow \text{link} = \text{NULL}$.

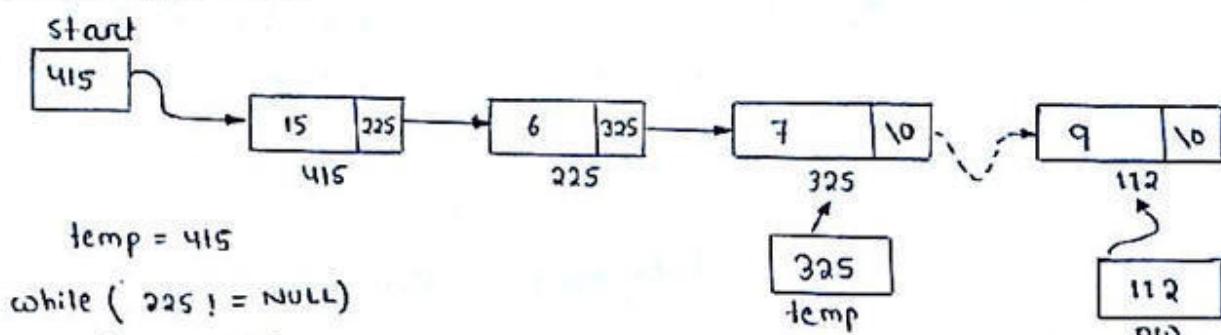
Step 3 : $\text{new} \rightarrow \text{Link} = \text{start}$.
 $\text{start} = \text{new}$.

Step 4 : Exit

C - procedure :

```
void insert_1stnode()
{
    struct node *nw;
    nw = (struct node *) malloc ( sizeof ( struct node ) );
    if ( nw == NULL )
    {
        printf (" overflow ");
        exit ();
    }
    else
    {
        printf (" Enter the item to be inserted as 1st node : " );
        scanf (" %d ", & nw->info );
        nw->link = NULL ;
        nw->link = start ;
        start = nw ;
    }
}
```

Inserting a node as a last node :



```

start = 415
temp = 415
while ( 225 != NULL)
    temp = 225
while ( 325 != NULL)
    temp = 325
while ( 10 != NULL)
    temp → link = 112
  
```

Algorithm :

Insert - Lastnode (Link, List, info, nw, temp, item)

- // Link : Link part of the node .
- // List : Given single link list .
- // info : Information part of the node .
- // nw : newly created node .
- // temp : Temporary pointer variable , which will be used for traversing and at last will hold address of the last node .
- // item : item to be inserted as a last node of the link list .

Step1 : [Checking for overflow]

```

if ( nw == NULL )
    a. print overflow .
    b. goto step 6
  
```

[End if]

Step2 : nw → info = item .
nw → link = NULL .

Step3 : [Checking for empty list]

```

if ( start == NULL )
    a. start = nw
    b. goto step 6
  
```

[End if]

Step4: [Traverse the list till last node]

- a. temp = start
- b. while (temp → link != NULL)
temp = temp → link .

[End of while]

Step 5: [Inserting the new node at last]

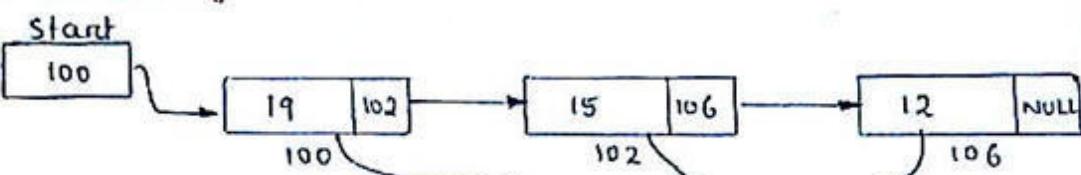
temp → link = nw .

Step6: Exit . LectureNotes.in

C - Procedure :

```
void insertLastnode ()  
{  
    struct node *nw, *temp;  
    nw = (struct node *) malloc ( sizeof ( struct node ) );  
    if ( nw == NULL )  
        printf (" overflow ");  
    else  
    {  
        printf (" Enter the item to be inserted as a last node : ");  
        scanf (" %d ", & nw → info );  
        nw → link = NULL ;  
        if ( start == NULL )  
            start = nw ;  
        else {  
            for ( temp = start ; temp → link != NULL ; temp = temp → link )  
                temp → link = nw ;  
        }  
    }  
}
```

Traversing the link list : (Forward Traversal)



temp = 100
while (100 != NULL)
 print temp → info = 19

temp = temp → link = 102

while (102 != NULL)

print 15

temp = 106

while (106 != NULL)

print 12

temp = NULL

Algorithm :

Traverse_linklist (List, start, temp, info, link)

// List : the given link list .

// start : Holds the address of the 1st node .

// temp : pointer variable used for traversing .

// info : information part of the node .

// link : Link part of the node .

Step1 : [check for empty list]

if (start == NULL)

a. print " List is empty " .

b. goto step 4 .

Step2 : temp = start .

Step3 : while (temp != NULL)

a. print temp → info

b. temp = temp → link

[End while]

Step4 : Exit

C - procedure :

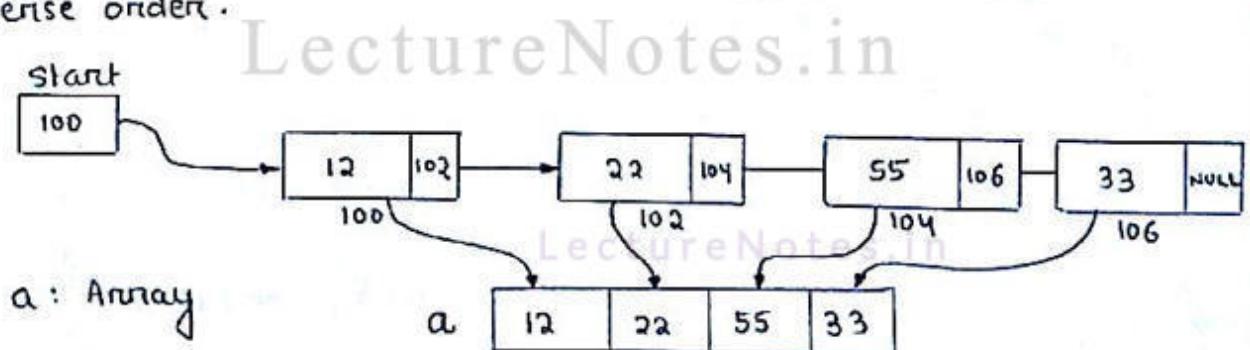
```

void traverse()
{
    struct node * temp;
    if (start == NULL)
        printf("List is empty");
    else
    {
        for( temp = start ; temp != NULL ; temp = temp->link )
            printf("%d\t", temp->info);
    }
}

```

Traversing the link list in reverse Order:

physically we cannot traverse a single link list in reverse order, because after reaching at the last node we could not move back. But we can display the elements in reverse order by storing the values of each node in an array at the time of forward traversing. After storing the elements in the array, display the array in reverse order.



Algorithm :

```

PROCEDURE Reverse-traverse ( List , start , temp , info , link , a )
// List : the given link list
// start : Holds the address of the 1st node
// temp : pointer variable used for traversing
// info : information part of the node .

```

// link : link part of the node .

// a : a is a linear array .

step1 : [check for empty list]

if (start == NULL)

a. print "List is empty"

b. goto step 6

step2 : temp = start and set i = 0 .

step3 : for (temp = start ; temp != NULL ; temp = temp->link)

a. a[i] = temp->info

b. i++

[End for] .

Step4 : i-- .

Step5 : while (i >= 0)

a. print a[i] .

b. i = i - 1

[End while]

Step6 : Exit

LectureNotes.in

C - procedure :

void Reverse_traverse ()

{ int a[30] , i=0 ;

struct node * start ;

struct node * temp ;

for (temp = start ; temp != NULL ; temp = temp->link)

{ a[i] = temp->info ;

i++ ;

}

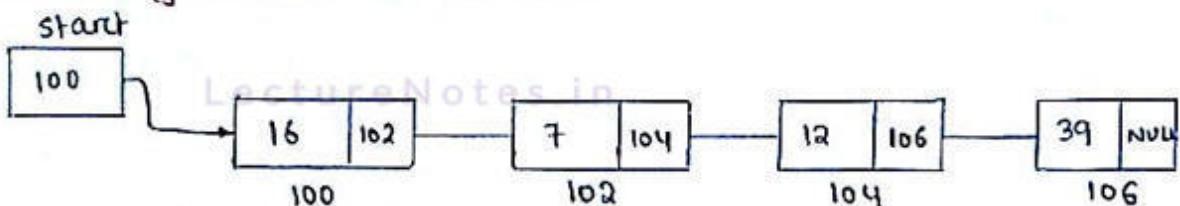
i-- ;

```

while ( i >= 0 )
{
    printf( "%d\n", a[i] );
    i = i - 1;
}

```

Searching an element in a link list :



```

temp = start = 100
while ( 100 != NULL && temp->info != 12 )
    temp = 102
while ( 102 != NULL && temp->info != 12 )
    temp = 104
while ( 104 != NULL && temp->info != 12 ) X
if ( temp->info == 12 )
    print item is found.

```

Algorithm :

```

PROCEDURE Searching ( List, info, link, item, start, temp )
// List : The given link list .
// info : Information part of a node .
// link : Link part of a node .
// item : item that is to be searched .
// start : points to the first node of the link .
// temp : temporary pointer variable .

```

Step1 : [check for empty list]

```

if ( start == NULL )
    a. print List is empty .
    b. exit .

```

Step 2 : temp = start

Step 3 : while (temp != NULL && temp->info != item)

 a. temp = temp->link

[End while]

Step 4 : if (temp->info == item)

 a. print item is found

[End if]

Step 5 : if (temp == NULL)

 a. print item is not found.

[End if]

Step 6 : Exit

C-Procedure :

```
Void searching( )
```

```
{
```

```
    if ( start == NULL )
```

```
        { printf ("list is empty"); }
```

```
        exit(); }
```

```
    temp = start;
```

```
    while ( temp != NULL && temp->info != item )
```

```
        temp = temp->link;
```

```
    if ( temp->info == item )
```

```
        { printf ("item is found"); }
```

```
    }
```

```
    if ( temp == NULL )
```

```
        printf ("item is not found"); }
```

```
}
```

Assignment: Write an algorithm and C-procedure for counting the number of nodes in the link list.

ALGORITHM:

PROCEDURE Count (list, start, link, info, c, temp)

// list : Existing single link list.

// start: pointer variable which holds the address of 1st node.

// link: link part of the node.

// info : variable which holds the information part of a node.

// temp: temporary pointer variable .

// c : Loop variable which will count the no. of nodes .

Step1: [checking of Underflow Condition]

if (start == NULL)

a. print underflow .

b. exit.

Step2: Else

for (temp = start to temp ≠ NULL , temp = temp → link)
 set C = C + 1 .

[End of if]

Step3: print number of node is C .

Step4: Exit

C - PROCEDURE :

void Count ()

{

 int C ;

 struct node * temp ;

 if (start == '10')

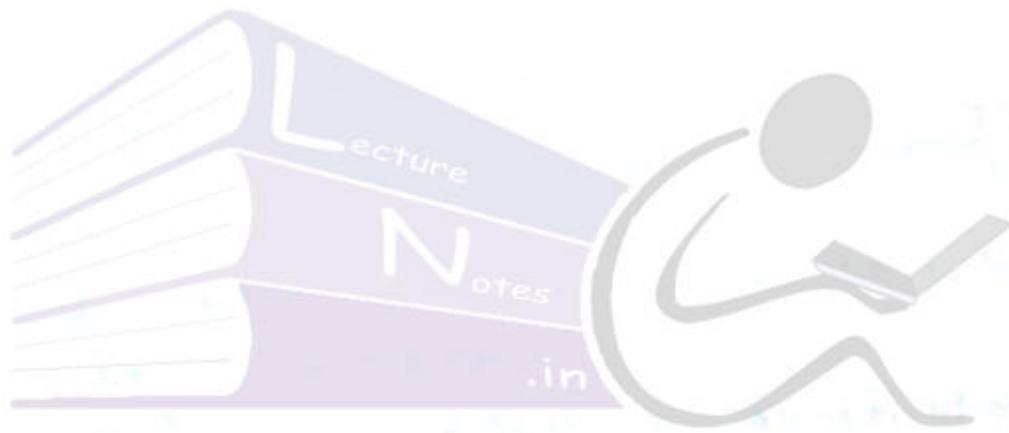
 printf ("Underflow");

 else

{

```
for ( temp = start ; temp != NULL ; temp = temp->link )  
    C++ ;  
}  
printf ("The no. of node present in the list is = %d", C) ;  
}
```

LectureNotes.in



LectureNotes.in

LectureNotes.in



Data Structure Using C

Topic:

Sorting In Single Link List

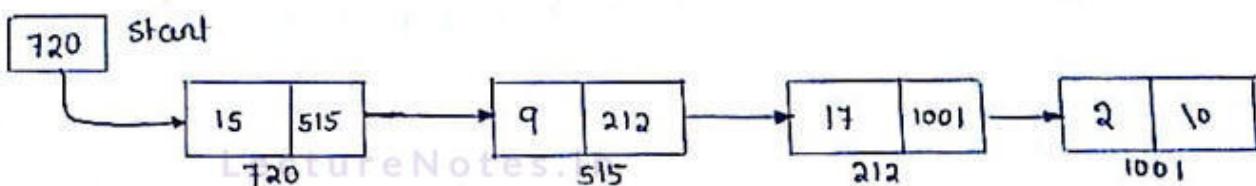
Contributed By:

Mamata Garanayak

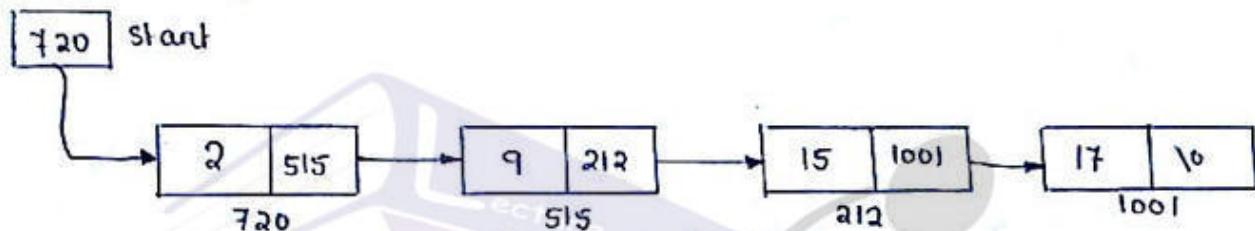
Lesson Number : 14

Sorting in Single link List :

Sorting is a process in which the data elements of a list are arranged either in ascending or descending order sequentially in the node of that list.



After sorting process, list will be ;



Algorithm :

```

PROCEDURE sort ( list, start, link, info, num, temp1, temp2 )
// list : Existing single link list .
// start : pointer variable which holds the address of 1st node in list .
// link : pointer variable which holds the address of next node in list .
// info : Information part of the node .
// num : variable which will be used during swapping the data .
// temp1, temp2 : Temporary pointer variable which will be used
in traversing the list .

```

Start

Step 1 : set temp1 = start .

Step 2 : while temp1 ≠ NULL repeat step(3) to step(4) .

Step 3 : set temp2 = temp1 → link .

Step 4 : A. While temp2 ≠ NULL repeat sub step @ and B .

@ if temp1 → info > temp2 → info then :

- i) set num = temp1 → info .
- ii) set temp1 → info = temp2 → info
- iii) temp2 → info = num .

B. set temp2 = temp2 → link

[End of step(4(n)) while loop]

Step 5 : set temp1 = temp1 → link

[End of step 2 while loop]

Step 6 : Exit .

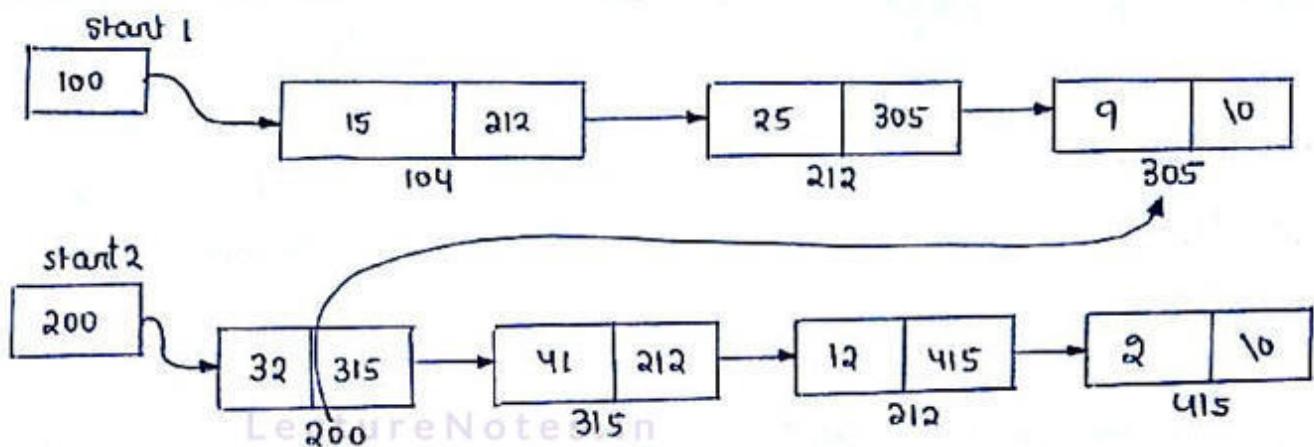
C - procedure :

```

void sort()
{
    // start , link , info are declared as a global variable .
    int num ;
    struct node * temp1 , * temp2 ;
    for( temp1 = start ; temp1 != "NULL" ; temp1 = temp1 → link )
        for( temp2 = temp1 → next ; temp2 != "NULL" ; temp2 = temp2 → link )
            if ( temp1 → info > temp2 → info )
                {
                    num = temp1 → info ;
                    temp1 → info = temp2 → info ;
                    temp2 → info = num ;
                }
}

```

Concatenation of Two Single link List :



Algorithm:

```
PROCEDURE Concatenate ( list1 , list2 , start1 , start2 , temp , link , info )  
// list1 : Existing link list 1 .  
// list2 : Existing link list 2 .  
// start1 : pointer variable , which holds the starting address of  
// the 1st node of list 1 .  
// start2 : pointer variable , which holds the starting address of  
// the 1st node of list 2 .  
// temp : temporary pointer variable used for traversing the list 1 .  
// link : pointer variable , which holds the address of next node .  
// info : Information part of the node .
```

Step1 : a. set $temp = start1$
b. while ($temp \rightarrow link \neq NULL$)
 set $temp = temp \rightarrow link$.
[End while]

Step2 : set $temp \rightarrow link = start2$.

Step3 : set $start2 = NULL$.

Step4 : exit .

C - procedure :

```
Void concatenate( )  
{  
// start1, start2, link, info, list1, list2 have been declared as a  
global variable.  
  
struct node * temp ;  
for ( temp = Start1 ; temp->link != '\0' ; temp = temp->link ) ;  
temp->link = start2 ;  
start2 = "NULL" ;  
}
```

LectureNotes.in

LectureNotes.in



Data Structure Using C

Topic:

Insertion Of A Node At A Given Position

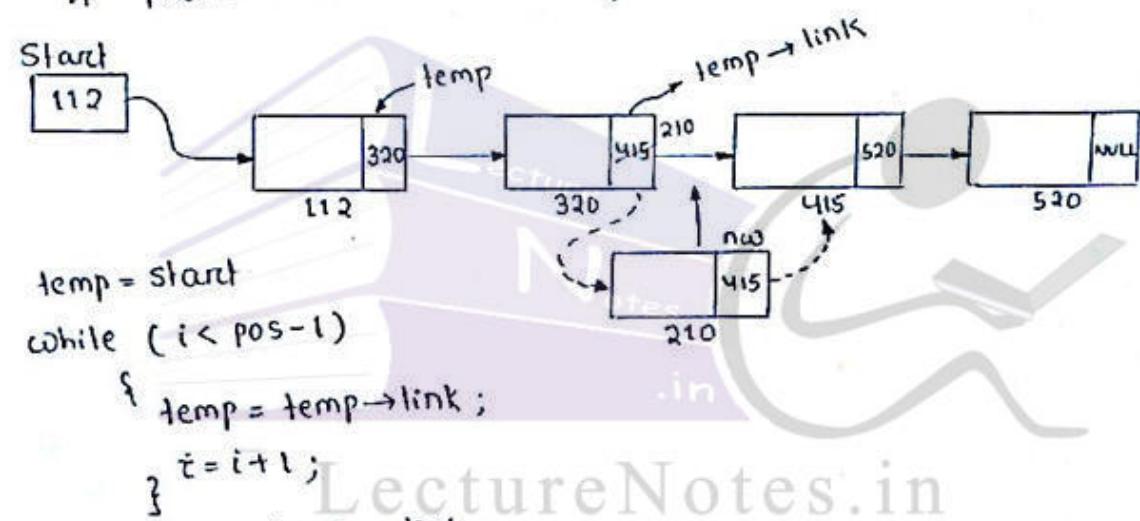
Contributed By:

Mamata Garanayak

Lesson Number: 15

Insertion of node at a given position :

- In this procedure, user will enter the position where the node will be inserted. This process starts with checking the validity of the position.
- If $(\text{position} > \text{Count} + 1)$, then the position is the invalid position where Count = the no. of nodes in the list.
- If position = 3 i.e. before node no: 3
 - If position = 1 i.e. inserting as 1st node.
 - If position = 5 i.e. inserting as last node.



Algorithm :

PROCEDURE Insert-at-any-pos (List, start, link, info, pos, nw, item, temp, c,)

- // List : Given link list .
- // start : pointer variable which holds the address of 1st node .
- // link : link part of the node
- // info : information part of the node .
- // pos : position at which the new node will be inserted .
- // nw : new node
- // temp : temporary pointer variable .
- // c : loop counter , which will count the no. of nodes .

Step 1: [create the new node]

nw = allocate memory.

Step 2: [check the overflow and store the information]

if ($nw == \text{NULL}$)

a. print overflow.

else b. exit.

$nw \rightarrow \text{info} = \text{item}$.

$nw \rightarrow \text{link} = \text{NULL}$.

Step 3: [checking of valid position]

if ($\text{pos} > c+1$)

a. invalid position

b. exit.

Step 4: if ($\text{pos} == 1$)

call the procedure Insert - begining().

Step 5: else if ($\text{pos} == c+1$)

then call the procedure Insert - last().

Step 6: else

set $i = 1$.

$\text{temp} = \text{start}$.

while ($i < \text{pos}-1$)

a. $\text{temp} = \text{temp} \rightarrow \text{link}$

b. $i = i + 1$.

Step 6+1: $nw \rightarrow \text{link} = \text{temp} \rightarrow \text{link}$

$\text{temp} \rightarrow \text{link} = nw$:

Step 7+1: Exit.

C - procedure :

```
void insert-at-pos()
{
    int c, pos;
    struct node *temp;
    printf ("Enter the position at where you want to insert the node:");
    scanf ("%d", &pos);
    /* creation of new node */
    nw = (struct node *) malloc (size of (struct node));
    if (nw == NULL)
        printf ("overflow");
    else
    {
        printf ("Enter the information part of the node :");
        scanf ("%d", &nw->info);
        nw->link = NULL;
    }
    /* codes for checking the validity of position */
    for( temp = start ; temp != '\0' ; temp = temp->link )
        c++;
    if ( pos > c+1 )
    {
        printf ("invalid position");
        exit();
    }
}
```

```

/* code for position one */

if (pos == 1)
{
    new->link = start;
    start = new;
}

/* Codes for LAST position */

else if (pos == c+1)
{
    for( temp = start ; temp->link != '\0' ; temp = temp->link );
    temp->link = new;
}

/* code for OTHER position */

else {
    i = 1;
    temp = start;
    while ( pos <= i < pos-1 )
    {
        temp = temp->link;
        i = i + 1;
    }

    new->link = temp->link;
    temp->link = new;
}
}

```

- ☞ The above procedure is similar to insert a node before a given position.
- ☞ Inserting a node after a given position is similar to earlier one (i.e. insert at any position), but with some little bit of modification. The condition in the "for" loop will be changed to (i < pos).



Data Structure Using C

Topic:
Deletion

Contributed By:
Mamata Garanayak

Lesson Number: 16

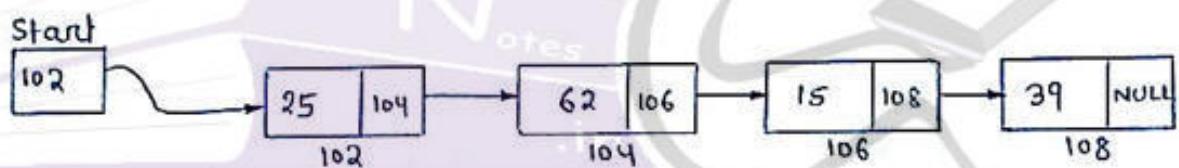
DELETION :

Deleting the node : At the time of each deletion process, the node which is to be deleted, its address is stored in a temporary variable and the link that established with other node is to be cut by assigning NULL to the link part of the node. Lastly the temp get freed by using free function.

Deletion of 1st Node :

In this case, first we need to check the underflow condition. If there exist no node in the list then node cannot be deleted. Otherwise we can go for deletion.

At the time of deleting 1st node, we need to check whether after deleting this node, is the list going to be empty. If it is, then start will be assigned to NULL.



temp = 102
if ($\text{temp} \rightarrow \text{link} = \text{NULL}$) // It is only '1' node is there in the list.
 free (temp);
 start = NULL
else
 start = temp \rightarrow link
 temp \rightarrow link = NULL
 free (temp);

Algorithm :

PROCEDURE Delete (List, temp, start, link)

// List : An existing link list.
// temp : Holds the address of node that is to be deleted.
// start : list pointer variable which has hold the address of 1st node.
// link : link part of the node.

Step1 : [checking for underflow]

if (start == NULL)

a. print underflow.

b. exit.

Step2 : temp = start .

Step3 : if (temp → link == NULL) // if it is only '1' node
is there in the list.

a. free (temp) ;

b. start = NULL .

Otherwise go to next step.

Step4 : start = temp → link

temp → link = NULL .

free (temp) .

Step5 : Exit .

C- Procedure :

Void Delete - 1st - node ()

// Here we are assuming that node has been declared globally
along with start .

```
{     Struct node * temp;
```

```
    if ( start == NULL )
```

```
        { printf ("Underflow"); }
```

```
        exit();
```

```
    temp = start ;
```

```
    if ( temp → link == NULL )
```

```
        { start = NULL ; }
```

```
        free (temp) ;
```

```
    }
```

```
else {
```

```

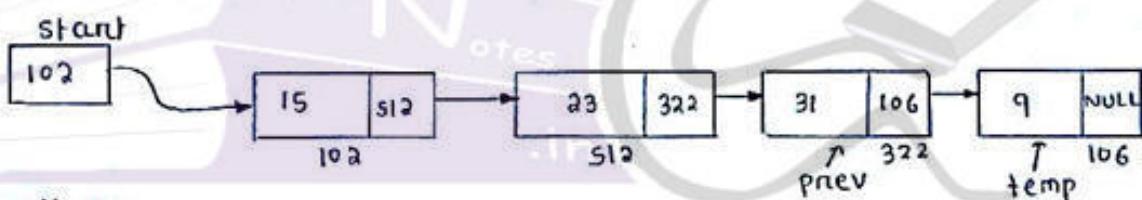
        start = temp->link;
        temp->link = NULL;
        free (temp);
    }
}

```

Deletion of Last Node: Here two pointers are required. One pointer will hold the address of last but 1 node and another will hold the address of the last node, which is to be deleted. As per the figure "prev" and "temp" are 2-pointer variable.

$prev \rightarrow link = NULL$ since prev is going to be last node, and
 $free (temp)$, since this node has been deleted.

For achieving the above objective, we need to traverse the list.



Algorithm:

PROCEDURE Delete-Last (list, link, start, temp, prev)

// list : single link list

// link : pointer variable of node used to hold the address of next node

// start : list pointer variable which has hold the address of 1st node of the list.

// temp : temporary pointer variable used for traversing.

// prev : pointer variable which holds the address of last but 1 node

Step1 : start->node * temp, * prev, [Checking of underflow]

if (start == NULL)

a. printf ("Underflow")

b. exit (0).

Step 2: [checking of existence of single node]

Else if $\text{start} \rightarrow \text{link} = \text{NULL}$ then

a) set $\text{temp} := \text{start}$

b) set $\text{start} := \text{NULL}$

c) Apply function $\text{free}()$ to temp .

Step 3: [if more than one node present]

else a) for $\text{temp} = \text{start}$ to $\text{temp} \rightarrow \text{link} \neq \text{NULL}$, $\text{temp} = \text{temp} \rightarrow \text{link}$

 set $\text{prev} := \text{temp}$.

[End for]

 b) set $\text{prev} \rightarrow \text{link} := \text{NULL}$.

 c) Apply $\text{free}()$ function to temp .

[End of step 2 if structure]

Step 4: Exit.

C - procedure :

```
void deleteLast()
{
    // start, link are global variable.

    struct node *temp, *prev;

    if (start == NULL)
        printf("underflow");

    else if (start → link == '\0')
    {
        temp = start;
        start = '\0';
        free(temp);
    }
    else
    {
```

```
for ( temp = start ; temp->link != '10' ; temp = temp->link )  
{  
    pprev = temp ;  
    pprev->link = NULL ;  
    free (temp) ;  
}  
}
```

LectureNotes.in



Data Structure Using C

Topic:

Deletion Of Node At Any Specific Position

Contributed By:

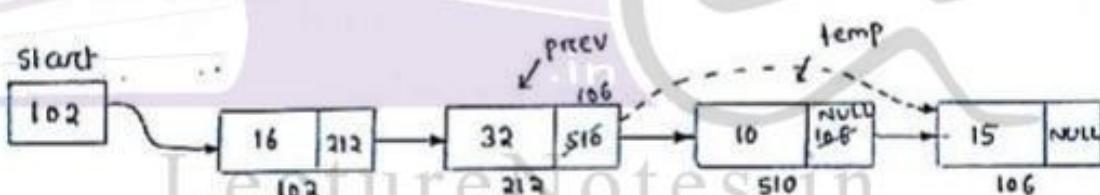
Mamata Garanayak

Lesson Number : 17

Deletion of Node at any specific position :

In this case, user will enter the position of the node that is to be deleted. If ($pos == 1$), then the 1st node will be deleted. If ($pos == C$) where $C = \text{Total no. of node}$, then last node will be deleted. Otherwise, the position is somewhere in between the first and last node.

Before all those position checking, we need to check whether the position entered by the user is a valid one or not. It can be checked by if ($pos > C$) [where $C = \text{No. of node in the list}$] then position is invalid.



$pos = 3$ Node is to be deleted.

```

for (i=1 ; i<pos ; i++)
{
    prev = temp;
    temp = temp->link;
}

Now temp = 510
prev = 212
prev->link = 106
temp->link = NULL

```

i	$i < pos$	Initially $prev = \text{NULL}$ $temp = 102$
1	1 < 3 ✓	$prev = 102$
2	2 < 3 ✓	$prev = 212$ $temp = 510$
3	3 < 3 ✗	

Algorithm :

PROCEDURE Delete-at-pos(List, temp, start, link, prev, c, i, pos)

// List : An existing list

// temp : temporary pointer variable used for traversing .

// start : pointer variable which holds the address of 1st node of the list.

// link: link part of the node.

// prev : pointer variable which holds the address of last but 1 node.

// c : Counter variable, which count the no. of nodes in the list .

// i : Loop variable

// pos : Location from which item is to be deleted .

Step1: [checking for underflow]

if (start == NULL)

a. print underflow

b. exit

Step2: [counting of nodes] Set C=0 .

for (temp = start , temp != NULL , temp = temp → link)

C = C + 1 .

[end of for loop]

Step3: [checking of valid position]

if (pos > C)

a. print invalid position

b. exit .

Step4: [checking for 1st position]

if (pos == 1)

a. if (start → link == NULL) // single node

i. temp = start

ii. start = NULL

iii. free the temp using free() function .

else

1) temp = start

2) start = temp → link

3) $\text{temp} \rightarrow \text{link} = \text{NULL}$

4) Free the temp variable using free function.

Step 5: [Checking for last position]

if ($\text{pos} == \text{c}$)

a. for ($\text{temp} = \text{start}, \text{temp} \rightarrow \text{link} != \text{NULL}, \text{temp} = \text{temp} \rightarrow \text{link}$)

i) $\text{prev} = \text{temp}$

Lecture 23 $\text{oprev} \rightarrow \text{link} = \text{NULL}$

b) Free the temp variable using free function.

Step 6: [Checking for other positions except 1st or last]

a. $\text{prev} = \text{NULL}$

b. $\text{temp} = \text{start}$

for ($i=1, i < \text{pos}, i++$)

1. $\text{prev} = \text{temp}$

2. $\text{temp} = \text{temp} \rightarrow \text{link}$

[End of for loop]

Step 7: $\text{prev} \rightarrow \text{link} = \text{temp} \rightarrow \text{link}$

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

Free the temp variable using free function.

C - Procedure :

void delete-at-pos()

If start and link are global variable

{

int i, c=0, pos;

struct node *prev, *temp;

printf("Enter the position for deletion");

scanf("%d", &pos);

if (start == '\0')

{

```

        printf (" underflow");
        exit();
    }

/* Counting of node */

for (temp = start; temp != '\0'; temp = temp->link)
    c++;

/* codes for checking valid position */

if (pos > c)
{
    printf (" invalid position");
    exit();
}

/* codes if entered position is first */

else if (pos == 1)
{
    if (start->link == '\0') // checking of single node.
    {
        temp = start;
        start = NULL;
        free (temp);
    }
    else
    {
        temp = start;
        start = temp->link;
        temp->link = '\0';
        free (temp);
    }
}

/* Codes if entered position is last */

else if (pos == c)
{
    for (temp = start; temp->link != '\0'; temp = temp->link)
}

```

```
    } prev = temp ;
    prev -> link = '\0' ;
    free (temp) ;
}

/* Codes if entered position is other than first or last */

else {
    prev = '\0' ;
    temp = start ;
    for ( i=1 ; i< pos ; i++ )
    {
        prev = temp ;
        temp = temp -> link ;
    }
    prev -> link = temp -> link ;
    temp -> link = '\0' ;
    free (temp) ;
} // End of else
} // end of Function.
```



Data Structure Using C

Topic:

Implementation Of Stack Using Link List

Contributed By:

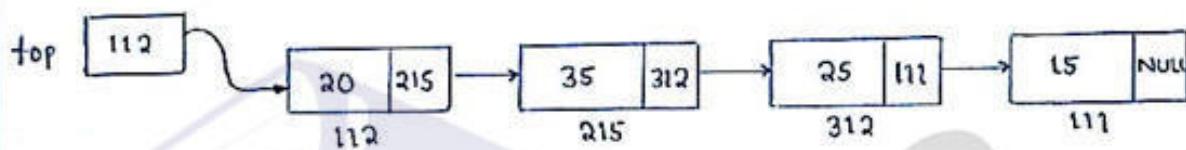
Mamata Garanayak

Implementation of stack Using Link List :

→ Representation of stack by an array:

20	3 ← top
35	2
25	1
15	0

→ Link list representation of stack:

Declaration of stack:

```

struct node
{
    int info;
    struct node * link;
};

struct node * top = '10';
  
```

NOTE:

1. The terminology we used here is top in place of start.
2. Any element we want to push into the stack must be inserted as first node, as top is the list pointer variable.
3. Top should hold the address of the node that we want to insert.

PUSH:

Push operation is exactly similar to insertion of a node at first position.

Algorithm:

```

PROCEDURE PUSH( top, info, link, list, nw )
// top: top is the pointer variable which holds the address
// of 1st node .
  
```

// info : information part of the list
// link : link part of the list
// nw : Newly created node .

Step1 : Create a node and store its address in "nw".

Step2 : [Check for overflow]

if (nw == NULL) in

- a. print overflow .
- b. exit .

[End if]

Step3 : Read info .

Step4 : set nw → link = NULL .

Step5 : [Insertion of node into list]

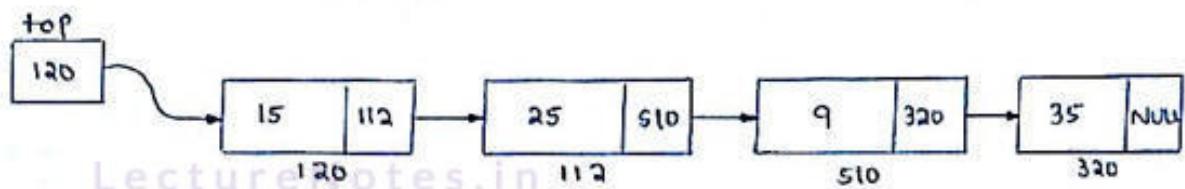
- a. set nw → link = top .
- b. top = nw .

Step6 : Exit .

C - Procedure :

```
void push ()  
{ struct node *nw;  
nw = (struct node *) malloc ( sizeof ( struct node ) );  
if ( nw == '10' )  
{ printf ("overflow");  
exit();  
}  
else {  
printf ("Enter the information part :");  
scanf ("%d", &nw → info);  
nw → link = '10';  
nw → link = top;  
top = nw;  
}
```

POP: We can pop the node whose address is holded by list pointer variable top. So, as top holds the address of 1st node, we can delete 1st node which has been inserted lastly.



Algorithm:

PROCEDURE POP (top, link, temp)

// top: top is the list pointer variable .

// link: Link part of the list .

// temp: Temporary pointer variable .

Step1: Set temp = '10' .

Step2: [check for underflow]

if (top == NULL)

a. print underflow .

b. exit .

Step3: [Single node]

else if (top → link = NULL)

a. temp = top

b. top = '10' .

c. free the temp .

Step4: else a. temp = top .

b. top = temp → link

c. temp → link = '10' .

d. free the temp using free() function .

Step5: Exit

C - Procedure :

```
void pop()
// top, link, temp are global variable
{
    temp = '\0';
    if (top == '\0')
    {
        printf (" underflow");
        exit();
    }
    else if (top->link == '\0') // single Node
    {
        temp = top;
        top = '\0';
        free (temp);
    }
    else
    {
        temp = top;
        top = temp->link;
        temp->link = '\0';
        free (temp);
    }
}
```

DISPLAY :

C - procedure :

```
void display()
// top, link, temp are global variable.
{
    temp = '\0';
    if (top == '\0')
    {
        printf (" List is empty");
        exit();
    }
}
```

```

else
{
    for (temp = top ; temp != '\0' ; temp = temp->link)
        printf ("%d ->", temp->info);
    printf ("NULL");
}
}

```

Algorithm :

PROCEDURE DISPLAY (top, link, temp)
 // top : List pointer variable.
 // link : Link part of the list
 // temp : temporary pointer variable.
 step1 : set temp = NULL.
 step2 : [check for underflow]
 if (top == NULL)
 a. print the list is empty
 b. exit .
 step3 : else
 for(temp = top ; temp != '\0' ; temp = temp->link)
 a. print (temp->info).
 b. print NULL.
 step4 : Exit .



Data Structure Using C

Topic:

Implementation Of Queue By Link List

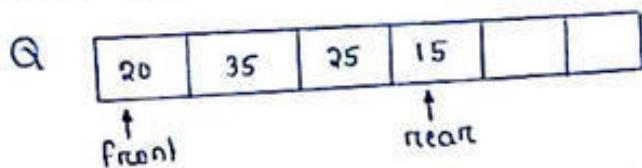
Contributed By:

Mamata Garanayak

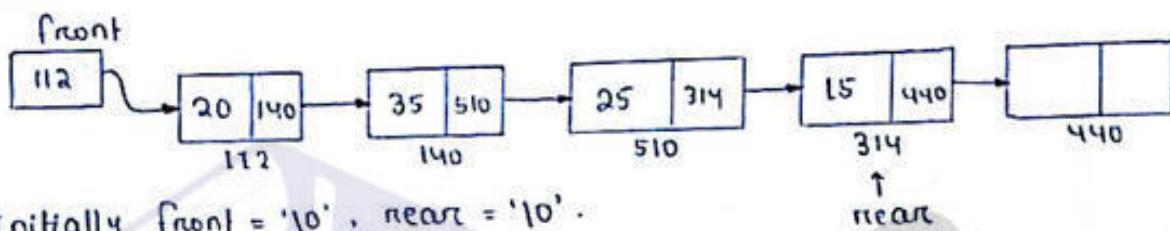
Lesson Number: 19

IMPLEMENTATION OF QUEUE BY LINK LIST:

Array Representation:



Link list Representation:



Declaration:

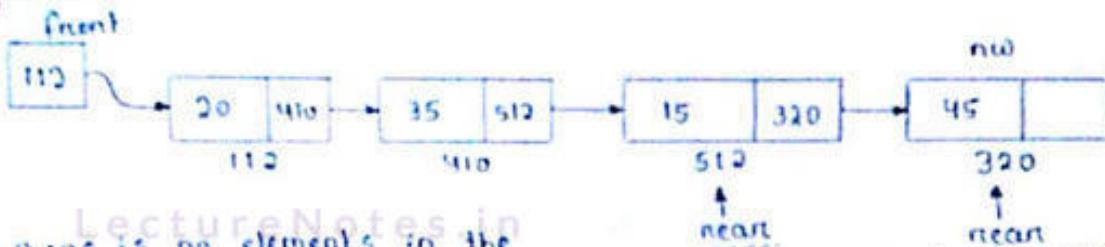
```
struct node  
{  
    int info;  
    struct node *link;  
}; *front = '10', *rear = NULL;
```

NOTE:

- In link list representation of queue, front acts as the list pointer variable and holds the address of node that is first inserted.
- Rear is the pointer variable which holds the address of node which has been inserted at last.
- Here we can insert node / element after last node.
- We can delete the node whose address is holded by front that is the first node.

Insertion :

Algorithm :



if there is no elements in the link list representation of queue
then : i.e

```
if (near == '10') then
    front = nw
    near = nw
```

otherwise . near->link = nw i.e. 320 = 45
 near = nw i.e. near = 320

Algorithm :

PROCEDURE Insert-Q (link, info, front, near, nw)

// link : pointer variable of node which hold the address of next node
// info : variable which hold the information part of each node
// front : List pointer variable which hold the address of 1st node of queue

// near : Pointer variable which hold the address of node that will be inserted last

// nw : pointer variable used during the creation of node dynamically

Step 1 : [check for overflow]

```
if nw == NULL then
    1. print overflow
    2. exit
```

Step 2 : else

1. Read nw->info
2. set nw->link := NULL
3. [steps for emptiness]

```
if( rear == '\0')  
    i> set front := nw  
    ii> set rear := nw  
[ steps if more than one node present ]
```

4) Else
 i> set rear → link := nw .
 ii> set rear = nw .

[End of step 2 (iii) if structure]

[End of step 1 end if structure]

Step 3 : Exit .

C - Procedure :

// rear, front , nw, link have been declared as global variable .

```
void insert ()  
{  
    nw = ( struct node*) malloc ( sizeof ( struct node ));  
  
    if ( nw == '\0')  
    {  
        printf (" overflow ");  
        exit ();  
    }  
  
    else  
    {  
        printf (" Enter the information part ");  
        scanf ("%d", & nw → info );  
        nw → link = '\0' ;  
  
        if ( rear == '\0') // CODES FOR EMPTYNESS  
        {  
            front = nw ;  
            rear = nw ;  
        }  
        else  
        {
```

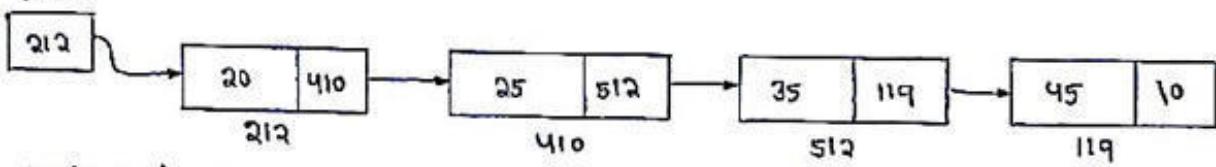
```

    rear → link = nw ;
    rear = nw ;
}
}
}

```

DELETION :

front



single node :

temp = front = 212

free (temp);

front = '10'.

rear = '10'

If more than one node present then ,

temp = front = 212

front = temp → link = 410

temp → link = '10' i.e. assign the link part of 212 to NULL.

free (temp).

Algorithm :

PROCEDURE Delete-Q (link, info, front, rear)

// link : pointer variable of node which hold the address of next node.

// info : variable which hold the information part of each node.

// front : list pointer variable which holds the address of first node of queue .

// rear : pointer variable which hold the address of node that will be inserted last .

step1: [checking for Underflow]

if (front == NULL)

a. print underflow .

b. exit .

Step 2 :

else

1. if (front → link = '\0') // check for single node.
 - i) temp = front.
 - ii) free temp.
 - iii) front = '\0' .
 - iv) rear = '\0' .

[End of if structure]

Step 3 : [Check for more than one node]

else

- i) temp = front .
- ii) front = temp → link
- iii) temp → link = '\0' .
- iv) free temp .

[End of else part]

Step 4 : Exit

C - Procedure :

// front , rear , link are declared as global variable .

void delete - Q ()

{

struct node *temp ;

if (front == '\0')

{
printf (" Underflow ");
exit ();
}

else { if (front → link == '\0') // CODE IF SINGLE NODE IS PRESENT .

temp = front ;

free (temp) ;

front = '\0' ;

} rear = '\0' ;

```
/* CODES IF MORE THAN ONE NODE PRESENT */
```

```
else
```

```
{
```

```
temp = front;
```

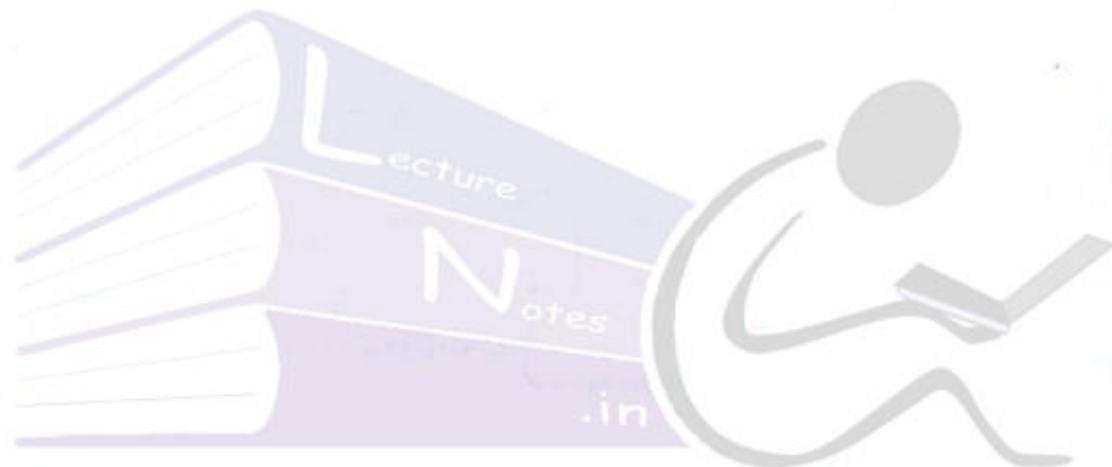
```
front = temp->link;
```

```
temp->link = '\0';
```

```
free (temp);
```

```
}
```

```
}
```



LectureNotes.in

LectureNotes.in



Data Structure Using C

Topic:

Polynomial Representation Through Link List

Contributed By:

Mamata Garanayak

POLYNOMIAL REPRESENTATION THROUGH LINK LIST:

$$\rightarrow 3x^4 + 4x^3 + 2x + 5$$

↓
co-efficient
↓
exponent

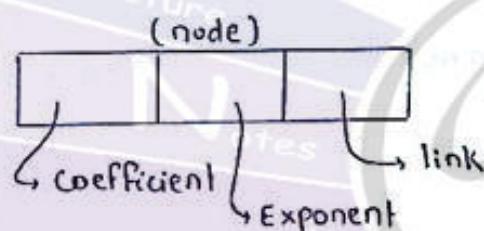
$$\rightarrow 3x^4y^4 + 3x^3y^3 + 3x^2 + 2y + 5$$

↓
co-efficient
↓
exponent of x
↓
exponent of y

Hence each term $3x^4y^4$, $3x^3y^3$, $3x^2$, $2y$, 5 having 3 things; coefficient, exponent of x, and exponent of y.

Polynomial Having One Variable:

Each term treated as a node.

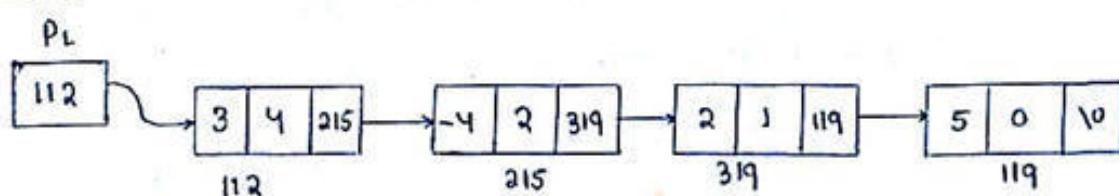


Declaration:

```

struct node
{
    int cof;
    int exp;
    struct node *link;
}
* p1 = '10';
  
```

Example: $3x^4 - 4x^2 + 2x + 5$



Display:

```

for ( temp = p1 ; temp != '\0' ; temp = temp->link )
{
    printf( ".1.d x ^ .1.d", temp->cof, temp->exp )
    if ( temp->cof > 0 )
        printf( "+" );
}

```

Output: $3x^4 + \underline{-}4x^2 + 2x^1 + 5x^0$

To avoid this write the following code.

```

for ( temp = p1 ; temp != '\0' ; temp = temp->link )
{
    if ( temp->cof > 0 )
        printf( "+" );
    printf( ".1.d x ^ .1.d", temp->cof, temp->exp );
}

```

Output: $+ 3x^4 - 4x^2 + 2x^1 + 5x^0$

Algorithm:

```

PROCEDURE display( list, cof, exp, link, temp, p1 )
// list : existing list of terms of polynomial
// cof : Data member of the struct node for holding the coefficient
// part of each term.
// exp : Data member of struct node type for holding the exponent part
// link : next pointer field of each node.
// temp : temporary pointer variable used in traversing .
// p1 : List pointer variable which has hold the address of 1st node.
Step1 : [Checking of underflow]
    if p1 = null then
        print list is having no terms.

```

Step2 : else

- a. for temp = p_1 to temp ≠ null , temp = temp → link
 - i> if temp → cof > 0
then print "+"
 - ii> print temp → cof and temp → exp.
[End for].

[End of step 1 if structure]

Step3 : exit

C - procedure :

```
void display ()  
// p1, link, cof, exp have been declared as global variable .  
{ struct node *temp ;  
if ( p1 == '\0' )  
printf (" List is empty " );  
else  
{ for ( temp = p1 ; temp != '\0' ; temp = temp → link )  
{ if ( temp → cof > 0 )  
printf (" + " );  
printf (" .d x N .d " , temp → cof , temp → exp ) ;  
}  
}  
}
```

Creation of polynomial :

Algorithm :

```
PROCEDURE Create-poly-list ( P , ch , nw , last )
// P : Data member of struct node type for holding the address of
    first node .
// ch : integer variable used to create more than one nodes .
// nw : new node ( pointer variable )
// last : pointer variable .

step1 : [ Create P ]
        ( struct node * ) create-poly-list ( struct node * P )

step2 : Create two pointer variable of structure type ; nw & last

step3 : do
        i) allocate the memory to nw .
        ii) if ( nw == NULL )
            • print cannot allocate the memory space .
            • exit .
        [ End if ]

step4 : Enter the coefficient and exponent of each term .

step5 : assign nw->link = NULL .

step6 : if ( P == NULL )
        a. P = nw .
        b. last = nw .
    [ End if ]

step7 : Else
        a. last->link = nw .
        b. last = nw .
```

Step 8 : print "Enter your choice".
print " 1. For creation of another node"
print " 0. For stop the creation."

Step 9 : while (ch != 0);

Step 10 : return P

Step 11 : Exit

C-procedure :

```
( struct node * ) create-poly-list ( struct node * P )
{
    int ch ;
    struct node * nw, * last ;
    do
    {
        nw = ( struct node * ) malloc ( sizeof ( struct node ) );
        if ( nw == '10' )
        {
            printf (" Enter Malloc cannot allocate memory space " );
            exit ( 0 );
        }
        printf (" Enter the coefficient and exponent of each term " );
        scanf (" %d %d ", & nw->cof , & nw->exp );
        nw->link = '10' ;
        if ( p == '10' )
        {
            p = nw;
            last = nw;
        }
        else
        {
            last->link = nw;
            last = nw;
        }
    }
```

```
printf (" Enter your choice\n");
printf (" Enter 1. For creation of another node\n");
printf (" Enter 0. for stopping creation\n");
scanf ("%d", &ch);
} while (ch!=0);
return (p);
}
```

LectureNotes.in



Data Structure Using C

Topic:

Addition Of Two Polynomial

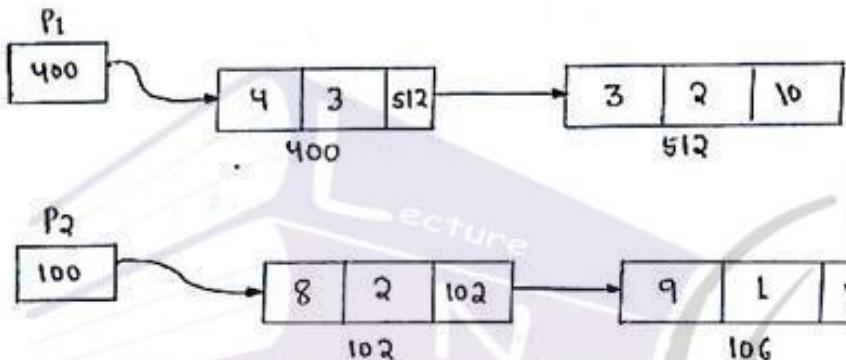
Contributed By:

Mamata Garanayak

Addition of Two Polynomials :

$$P_1 = 4x^3 + 3x^2$$

$$P_2 = 8x^2 + 9x$$



Algorithm:

PROCEDURE addpolynomial ($P_1, P_2, t_1, t_2, P_3, nw, last, cof, exp$)

- // P_1 : First polynomial's pointer variable , which holds the address of the 1st node .
- // P_2 : second polynomial's pointer variable which holds the address of the 1st node .
- // t_1 : temporary pointer variable which hold P_1 .
- // t_2 : temporary pointer variable which stores the value of P_2 .
- // P_3 : new node's pointer variable , which holds the address of the 1st node of the resulting polynomial .

// nw : new node (pointer variable).
// last : temporary pointer variable.
// cof : coefficient part of a polynomial
// exp: Exponent part of a polynomial.

Step1: $P_1 = 4x^3 + 3x^2$ [create two polynomials P_1 & P_2]
 $P_2 = 8x^2 + 9x$

Step 2: $t_1 = P_1$
 $t_2 = P_2$

Step3: while ($\star t_1 \neq '10'$ and $\star t_2 \neq '10'$)
a. if ($t_1 \rightarrow \text{exp} == t_2 \rightarrow \text{exp}$ & $t_1 \rightarrow \text{cof} + t_2 \rightarrow \text{cof} == 0$)
i) $t_1 = t_1 \rightarrow \text{link}$
ii) $t_2 = t_2 \rightarrow \text{link}$
Continue ;
[End if]

Step4: [NODE INSERTION]

nw = allocate memory

Step5: if ($nw == '10'$)
a. print no memory space is available.

b. else

$nw \rightarrow \text{link} = '10'$

if ($P_3 == '10'$)

i) $P_3 = nw$

ii) $last = nw$

[end if]

else

i) $last \rightarrow \text{link} = nw$;

ii) $last = nw$;

Step 6 : if ($t_1 \rightarrow \text{exp} == t_2 \rightarrow \text{exp}$ & $t_1 \rightarrow \text{cof} + t_2 \rightarrow \text{cof} != 0$)
 i) $nw \rightarrow \text{cof} = t_1 \rightarrow \text{cof} + t_2 \rightarrow \text{cof}$
 ii) $nw \rightarrow \text{exp} = t_1 \rightarrow \text{exp}$
 iii) $t_1 = t_1 \rightarrow \text{link}$
 iv) $t_2 = t_2 \rightarrow \text{link}$.

[End if]

Step 7 : else if ($t_1 \rightarrow \text{exp} > t_2 \rightarrow \text{exp}$)
 i) $nw \rightarrow \text{cof} = t_1 \rightarrow \text{cof}$.
 ii) $nw \rightarrow \text{exp} = t_1 \rightarrow \text{exp}$.
 iii) $t_1 = t_1 \rightarrow \text{link}$.

[End if]

Step 8 : else if ($t_1 \rightarrow \text{exp} < t_2 \rightarrow \text{exp}$)
 i) $nw \rightarrow \text{cof} = t_2 \rightarrow \text{cof}$.
 ii) $nw \rightarrow \text{exp} = t_2 \rightarrow \text{exp}$;
 iii) $t_2 = t_2 \rightarrow \text{link}$.

[End if]

[End of while]

Step 9 : while ($t_1 != '10'$)
 i) $nw = \text{Allocate memory}$
 ii) if ($nw == '10'$)
 a. print overflow.
 b. else
 $nw \rightarrow \text{link} = '10'$
 $nw \rightarrow \text{cof} = t_1 \rightarrow \text{cof}$;
 $nw \rightarrow \text{exp} = t_1 \rightarrow \text{exp}$.
 $\text{last} \rightarrow \text{link} = nw$.
 $\text{last} = nw$.
 $t_1 = t_1 \rightarrow \text{link}$.

[End while]

Step 10 : while ($t_2 \neq '10'$)
 i) $nw = \text{Allocate memory}$
 ii) if ($nw == '10'$)
 a. print overflow
 b. else

LectureNotes.in
 $nw \rightarrow \text{link} = '10'$.
 $nw \rightarrow \text{cof} = t_2 \rightarrow \text{cof}$.
 $nw \rightarrow \text{exp} = t_2 \rightarrow \text{exp}$.
 $\text{last} \rightarrow \text{link} = nw$.
 $\text{last} = nw$.
 $t_2 = t_2 \rightarrow \text{link}$.

[End while]

lecture

Notes

step 11 : Exit

C-procedure :

```
void add-poly (struct node *P1, struct node *P2)
{
    // P3, exp, cof, link have been declared as global variable.
    struct node *l1, *t2, *nw, *last;
    l1 = P1;
    t2 = P2;
    while (l1 != '10' && t2 != '10')
    {
        if (l1->exp == t2->exp && l1->cof + t2->cof == 0)
        {
            l1 = l1->link;
            t2 = t2->link;
            continue;
        }
        /* NODE INSERTION */
        nw = (struct node *) malloc (sizeof (struct node));
        nw->link = l1->link;
        nw->cof = l1->cof;
        nw->exp = l1->exp;
        l1->link = nw;
    }
}
```

```

if (nw == '\0')
    printf (" memory space can't allocated");
else {
    nw->link = '\0';
    if (P3 == '\0')
        {
            P3 = nw;
            last = nw;
        }
    else {
        last->link = nw;
        last = nw;
    }
}
if (t1->exp == t2->exp && t1->cof + t2->cof != 0)
{
    nw->cof = t1->cof + t2->cof;
    nw->exp = t1->exp;
    t1 = t1->link;
    t2 = t2->link;
}
else if (t1->exp > t2->exp)
{
    nw->cof = t1->cof;
    nw->exp = t1->exp;
    t1 = t1->link;
}
else if (t1->exp < t2->exp)
{
    nw->cof = t2->cof;
    nw->exp = t2->exp;
    t2 = t2->link;
}
}
    
```

? A end of while

```

while (t1 != '\0')
{
    nw = (struct node *) malloc(sizeof(struct node));
    if (nw == '\0')
        printf ("Overflow");
    else {
        nw->link = '\0';
        nw->cof = t1->cof;
        nw->exp = t1->exp;
        last->link = nw;
        last = nw;
        t1 = t1->link;
    }
}

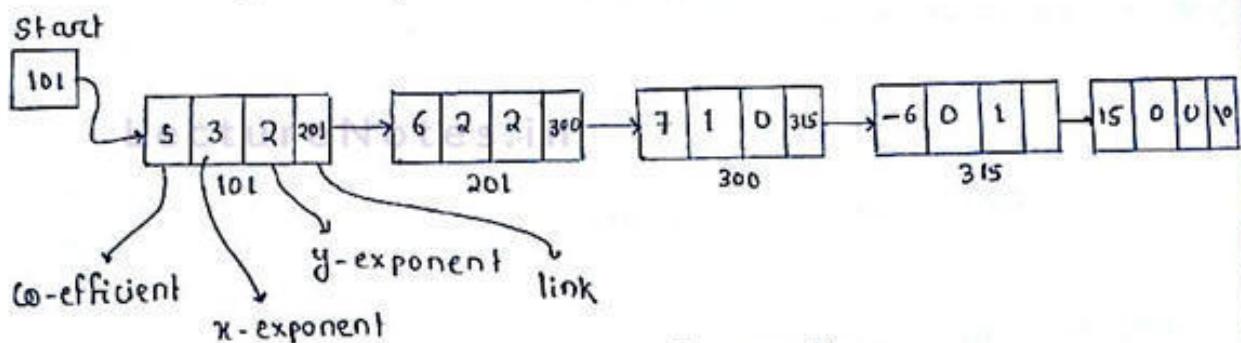
while (t2 != '\0')
{
    nw = (struct node *) malloc(sizeof(struct node));
    if (nw == '\0')
        printf ("Overflow");
    else {
        nw->link = '\0';
        nw->cof = t2->cof;
        nw->exp = t2->exp;
        last->link = nw;
        last = nw;
        t2 = t2->link;
    }
}

```

Representing more than one variable :

let the polynomial is:

$$5x^3y^2 + 6x^2y^2 + 7x - 6y + 15$$



(Representation of more than one variable)

NOTE :

- Total no. of nodes = Total no. of terms in
in the list the polynomial
 - if total number of variables in the polynomial = n , then
each node will have $n+2$ parts.



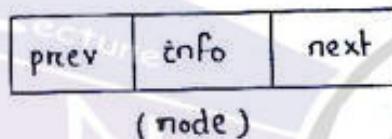
Data Structure Using C

Topic:
Double Link List

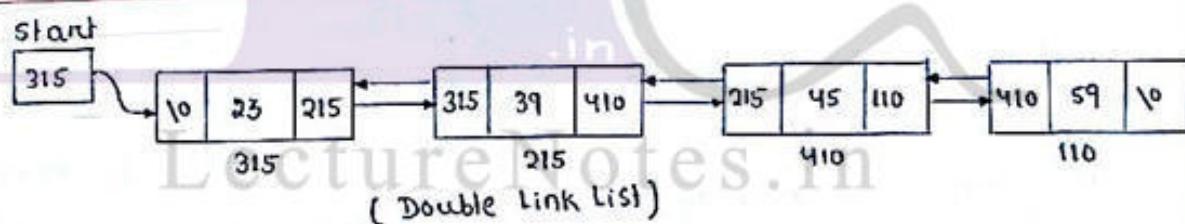
Contributed By:
Mamata Garanayak

II DOUBLE LINK LIST :

- A double link list is a two-way list, because in this list, one can move in either from left to right or from right to left.
- In case of a double link list each node having three parts; one information part and two link parts.
- Out of the two link part, one link part will hold the address of next node and another link part will hold the address of previous node.
- So, in double link list information part has been named as info, and two link part has been named as next and prev respectively.



Representation of double link list :



Declaration :

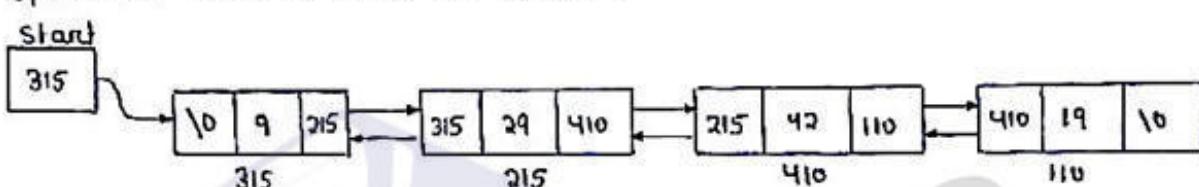
```
struct dnode
{
    int info;
    struct dnode *next;
    struct dnode *prev;
} * start = '\0';
```

OPERATIONS :

Operations in a double link list includes :

- 1) Creation
- 2) Traversing
- 3) Insertion
- 4) Deletion.

1) Creation of a Double Link List : The creation operation is a operation which is used to create a double link list .



Algorithm :

create (nw, n, pprev, next , start , info)

// nw : The node which is to be created .

// n : Number of nodes

// pprev : pointer variable , which holds the address of previous node .

// next : pointer variable which holds the address of next node .

// start : Declared as global variable , which holds the address of 1st node of the list .

// info : Information part of a node .

Step1 : Assign the no. of nodes you want to create .

Step2 : [Creation of new node]

```
for ( i = 0 ; i < n ; i++ )  
    i) nw = allocate memory .
```

Step3 : [check overflow]

```
if ( nw == NULL )  
    a. print overflow  
    b. exit .
```

[End if]

Step 4 : Enter the information to the info part of a node .

Step 5 : $nw \rightarrow p_{prev} = \text{NULL}$
 $nw \rightarrow next = \text{NULL}$

Step 6 : [checking whether the entered node is 1st node or not]
if ($start == \text{NULL}$)
 i) $start = nw$. ii) otherwise goto step 7 .

LectureNotes.in

Step 7 : else
 $start \rightarrow prev = nw$.
 $nw \rightarrow next = start$.
 $start = nw$

[End of for]

Step 8 : Exit

C - Procedure :

```
void create()
{
    struct node *nw;
    int i, n;
    printf ("Enter no of nodes in a double link list");
    scanf ("%d", &n);

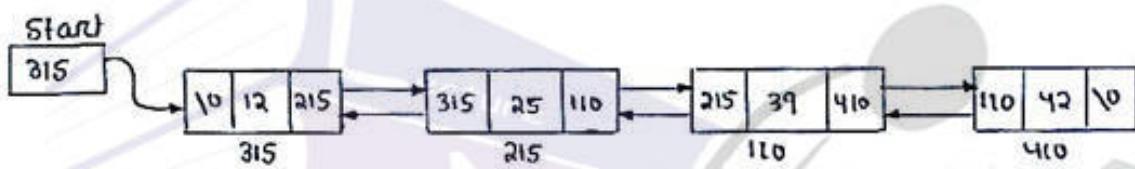
    for (i=0; i<n; i++)
    {
        nw = (struct node *) malloc (sizeof(struct node));
        if (nw == NULL)
        {
            printf ("overflow");
            exit();
        }
        printf ("Enter the information to the node :");
        scanf ("%d", &nw->info);
        nw->prev = NULL;
        nw->next = NULL;
    }
}
```

```

if ( start == NULL )
    start = new;
else {
    start->next = new;
    new->prev = start;
    start = new;
}
}

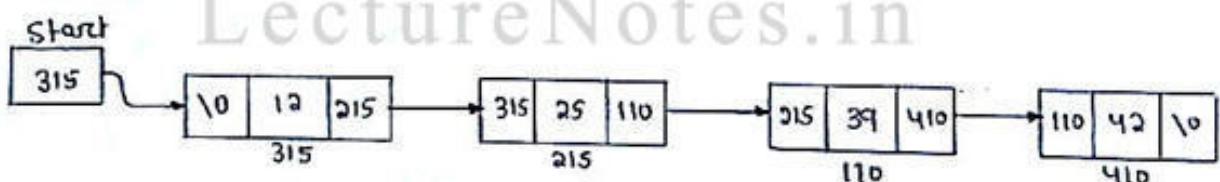
```

2) Traversing a Double Link List : This operation is used to visit all the elements in a double link list.



- There are two types of traversing.
 - a. Forward Traversing
 - b. Backward Traversing

a. Forward Traversing :



Algorithm :

```

Forward traverse ( list , temp , prev , next , info , start )
    // list : double link list
    // temp : Temporary variable
    // prev: pointer variable that holds the address of the last node
    // next: pointer variable that holds the address of the prev next
    //       node .
    // info: Hold the information part of a node.
    // start: List pointer variable which has hold the address of
    //        1st node of the list .

```

Step1 : Set temp := start .
Step2 : if temp = NULL then
 print underflow .

Step3 : else
 for temp = start , temp != NULL , temp = temp → next
 i) print temp → info .
 ii) print NULL .

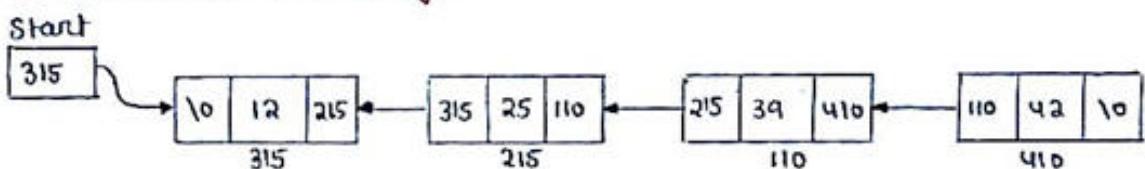
[End For]

Step4 : Exit

C - procedure :

```
void forward-traverse()
{
    // start, prev, next, info has been declared as global variable.
    struct node *temp;
    temp = start;
    if (temp == NULL)
        printf ("Underflow");
    else {
        for (temp = start; temp != NULL; temp = temp → next)
            printf ("%d→", temp → info);
        printf ("NULL");
    }
}
```

b. Backward Traversing:



Algorithm:

PROCEDURE Backward-Traverse (list, start, next, pprev, info, temp)

// List : Existing double link list

// start : List pointer variable which has hold the address of 1st node of the list.

// next : pointer variable which has hold the address of next node

// pprev : pointer variable which has hold the address of previous node.

// info : variable which hold the information part of each node.

// temp : temporary pointer variable.

Step1 : set temp := start .

Step2 : if temp = null then
 print List is empty .

Step3 : else

 for (temp = start ; temp → next != null , temp = temp → next);

 for (temp != null , temp = temp → pprev)

 i) print temp → info ;

 ii) print null .

Step4 : Exit .

C - procedure : void Backward-Traverse()

{

// start, next, info, pprev has been declared as global variable.

struct dnode * temp;

```

temp = start ;
if (temp == '\0')
    printf (" Underflow");
else {
    for ( ; temp->next != '\0' ; temp = temp->next);
    for ( ; temp != '\0' ; temp = temp->prev)
        printf (" .d-> ", temp->info);
    printf (" NULL");
}
}

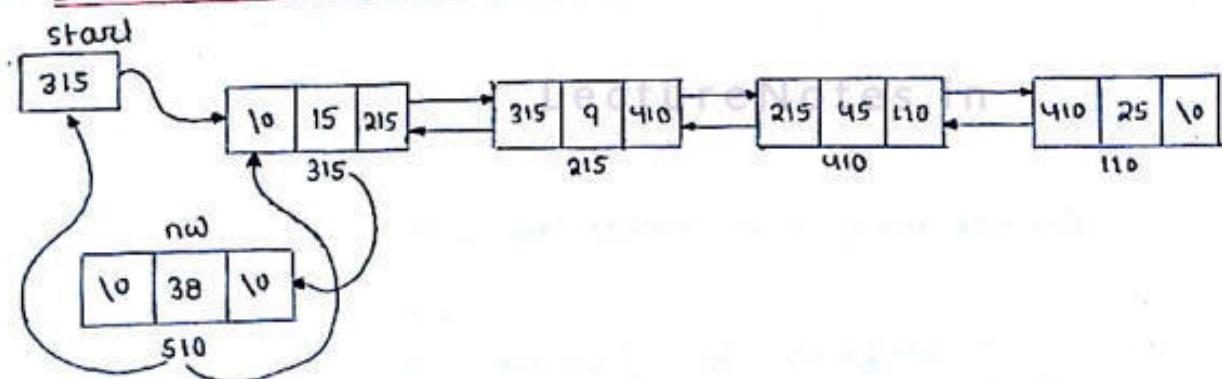
```

3) Insertion :

The insertion operation is used to add an element in a double link list. There are various positions where a node can be inserted.

- Insert a node as a first node.
- Insert a node as a last node.
- Insert a node at any position.
- Insert a node after a given position.
- Insert a node before a given position.

a. Insert a node as a first node :



Algorithm :

Insert-a-node-as-a-first node (list, start, next, prev, info, now)

// list : Existing double link list

// start : list pointer variable which holds the address of first node.
// next : pointer variable which holds the address of next node.
// prev : pointer variable which holds the address of previous node.
// info : variable for holding the information part of each node.
// nw : pointer variable which holds the address of node created dynamically.

Step 1 : Set nw = (struct node *) malloc (sizeof (struct node))

Step 2 : If nw = NULL then
 print overflow.

Step 3 : else
 a. set nw → next = NULL .
 b. set nw → prev = NULL .
 c. Read : nw → info .

Step 4 : [check for Emptyness]
 if start = NULL then start = nw .

else
 d. set nw → next = start
 e. set start → prev = nw .
 f. set start = nw .

[End of Step-4 if structure]

Steps : Exit .

C-procedure :

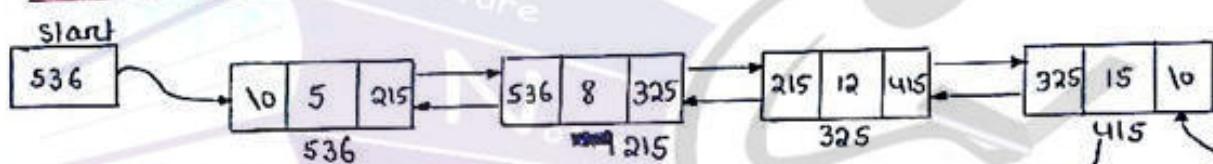
```
Void insert-a-node-as-a-first-node ()  
{  
    // start , info , next , prev are declared as a global variable .  
    struct dnode *nw ;  
    nw = (struct node *) malloc (sizeof (struct node)) ;  
    if ( nw == NULL )  
        printf ("overflow") ;  
    else
```

```

nw->next = NULL;
nw->prev = NULL;
printf (" Enter the information part");
scanf (" %d", &nw->info);
if ( start != NULL)
{
    nw->next = start;
    start->prev = nw;
}
start = nw;
}
}

```

b. Insert a node as a last node :



Algorithm:

```

Insert-a-node-as-last-node (list, next, prev,
                           start, nw, info, temp)
// List : Existing double link list.
// next : pointer variable , which holds the address of the next node .
// prev : pointer variable, which holds the address of the previous
       node .
// start : list pointer variable, which holds the address of 1st node
       of the double link list .
// nw : pointer variable, which holds the address of dynamically
       created node .
// info : variable which holds the information part of each node .
// temp : Temporary pointer variable .

```

Step 1 : Set $nw = (\text{struct node} *) \text{malloc}(\text{sizeof}(\text{struct node}))$

Step 2 : If $nw = \text{NULL}$ then :

 print overflow.

Step 3 : Else

 (a) Set $nw \rightarrow \text{next} = \text{NULL}$

 (b) Set $nw \rightarrow \text{prev} = \text{NULL}$

 (c) Read $nw \rightarrow \text{info}$.

 (d) If $\text{start} = \text{NULL}$ then :

 set $\text{start} = nw$.

 (e) else

 i) $\text{for}(\text{temp} = \text{start} \text{ to } \text{temp} \rightarrow \text{next} \neq \text{NULL},$
 $\text{temp} = \text{temp} \rightarrow \text{next})$

 [End for]

 ii) set $\text{temp} \rightarrow \text{next} = nw$

 iii) set $nw \rightarrow \text{prev} = \text{temp}$

 [End of step 3(d) if structure]

 [End of step 2 if structure]

Step 4 : Exit.

C - Procedure :

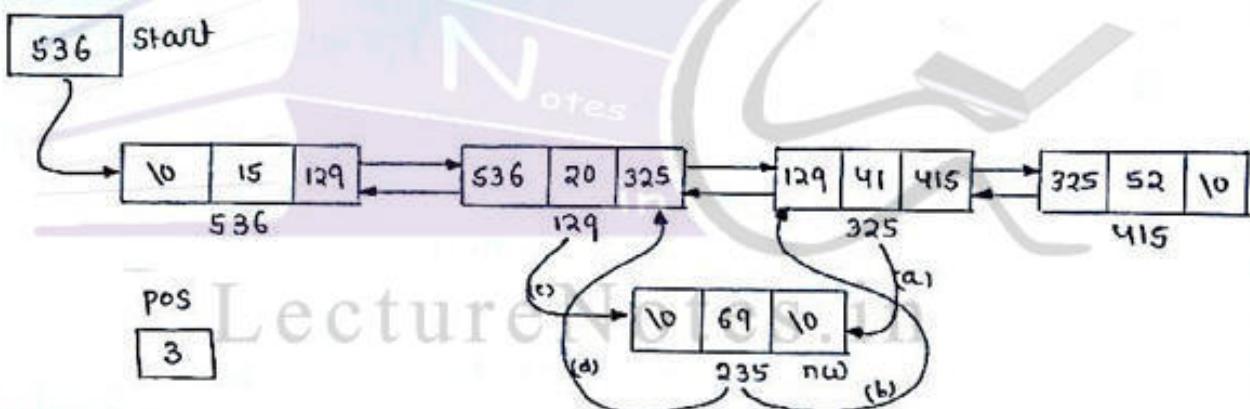
```
void insert-at-end()
{
    struct dnode *nw;
    nw = (struct node *) malloc (sizeof (struct node));
    if (nw == NULL)
        printf ("Overflow");
    else
    {
        nw->prev = NULL;
        nw->next = NULL;
```

```

printf (" Enter the information part :");
scanf ("%d", &nw->info);
if ( start->next == NULL)
    start = nw;
else
{
    for( temp = start; temp->next != NULL ; temp = temp->next );
        temp->next = nw;
        nw->prev = temp;
}
}

```

Insert a node at any position :



Algorithm :

insert-a-node-at-any-position (nw, start, list, next, prev, i, c, pos, temp)

// nw: pointer variable which holds the address of the newly created node.

// start: pointer variable which holds the address of the 1st node in the list .

// list : Existing double link list .

// next: pointer variable which holds the address of the next node.

// prev: pointer variable which holds the address of the previous node.

// i: variable which will be used in loop control.

// pos: position at which we want to insert the new node .

① C Counter variable

② temp: Temporary pointer variable which will be used for traversing the list

step1: [checking for valid position]

a. Read pos.

b. if pos > c+1 then :

 point : Invalid position.

[End if]

step2: [if entered position is one]

else if pos = 1 then :

 a. set nw → next = start

 b. set start → pnext = nw

 c. set start = nw

[End of else if]

step3: [if entered position is last]

else if pos = c+1 then :

 a. set temp = start

 b. while temp → next != NULL
 set temp = temp → next

[End while]

 c. set temp → next = nw

 d. set nw → pnext = temp

step4: [if entered position is other than first & last position]

else

 a. set temp = start

 b. set i = 1

 c. while i < pos-1

 set temp = temp → next

[End while]

- d. set $nw \rightarrow next = temp \rightarrow next$ (a)
- e. set $(temp \rightarrow next) \rightarrow pnext = nw$ (b)
- f. set $nw \rightarrow pnext = temp$ (c)
- g. $temp \rightarrow next = nw$ (d)

[End of else]

[End of step-1(b) if structure]

Step 5 : Exit .

C - procedure :

```

void insert-a-node-at-any-position()
{
    int c, pos, c=0;
    struct node *temp, *nw;
    printf("Enter the position at where new node is to be inserted :");
    scanf("%d", &pos);
    for( temp = start; temp != NULL; temp = temp->next )
        c=c+1;
    if( pos > c+1)
        printf(" Invalid position");
    else
    {
        nw = (struct node *) malloc(sizeof(struct node));
        if( nw == NULL)
            printf(" malloc can not allocate memory");
        else
        {
            nw->next = NULL;
            nw->pnext = NULL;
            printf(" Enter the information part ");
            scanf("%d", &nw->info);
        }
    }
}

```

```

if ( pos == 1 )
{
    if ( start != NULL )
    {
        nw → next = start ;
        start → prev = nw ;
    }
    start = nw ;
}
else if ( pos == c+1 )
{
    for( temp = start ; temp → next != NULL ; temp = temp → next ) ;
        temp → next = nw ;
        nw → prev = temp ;
    }
}
else
{
    for( temp = start , i=1 ; i < pos-1 ; temp = temp → next , i++ ) ;
        nw → next = temp → next ;
        (temp → next) → prev = nw ;
        nw → prev = temp ;
        temp → next = nw ;
    }
}
}
}

```



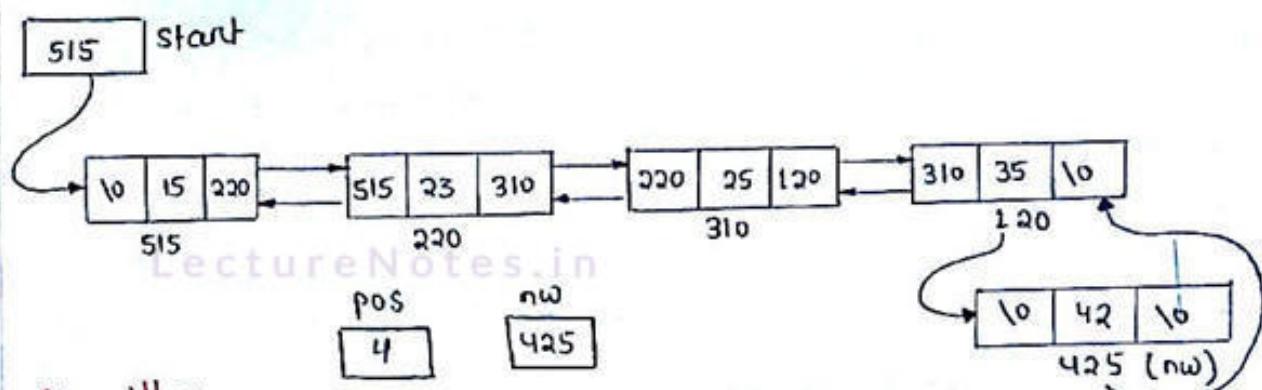
Data Structure Using C

Topic:

Insertion After A Given Position

Contributed By:

Mamata Garanayak

Insertion after a given position :Algorithm :

```
insert-after-a-given-position (List, start, next, prev, nw, temp,
                               pos, i, c)
```

- // List : Existing double link list.
- // start : List pointer variable which has hold the address of 1st node of the list.
- // next : pointer variable which hold the address of next node.
- // prev : pointer variable which hold the address of previous node.
- // nw : pointer variable which holds the address of newly created node.
- // temp : Temporary pointer variable , which will be used in traversing.
- // i : loop Control variable.
- // c : Counter variable .
- // pos : Variable which has hold the entered position after which node is to be inserted.

Step 1: if (pos > c) then :

print : Invalid position .

[End if]

Step 2: Else

A. set nw = (struct node *) malloc (sizeof (struct node));

B. if nw == NULL then :

print : overflow .

[End if]

else

- a. set nw → next = NULL
- b. set nw → prev = NULL
- c. Read nw → info

Step 3 : if pos = C then : for(temp = start ; temp → next != NULL ;
temp = temp → next);
 i. Set temp → next = nw
 ii. set nw → prev = temp

Step 4 : else for(temp = start , i=1 ; i < pos ; temp = temp → next , i++); *

on a. set temp = start

b. set i=1 .

c. while (i < pos - 1)

 set temp = temp → next

[End while]

• d. nw → next = temp → next

e. set (temp → next) → prev = nw

f. set nw → prev = temp

g. set temp → next = nw

[End of else]

[End of step 1(a) if structure]

Step 5 : Exit

LectureNotes.in

C - procedure :

```
Void insert-at-any-position()
{
    // Start , next , prev , info have been declared as global variable
    int i , pos , C = 0 ;
    struct node * temp , * nw ;
    printf ("Enter the position after which the node is to be inserted") ;
```

```

scanf ("%d", &pos);
for (temp = start ; temp != NULL ; temp = temp->next)
    c = c + 1;
if ( pos > c )
    printf (" Invalid position");
else
{
    nw = (struct node *) malloc (sizeof (struct node));
    if ( nw == NULL )
        printf (" overflow");
else
{
    nw->next = NULL;
    nw->pnext = NULL;
    printf (" Enter the information part of the node : ");
    scanf ("%d", &nw->info);
    if ( pos == c )
    {
        for( temp = start ; temp->next != NULL ; temp = temp->next)
            ;
        temp->next = nw;
        nw->pnext = temp;
    }
    else
    {
        for (temp = start , i=1 ; i<pos ; temp = temp->next , i++)
            ;
        nw->next = temp->next ;
        (temp->next)->pnext = nw ;
        nw->pnext = temp ;
        temp->next = nw ;
    }
}
}
}

```

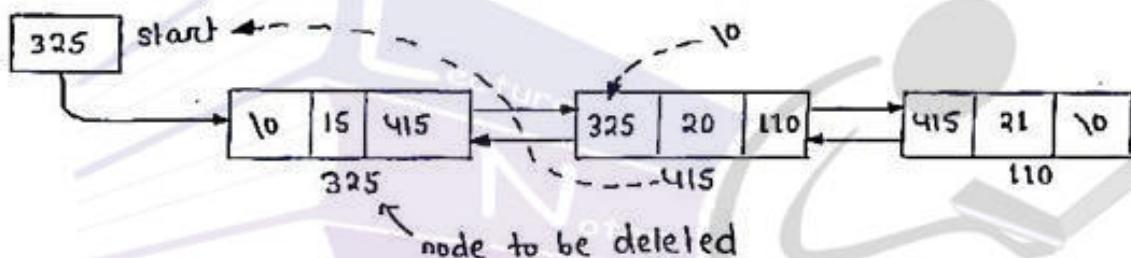
NOTE: Inserting a node at a given position is similar to inserting a node before a given position.

4) Deletion :

Deleting a node from the double link list is called deletion.

- 1) Deletion of first node
- 2) Deletion of last node
- 3) Deletion at given position.
- 4) Deletion after a given position

Deletion of First node :



Algorithm :

PROCEDURE delete-first (list, start, next, prev, temp)

// list : Existing double link list
// start : pointer variable which has hold the address of first node.
// next : pointer variable which has hold the address of next node.
// prev : pointer variable which has hold the address of previous node.
// temp : temporary pointer variable used for traversing.

Step1 : if $\text{start} = \text{NULL}$ then :
 print underflow.
 [End if]

Step2 : Else if $\text{start} \rightarrow \text{next} = \text{NULL}$ then :
 a. set $\text{temp} = \text{start}$
 b. set $\text{start} = \text{NULL}$

c. `free (temp)`.
[End else if]

Step3 : Else

- a. `set temp = start.`
- b. `set start = start → next`
- c. `(temp → next) → prev = null`
- d. `temp → next = null.`
- e. `free (temp).`

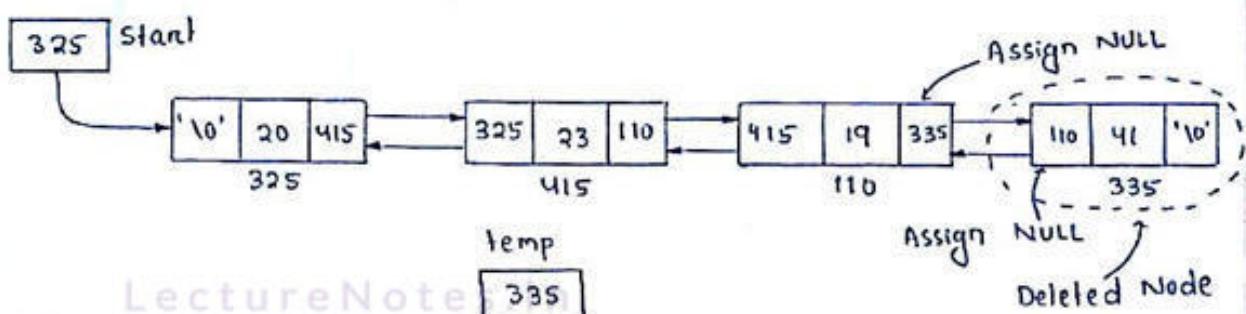
[End else]

Step4 : Exit.

C - Procedure :

```
void delete-first-node()
{
    // start, next, prev have been declared as global variable.
    struct node *temp;
    if (start == NULL)
        printf ("Underflow");
    else if (start → next == NULL)
    {
        temp = start;
        start = NULL;
        free (temp);
    }
    else
    {
        temp = start;
        start = start → next;
        (temp → next) → prev = NULL;
        temp → next = NULL ;
        free (temp);
    }
}
```

Deletion Of Last Node :



NOTE: In single link list, two pointer variables are required to delete the last node but here we required only one pointer variable to delete last node. This is the advantage of double link list.

Algorithm :

```

PROCEDURE delete-last ( List , start , next , prev , temp )
// list : Existing double link list .
// start : pointer variable which holds the address of first node .
// next : pointer variable which has hold the address of next node .
// prev : pointer variable which has hold the address of previous node .
// temp : temporary pointer variable used for traversing .

step1 : if ( start = null ) then
        1. print Underflow .
        2. exit
    [ End if ]

Step2 : else if start → next = null then :
        a. set temp = start
        b. start = null .
        c. free ( temp ) .
    [ End of else if ]

Step3 : else
        a. temp = start .
        b. while temp → next ≠ null
            set temp = temp → next
    [ End while ]

```

- c. $\text{temp} \rightarrow \text{prev} \rightarrow \text{next} = \text{null}$
- d. $\text{temp} \rightarrow \text{prev} = \text{null}$
- e. free temp.

[End if else]

[End of step 1 if structure]

Step 4 : exit.

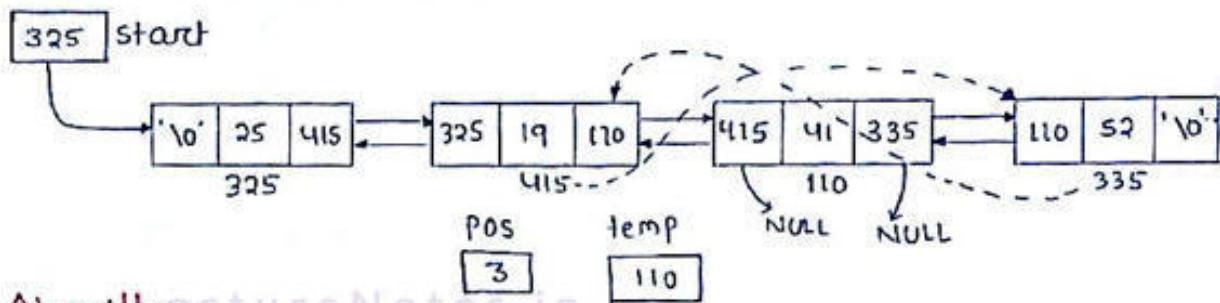
C - Procedure :

```

{ void deletLast()
// start, next, prev, have been declared as global variable .
struct node *temp;
if ( start == NULL )
    printf ("Underflow");
else if ( start->next == NULL )
{
    temp = start;
    start = NULL;
    free (temp);
}
else
{
    for ( temp = start ; temp->next != NULL ; temp = temp->next )
        ;
    temp->prev->next = NULL;
    temp->prev = NULL;
    free (temp);
}

```

Deletion at Given position:



Algorithm:

```
PROCEDURE delete-at-pos (List, start, next, pprev, temp, i, c, pos)
// List : Existing double link list .
// start : List pointer variable which has hold the address of 1st node
// of the list .
// next : pointer variable of each node which has hold the address of
// next node .
// pprev : pointer variable of each node which has hold the address
// of previous node .
// temp : Temporary pointer variable which is used for traversing .
// i : Loop Counter variable .
// c : variable which has hold the no. of nodes present in the list .
// pos : variable which has hold the position of the node which is
// to be deleted .
```

Step1: if start = null then :

1. print underflow
2. exit.

[End if]

Step2: else

<A> if pos > c then
print invalid position .

[End if]

 Else if pos = 1 then
a. if start → next = null then
i. temp = start

ii. $\text{start} = \text{null}$.
iii. $\text{free}(\text{temp})$
b) [End if]
else if $i > \text{temp} = \text{start}$.
i) $\text{start} = \text{start} \rightarrow \text{next}$
ii) $\text{temp} \rightarrow \text{next} \rightarrow \text{prev} = \text{null}$.
iv) $\text{temp} \rightarrow \text{next} = \text{null}$.
v) $\text{free}(\text{temp})$.

[End of else]
[End of step 2(B) if structure]

[End of else if]

c) Else if $\text{pos} = \text{c}$ then

- $\text{temp} = \text{start}$
- while $\text{temp} \rightarrow \text{next} \neq \text{NULL}$
set $\text{temp} = \text{temp} \rightarrow \text{next}$.
[End of while]
- $\text{temp} \rightarrow \text{prev} \rightarrow \text{next} = \text{null}$.
- $\text{temp} \rightarrow \text{prev} = \text{null}$
- $\text{free}(\text{temp})$

[End of else if]

d) Else

- $\text{temp} = \text{start}$.
- for i=1 to $\text{pos}-1$
 $\text{temp} = \text{temp} \rightarrow \text{next}$
[End of for]
- $\text{temp} \rightarrow \text{next} \rightarrow \text{prev} = \text{temp} \rightarrow \text{prev}$.
- $\text{temp} \rightarrow \text{prev} \rightarrow \text{next} = \text{temp} \rightarrow \text{next}$
- $\text{temp} \rightarrow \text{next} = \text{null}$.
- $\text{temp} \rightarrow \text{prev} = \text{null}$.
- $\text{free}(\text{temp})$.

[End of else]
[End of step 2(A) if structure]
[End of step 2 else]
[End of step 1 if structure]

Step 3 : Exit .

C - Procedure :

```
void delete-at-pos()
{
    // start, prev, next have been declared as global variable.

    struct node * temp;
    int i, c=0, pos;
    if ( start == NULL )
        printf (" Underflow ");
        exit();
    else
    {
        printf (" Enter the position of the node that is to be deleted ");
        scanf ("%d", & pos);
        for ( temp = start ; temp != NULL ; temp = temp->next )
            c = c + 1;
        if ( pos > c )
            printf (" Invalid Position ");
        else if ( pos == 1 )
        {
            if ( start->next == NULL )
            {
                temp = start;
                start = NULL;
                free (temp);
            }
        else {
```

```

for (first)
{
    temp = start;
    start = start->next;
    temp->next->pnext = NULL;
    temp->next = NULL;
    free (temp);
}

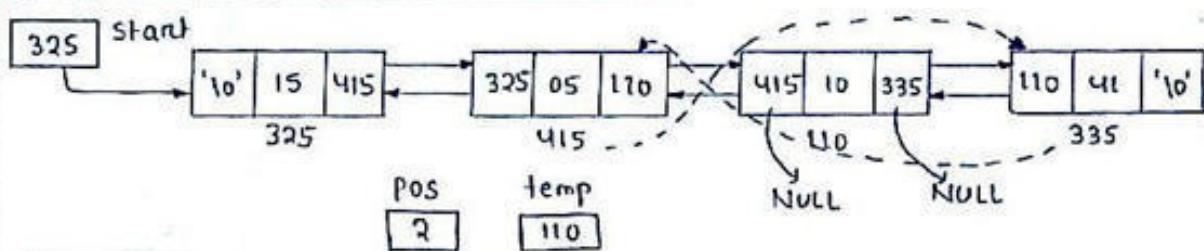
}

else if (pos == c)
{
    for( temp = start ; temp->next != NULL ; temp = temp->next );
    temp->pnext->next = NULL;
    temp->pnext = NULL;
    free (temp);
}

else
{
    for( i=1 , temp = start ; i<pos ; temp = temp->next , i++ );
    temp->next->pnext = temp->pnext;
    temp->pnext->next = temp->next;
    temp->pnext = NULL;
    temp->next = NULL;
    free (temp);
}
}
}

```

Deletion After a Given position :



Algorithm :

```
PROCEDURE delete - after - pos (list, start, next, prev, i, temp, c, pos)
// List : Existing double link list .
// start : List pointer variable which has hold the address of 1st
//          node of the list .
// next : pointer variable which is used to hold the address of next
//          node .
// prev : pointer variable which is used to hold the address of prev
//          node .
// i : Loop Counter variable .
// temp : temporary pointer variable used for traversing .
// c : variable which has hold the no. of nodes present in the list .
// pos : variable which has hold the position of node which is to
//          be deleted .
```

Step1 : Read POS .

Step2 : if start = null then
 print underflow .

Step3 : Else

- temp = start .
- while temp ≠ null
 - set temp = temp → next .
 - c = c + 1 .

[End while]

- if pos ≥ c then :

print invalid position.

d) else if pos = c - 1 then :

- i) temp = start
- ii) while temp → next ≠ null
temp = temp → next .
[End while]

iii) temp → prev → next = null .

- iv) temp → prev = null .
- v) free (temp) .

[End of else if]

e) else

- i) temp = start .
- ii) for i = 1 to pos
temp = temp → next
[End of for]

iii) temp → next → prev = temp → prev .

iv) temp → prev → next = temp → next .

v) temp → next = null .

vi) temp → prev = null .

vii) free (temp) .

[End of else]

[End of step 3(c) if structure]

[End of step 3 else] .

[End of step 2 if structure .

C - procedure :

```
void delete-after-pos()
{
    // start, next and prev have been declared as global variable
    struct node *temp;
    int i, c=0, pos;
    printf("Enter the position after which the node will be deleted");
    scanf("%d", &pos);

    if (start == NULL)
        printf("underflow");

    else
    {
        for (temp = start; temp != NULL; temp = temp->next)
            c = c + 1;

        if (pos >= c)
            printf("Invalid position");

        else if (pos == c - 1)
        {
            temp = start;
            for ( ; temp->next != NULL; temp = temp->next);
            temp->prev->next = NULL;
            temp->prev = NULL;
            free(temp);
        }
        else
        {
            for (i=1, temp = start; i < pos+1; temp = temp->next)
                temp->prev->next = temp->prev;
            temp->next->prev = temp->next;
            temp->next = NULL;
            temp->prev = NULL;
            free(temp);
        }
    }
}
```



Data Structure Using C

Topic:

Searching In Double Link List

Contributed By:

Mamata Garanayak

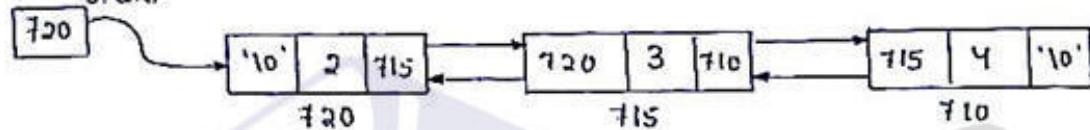
NOTE: Inserting a node at a given position is similar to inserting a node before a given position.

SEARCHING in Double Link List :

Searching in double link list is the process in which an entered item ~~with~~ is searched in a given list by comparing with each data element sequentially and position of item is found out.

Algorithm :

start



PROCEDURE Search (list, start, info, next,
prev, temp, item, c)

item
4

- // List : Existing double link list.
- // start : List pointer variable which has hold the address of 1st node
- // info : variable which has hold the information part.
- // next : pointer variable which has hold the address of next node.
- // prev : pointer variable which has hold the address of previous node
- // temp : temporary pointer variable used for traversing.
- // item : variable which has hold the item to be searched.
- // c : variable which holds the no. of nodes present in the list.

Step 1 : [CHECKING OF UNDERFLOW]

if start = null then :

print underflow

[End if].

Step 2 : else

A) temp = start

B) while temp ≠ null and temp → info ≠ item
i) c = c + 1

- ii) $\text{temp} = \text{temp} \rightarrow \text{next}$.
 [End of while]
- c) if $\text{temp} = \text{null}$ then
 print searching is not possible .
- d) else
 print item is found in C'th node having address temp.
 [End of step c if-else]
 [End of step d if-else]

step 3 : Exit .

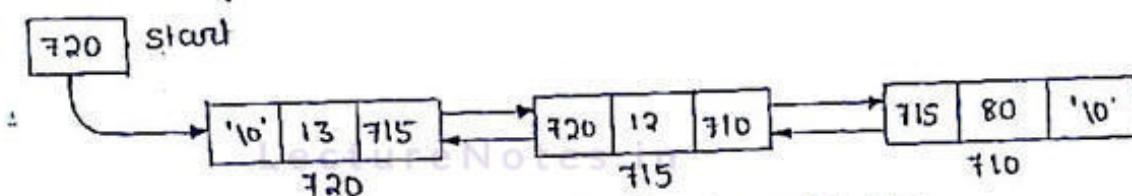
C - Procedure :

```

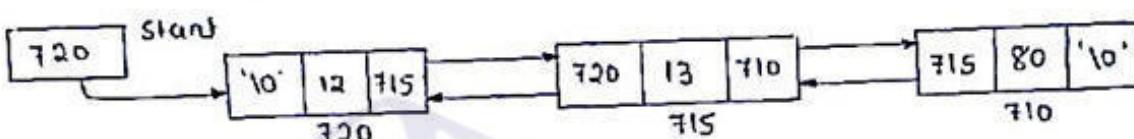
void search()
{
  // start, info, next are declared as global variable
  int c = 0, item;
  struct node *temp;
  printf("Enter the item to be searched");
  scanf("%d", &item);
  if (start == '\0')
    printf("List is empty");
  else
  {
    for (temp = start; temp != '\0' && temp->info != item;
         temp = temp->next)
      c = c + 1;
    if (temp == '\0')
      printf("Searching is not possible");
    else
      printf("Item is found in %d th node having the
             address %u", c+1, temp);
  }
}
  
```

SORTING IN DOUBLE LINK LIST :

Sorting is the process in which the data elements of a list are arranged either in ascending order or descending order sequentially in the node of that list.



After sorting, the process will be list will be;



Algorithm :

```
PROCEDURE Sort( list, start, next, info, temp1, temp2, num )
// List : Existing double link list
// start : List pointer variable which has hold the address of 1st node
// next : pointer variable which holds the address of next node .
// info : variable for hold the information part
// temp1: temp2 : temporary pointer variable , used for traversing .
// num: variable which will be used during swapping the data element
```

Step1: temp = start .

Step2: while temp ≠ null repeat step(3) to step(4) .

Step3: temp2 = temp → next .

Step4: A) while temp2 ≠ null repeat sub steps @ and B).

@ if temp1 → info > temp2 → info

i) num = temp1 → info

ii) temp1 → info = temp2 → info

iii) temp2 → info = num

[End if]

B) temp2 = temp2 → next

[End of u(n) while loop]

Step 5 : $\text{temp}_1 = \text{temp}_1 \rightarrow \text{next}$.

[End of step 2 while loop]

Step 6 : Exit.

C - procedure :

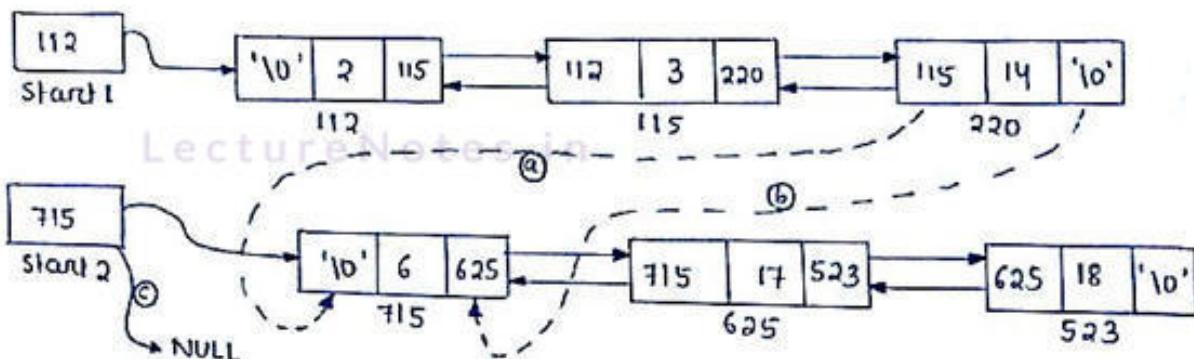
```
void sort()
{
    // start, next, info are declared as global variable.

    int num;
    struct node * temp1, * temp2;

    for (temp1 = Start; temp1 != '\0'; temp1 = temp1->next)
    {
        for (temp2 = temp1->next; temp2 != '\0'; temp2 = temp2->next)
        {
            if (temp1->info > temp2->info)
            {
                num = temp1->info;
                temp1->info = temp2->info;
                temp2->info = num;
            }
        }
    }
}
```

CONCATENATION OF DOUBLE LINK LIST :

Merging of two similar type of double link list to make a single double link list is called concatenation of two double link list.



Algorithm :

```
PROCEDURE Concatenate (list1, list2, start1, start2, temp, pprev, next)
// list1, list2 : 2 existing double link list .
// start1, start2 : List pointer variables which has hold the address
// of the 1st node of list1 and list2 respectively .
// temp: temporary pointer variable used for traversing .
// pprev: pointer variable which has hold the address of previous
// node in the list .
// next : pointer variable which has hold the address of next node of
// the list .

Step1 : temp = start1 .
Step2 : While temp → next ≠ null
        temp = temp → next
        [End while]

Step3 : temp → next = start2 .

Step4 : start2 → pprev = temp .

Step5 : start2 = null .

Step6 : Exit .
```

C - Procedure :

```
Void Concatenate()
{
// start1, start2, next, pnext, temp are declared as global variable.

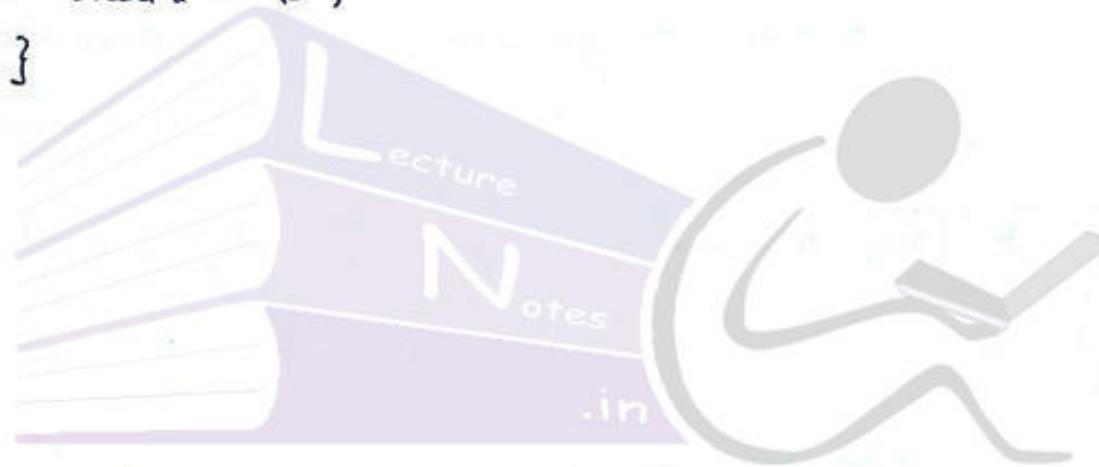
for( temp = start1 ; temp->next != '\0' ; temp = temp->next )
    ;LectureNotes.in

temp->next = start2;

start2->pnext = temp;

start2 = '\0';

}
```



LectureNotes.in

LectureNotes.in



Data Structure Using C

Topic:
Circular Link List

Contributed By:
Mamata Garanayak

CIRCULAR LINK LIST :

It is a type of linear link list in which next pointer field of the last node contains the address of the first node rather than the NULL pointer.

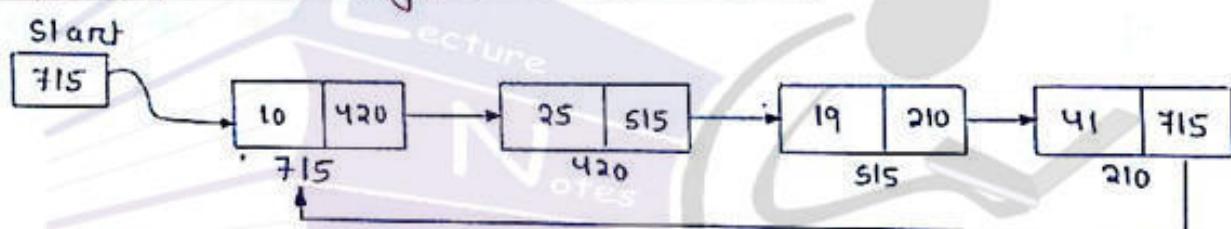
→ Two types of circular link list are there. These are ;

1. Single circular link list .
2. double circular link list .

1. Single circular link list :

In case of single circular link list , the last node points back to the first node i.e. the link part of the last node holds the address of first node instead of containing NULL pointer.

Representation of Single circular link list :



Condition for Underflow :

If the list is empty then start must hold NULL . So

if (start == NULL) then Underflow
the underflow condition arises .

Condition for overflow :

if (nw == NULL) then overflow
condition arises .

Condition for existence of single node :

If there exist a single node in the list then the link part or next pointer field of that node holds the address of it self .

if (start = start → link) then :

Single node is present in the list .

Creation of Single Circular link list :

Algorithm :

PROCEDURE single-circular-link-list (start , info , link , ch , c , nw)

// start : list pointer variable which will hold the address of first node .

// info : variable which will be used to hold the information part of each node .

// link : Next pointer field of each node which will hold the address of next node .

// nw : pointer variable which will hold the address of dynamically created node .

// ch : variable which will hold the entered choice .

// c : variable which will count the no. of node that will be created .

Step1 : do

 nw = (struct node *) malloc (sizeof (struct node))

Step2 : if nw = NULL then

 1. print overflow

 2. exit

[End if]

Step3 : Else

 i) nw → link = NULL

 ii) Read nw → info .

 iii) nw → link = Start

 iv) start = nw

 v) C = C + 1

[End of Step 2, 3 if else structure]

Step4 : Read ch .

 1. For creation of another node

 Q . For stop creation .

Step5 : while ch ≠ 0 repeat step1 to step4

[End of do - while loop]

Step 6 : $\text{temp} = \text{start}$.

Step 7 : while $\text{temp} \rightarrow \text{link} \neq \text{null}$

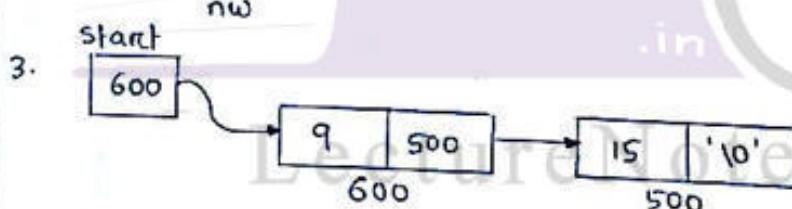
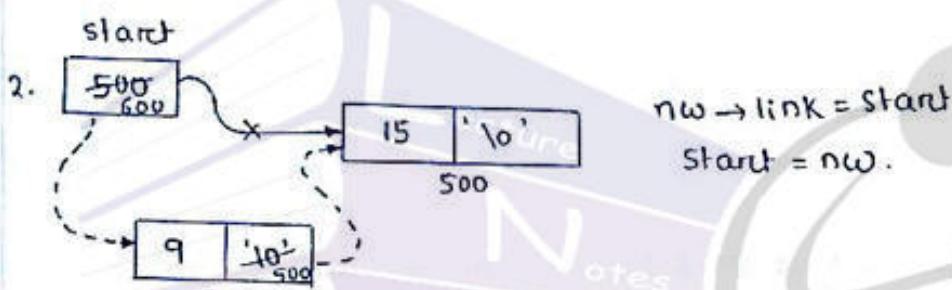
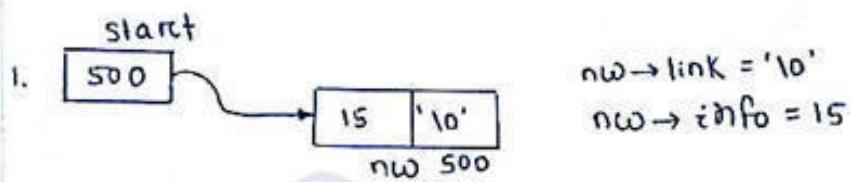
$\text{temp} = \text{temp} \rightarrow \text{link}$

[End of while loop]

Step 8 : $\text{temp} \rightarrow \text{link} = \text{start}$.

Step 9 : print the no. of nodes created are C

Step 10 : LectureNotes.in



4. while ($\text{ch} = 0$) then

$\text{temp} = 600$

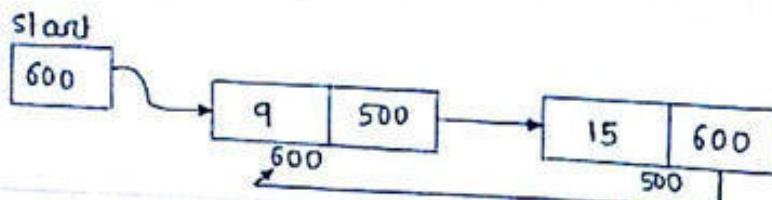
while ($\text{temp} \rightarrow \text{link} \neq \text{null}$)

then $\text{temp} = \text{temp} \rightarrow \text{link}$.

so $\text{temp} = 500$

$\text{temp} \rightarrow \text{link} = \text{start}$

$\Rightarrow \text{temp} \rightarrow \text{link} = 600$

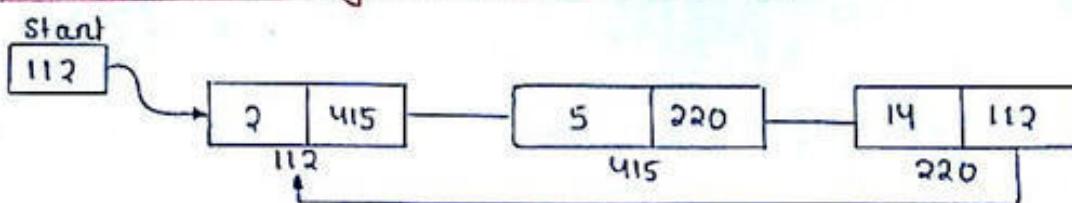


C - Procedure :

```
Void create-circular-link-list()
{
    // start, info and link are declared as global variable .
    Struct node *temp, *nw;
    int ch, C = 0;
    do
    {
        nw = (Struct node *) malloc (sizeof (Struct node));
        if ( nw == '\0' )
        {
            printf ("Overflow");
            exit();
        }
        else
        {
            nw->link = '\0';
            printf ("Enter the information part of the node\n");
            scanf ("%d", & nw->info);
            nw->link = Start;
            Start = nw;
            C = C + 1;
        }
        printf (" Enter your choice\n");
        printf (" 1. For creation \n 0. to stop\n");
    } while (ch != 0);

    for ( temp = Start ; temp->link != '\0' ; temp = temp->link )
    ;
    temp->link = Start;
    printf (" The no. of nodes created are %d", C );
}
```

Traversal of Single Circular link list :



<u>temp</u>	<u>Output</u>
112	2
415	5
220	14
112	loop terminate.

Algorithm:

```
PROCEDURE traverse ( start, temp, link, info )
// start : pointer variable which holds the address of 1st node
// temp : temporary pointer variable used for traversing .
// link : pointer variable which hold the address of next node in
//        the list .
// info : variable which hold the information part of the new nodes .
Step1 : Set temp = start .
Step2 : if start = null then :
        i. print the list is empty .
        ii. exit .
        [End if ]
Step3 : Else
        A> do
                i) print temp → info .
                ii) temp = temp → link
        B> while temp ≠ start repeat step 3-A to step 3-B .
        [End of do while loop ]
        [End of Step 2 if else structure ]
Step4 : Exit .
```

C - procedure :

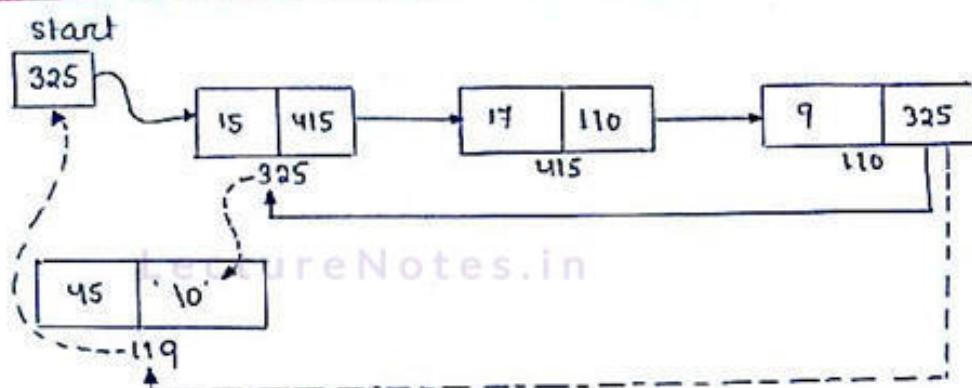
```
void display()
{
// start , info,link have been declared as global variable .
struct node * temp ;
if ( start == '10' )
{
    printf("The list is empty");
    exit();
}
else
{
    temp = start ;
    do
    {
        printf( ".d->", temp->info );
        temp = temp->link ;
    } while ( temp != start );
    printf( ".u", start );
}
}
```

NOTE: Other operations that can be performed in single circular link list are ; searching , sorting , insertion , deletion etc.

LectureNotes.in

Insertion in single Circular link list :

Insertion at first position :



Algorithm :

insert-at-first (start, info, link, nw, temp)

- // start : pointer variable which stores the address of first node .
- // info : information part of the node .
- // link : pointer variable which hold the address of next node .
- // nw : pointer variable which hold the address of newly created node .
- // temp : temporary pointer variable used for traversing .

Step1: Create a new node and store its address in "nw".

Step2: if ($nw = \text{NULL}$) then :

1. print overflow .
2. exit .

[End if]

Step3: Else

1. Enter the information part of the node .
2. $nw \rightarrow link = '10'$.

Step4: if (start = NULL) then .

- i) $nw \rightarrow link = \text{pre. start nw}$
- ii) $start = nw$.

[End if]

Step5 : else

- i) $nw \rightarrow link = start$

ii) $\text{start} = \text{nw}$.
 for ($\text{temp} = \text{start} \rightarrow \text{link}$; $\text{temp} \rightarrow \text{link} \neq \text{start} \rightarrow \text{link}$; $\text{temp} = \text{temp} \rightarrow \text{link}$)
 ;
 $\text{temp} \rightarrow \text{link} = \text{nw}$.
 [End for]
 [End of step 5 else]
 [End of step 3 else].
 Step 6 : Exit

C - procedure :

```

void insert-at-final()
{
  // start, info and link are declared as global variable.
  struct node *nw, *temp;
  nw = (struct node *) malloc (size of (struct node));
  if (nw == '\0')
    { printf ("Overflow");
      exit();
    }
  else
    {
      printf ("Enter the information part of the node");
      scanf ("%d", &nw->info);
      nw->link = '\0';
      if ( start == '\0' ) // CODE FOR EMPTYNESS.
        {
          nw->link = nw;
          start = nw;
        }
      else
        {
          nw->link = start;
          nw = start = nw;
        }
      for ( temp = start->link ; temp->link != start->link ; temp = temp->link )
        ;
      temp->link = nw;
    }
}
  
```

NOTE: After inserting new node, traversal operation is performed. If we will traverse before insertion then else part will be,

```
for (temp = start; temp->link != start; temp = temp->link);  
    temp->link = nw;  
    nw->link = start;  
    start = nw;
```

Insertion at last position:

Algorithm:

```
PROCEDURE insert-at-last (list, start, info, link, temp, nw)
```

// list : Existing circular link list.
// start : pointer variable which hold the address of the first node.
// info : variable which hold the information part of each node.
// link : pointer variable which hold the address of next node.
// temp : temporary pointer variable used for traversing.
// nw : pointer variable which hold the address of newly created node.

Step1: Create a new node and store it in address "nw".

Step2: if nw = null then:
 i. print overflow
 ii. exit

[End if]

Step3: Else
 i) Read nw->info
 ii) Assign nw->link = '10'.

Step4: if (start = null) then:
 a) nw->link = nw.
 b) start = nw.

[End if step 4]

Step 5 :

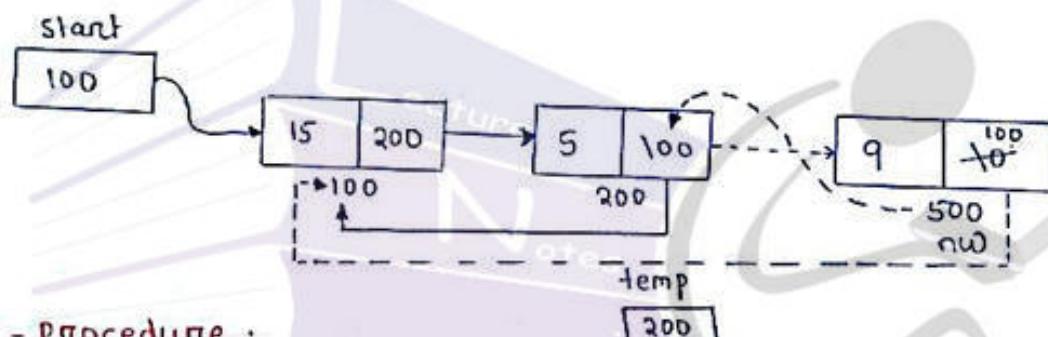
Else

- a) temp = start
- b) while temp → link ≠ start
 - i) temp = temp → link
- [End of while]
- c) temp → link = nw.
- d) nw → link = start

[End of step 4 if..else structure]

[End of Step 3 else]

Step 6 : Exit



C - procedure :

```

void insert-at-last()
{
    // start, info, and link are declared as a global variable.
    struct node *nw, *temp;
    nw = (struct node *) malloc (size of (struct node));
    if (nw == '10')
        printf (" overflow");
        exit();
    }
    else {
        printf (" Enter the information part");
        scanf (" f.d", &nw→info);
        nw→link = '10';
    }
}

```

```
if ( start == '\0' )
{
    now->link = now;
    start = now;
}
else
{
    for ( temp = start ; temp->link != start ; temp = temp->link ) ;
    temp->link = now;
    now->link = start;
}
}
```



Data Structure Using C

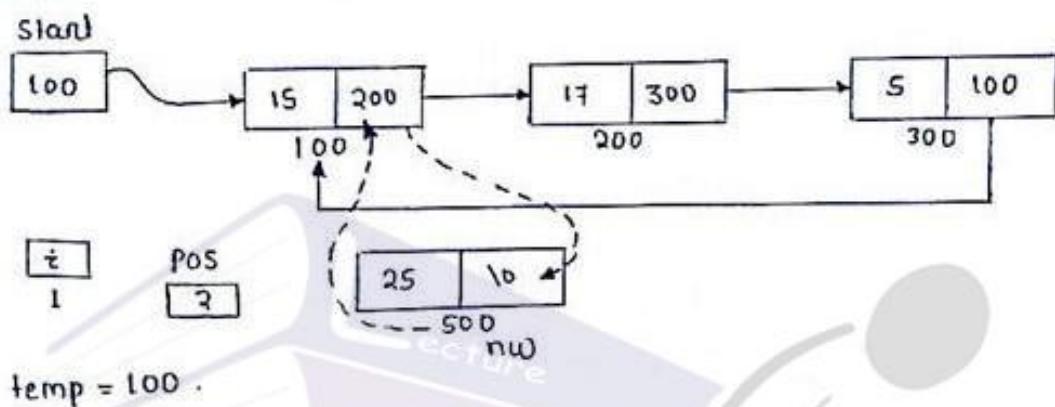
Topic:

Insertion At Any Position

Contributed By:

Mamata Garanayak

Insertion at any position :



Algorithm :

```

PROCEDURE insert-at-any-pos( info, start, nw, link, i, c, pos, temp )
// info : variable which holds the information part of each node .
// start : pointer variable which holds the address of 1st node .
// nw : pointer variable which hold the address of next node .
// link : pointer variable which hold the address of previous node .
// i : loop counter variable .
// c : variable used for counting the no. of nodes present
// in the list .
// pos : position at which the newnode is to be inserted .
// temp : Temporary pointer variable used for traversing .

```

Step1: Enter the position where the node is to be inserted.

Step2: [CHECKING FOR VALIDITY OF POSITION]

i) temp = start

ii) do

 i) C = C + 1,

 ii) temp = temp → link

 iii> while (temp != start);

LectureNotes.in

if (pos > C + 1)

 a) print invalid position

 b) exit

[End if]

Step3: else

 a) nw = allocate memory

 b) if (nw = null) then

 i) print overflow

 c) else

 i) Read nw → info

 ii) nw → link = null.

Step4: [CODE FOR POSITION ONE]

if (pos = 1) then :

 a) if (start = null) then

 i) nw → link = nw

 ii) start = nw.

[End if]

 b) else

 i) fun(temp = start , temp → link != start ; temp = temp → link);

 ii) temp → link = nw

 iii) nw → link = start

 iv) start = nw

[End of else]

LectureNotes.in
[End of step 4 if structure]

Step 5 : [CODES FOR LAST POSITION]

```
else if ( pos == c+1 ) then
    i) if ( start == null ) then
        a) nw → link = nw
        b) start = nw
    [End if]
    ii) else
        a) for ( temp = start ; temp → link != start ; temp = temp → link )
            [End for]
        b) temp → link = nw ;
        c) nw → link = start ;
    [End of else]
[End of step 5 if ]
```

Step 6 : [CODES FOR OTHER POSITION]

```
else
    i) i = 1 .
    ii) temp = start ;
    iii) for ( ; i < pos - 1 ; temp = temp → link , i++ ) ;
        [End for]
    iv) nw → link = temp → link .
    v) temp → link = nw .
[End of else]
[End of step 6 else]
```

Step 7 : Exit

C - procedure :

```
Void insert-at-any-pos( )
{
    // info, start, nw, link are declared as global variable .
    Struct node *temp;
    int i, c, pos;
```

```

printf("Enter the position where the node is to be inserted");
scanf("%d", &pos);

temp = start;
do {
    c = c + 1;
    temp = temp->link;
} while (temp != start);

if (pos > c + 1)
{
    printf("Invalid position");
    exit();
}
else
{
    nw = (struct node *) malloc ( sizeof ( struct node));
    if (nw == '\0')
    {
        printf ("overflow");
        exit();
    }
    else
    {
        printf("Enter the information part of the node");
        scanf ("%d", &nw->info);
        nw->link = '\0';
    }
}

if (pos == 1)
{
    if (start == '\0')
    {
        nw->link = nw;
        start = nw;
    }
    else
    {
        for (temp = start; temp->link != start; temp = temp->link)
        ;
        temp->link = nw;
        nw->link = start;
    }
}

```

```

        start = nw ;
    }

}

else if ( pos == c+1 )
{
    if ( start == '\0' )
    {
        nw->link = nw;
        start = nw; n
    }
}

else
{
    for( temp = start ; temp->link != start ; temp = temp->link )
    {
        ;
        temp->link = nw;
        nw->link = start;
        start = nw;
    }
}

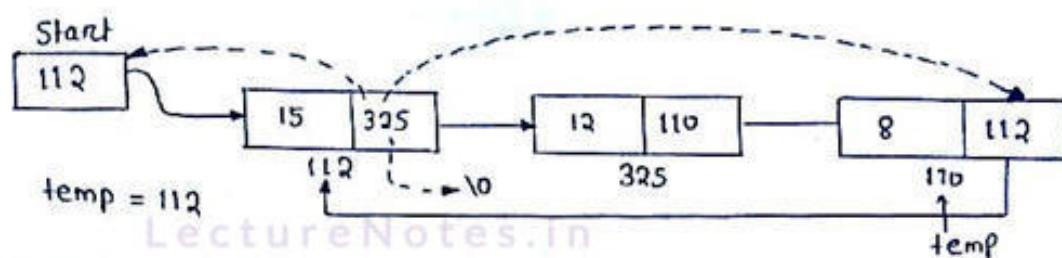
else
{
    i = l;
    temp = start;
    for( ; i < pos-1 ; temp = temp->link, i++ );
    nw->link = temp->link;
    temp->link = nw;
}
}
}

```

NOTE: Codes for insertion at last position will be satisfied by
 Codes for insertion before a given position as well as
 insertion after a given position .

Deletion in Single Circular link list :

Deletion Of First Node :



Algorithm :

PROCEDURE delete-first-node (list, start, link, temp)

- // list : Existing single circular link list .
- // start : pointer variable which holds the address of first node .
- // link : pointer variable which hold the address of next node .
- // temp : Temporary pointer variable used for traversing .

Step1 : if (start = null) then

- i) print the list is empty .
- ii) exit .

[End if]

Step2 : else if (start → link = start) then :

- i) temp = start
- ii) start → link = null
- iii) start = null .
- iv) free (temp).

Step3 : else

- i) temp = start
- ii) while (temp → link ≠ start)
- a. set temp = temp → link .
- [End while]
- iii) temp → link = start → link
- iv) temp = start
- v) start = start → link

vi) $\text{temp} \rightarrow \text{link} = \text{null}$.
vii) $\text{free}(\text{temp})$.
[End of else]
[End of Step 1 if structure]

Step 4: Exit

C - procedure :

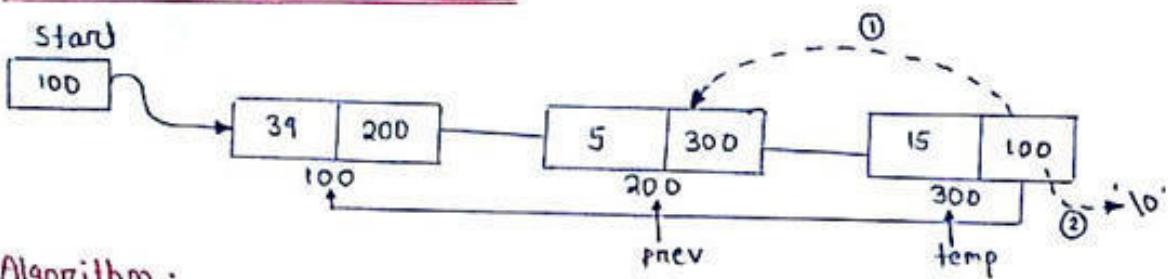
```
void delete-first-node()
```

{
!! start, link are declared as a global variable.

```
struct node *temp;  
if (start == '\0')  
{ printf("Underflow");  
exit();}  
else if (start->link == start)  
{  
temp = start;  
start->link = '\0';  
start = '\0';  
free(temp);}
```

```
else {  
temp = start;  
while (temp->link != start)  
{  
temp = temp->link;  
}  
temp->link = start->link;  
temp = start;  
start = start->link;  
temp->link = NULL;  
free(temp);  
}
```

Deletion of Last Node :



Algorithm :

PROCEDURE Delete-last-node (list , start , link , temp , pnext)

- // list : Existing single circular link list
- // start : pointer variable which holds the address of 1st node .
- // link : pointer variable which holds the address of next node .
- // temp : temporary pointer variable used for traversing .
- // pnext : pointer variable which holds the address of last but one node .

Step1 : if (start = null) then :

 1) print underflow

 2) exit .

[End if]

Step2 : else if (start → link = start)

 1) temp = start

 2) temp → link = '10'

 3) start = '10'

 4) free (temp)

Step3 : Else

 i) for (temp = start ; temp → link != start ; temp = temp → link)
 a) pnext = temp .

[End for]

 ii) pnext → link = temp → link

 iii) temp → link = '10'

 iv) free (temp)

Step4 : Exit

C - procedure :

```
void delete - last - node ()  
{  
    // start , link are declared as global variable .  
    struct node * temp , * pprev ,  
    if ( start == NULL )  
    { printf ("Underflow");  
        exit ();  
    }  
    else if ( start -> link == start )  
    {  
        temp = start ,  
        temp -> link = '\0' ,  
        start = '\0' ;  
        free ( temp );  
    }  
    else  
    {  
        for ( temp = start , temp -> link != start ; temp = temp -> link )  
            pprev = temp ,  
            pprev -> link = temp -> link ;  
            temp -> link = NULL ;  
            free ( temp );  
    }  
}
```



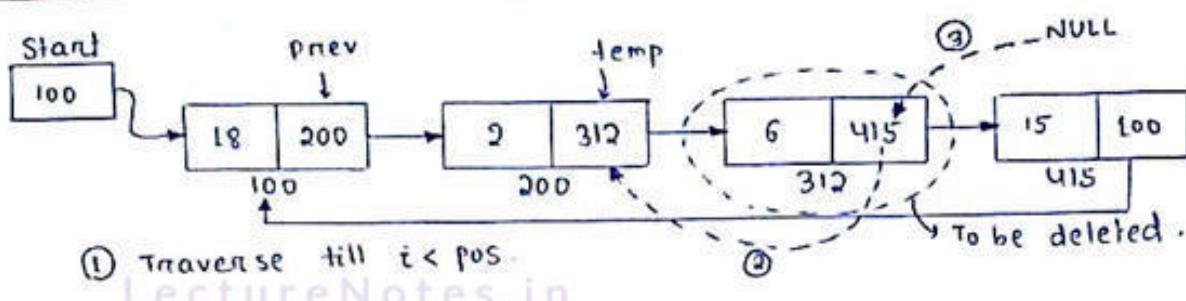
Data Structure Using C

Topic:

Deletion Of Node At Any Position

Contributed By:

Mamata Garanayak

Deletion of Node at any position :Algorithm:

```
PROCEDURE delete -of -node -at -any -pos ( list , start , next , prev , temp , i , c ,
                                             pos )
```

- // List : Existing Circular link list
- // start : pointer variable which holds the address of 1st node in the list
- // next : pointer variable which holds the address of prn next node
- // prev : pointer variable which holds the address of previous node
- // temp : temporary pointer variable used for traversing
- // i : Loop Counter variable
- // c : Counter variable used for counting the no of nodes
- // pos : position at which we want to delete the node.

Step1: if (start = null) then :

- i) print the list is empty
- ii) exit

Step2: else

- i) temp = start
- ii) do
 - a) $c++$
 - b) temp = temp \rightarrow link
 - c) while (temp != start)

[End of do-while loop]

Step3: if (pos > c)

- i) print invalid position .
- ii) exit

Step4: else if (pos = 1) then :

i) if ($\text{start} \rightarrow \text{link} = \text{start}$)

a) $\text{temp} = \text{start}$

b) $\text{start} = \text{null}$

c) $\text{temp} \rightarrow \text{link} = '10'$

d) $\text{free}(\text{temp})$

ii) else

LectureNotes.in

a) for ($\text{temp} = \text{start} ; \text{temp} \rightarrow \text{link} \neq \text{start} ; \text{temp} = \text{temp} \rightarrow \text{link}$) ;

 i) $\text{temp} \rightarrow \text{link} = \text{start} \rightarrow \text{link}$

 2) $\text{temp} = \text{start}$

 3) $\text{start} = \text{temp} \rightarrow \text{link}$.

 4) $\text{temp} \rightarrow \text{link} = \text{null}$.

 5) $\text{free}(\text{temp})$

Steps : else if ($\text{pos} = \text{c}$) then

 i) if ($\text{start} \rightarrow \text{link} = \text{start}$)

 a) $\text{temp} = \text{start}$

 b) $\text{temp} \rightarrow \text{link} = \text{null}$

 c) $\text{start} = \text{null}$

 d) $\text{free}(\text{temp})$

 ii) else

 a) for ($\text{temp} = \text{start} ; \text{temp} \rightarrow \text{link} \neq \text{start} ; \text{temp} = \text{temp} \rightarrow \text{link}$)

 i) $\text{prev} = \text{temp}$

 [End for]

 b) $\text{prev} \rightarrow \text{link} = \text{temp} \rightarrow \text{link}$

 c) $\text{temp} \rightarrow \text{link} = \text{null}$

 d) $\text{free}(\text{temp})$

Step 6 : else

 for ($\text{temp} = \text{start} ; i=1 ; i < \text{pos} ; i++ , \text{temp} = \text{temp} \rightarrow \text{link}$)

 i) $\text{prev} = \text{temp}$

 ii) $\text{prev} \rightarrow \text{link} = \text{temp} \rightarrow \text{link}$

 iii) $\text{temp} \rightarrow \text{link} = \text{null}$.

Step 7 : Exit iv) $\text{free}(\text{temp})$

C - procedure :

```
void delete-at-any-pos()
{
    // start, link are declared as global variable.

    int c, pos, c = 0;
    struct node *prev, *temp;

    printf (" Enter the position of node which is to be deleted");
    scanf ("%d", &pos);

    if (start == NULL)
    {
        printf (" List has no node");
        exit();
    }
    else
    {
        temp = start;
        do
        {
            c++;
            temp = temp->link;
        } while (temp != start);

        if (pos > c)
        {
            printf (" Invalid position");
            exit();
        }
        else if (pos == 1)
        {
            if (start->link == start)
            {
                temp = start;
                start = NULL;
                temp->link = NULL;
                free (temp);
            }
        }
    }
}
```

```

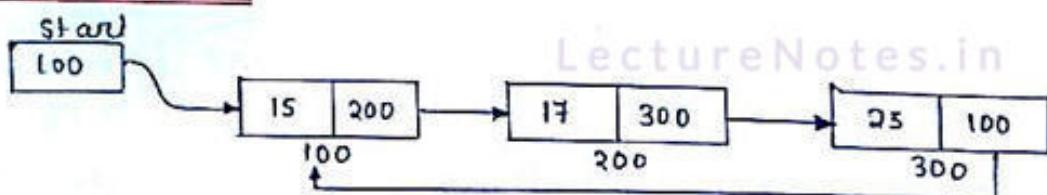
else
{
    for( temp = start ; temp->link != start ; temp = temp->link);
        temp->link = start->link;
        temp = start;
        start = temp->link;
        temp->link = NULL;
    } free(temp);
}

else
{
    for(i=1; temp = start; i<pos; i++, temp = temp->link)
        pprev = temp;
        pprev->link = temp->link;
        temp->link = NULL;
    } free(temp);
}

```

NOTE : Deletion of last node , code is not required as it is satisfied by the deletion-at-any-position .

SEARCHING :



Algorithm :

```

PROCEDURE searching-in-CL ( list, start, info, link, item, c, nd-node,
                           temp )
// list : Existing circular link list .
// start : list pointer variable which holds the address of 1st node.

```

// info : variable which holds the information part of a node
// link: pointer variable which holds the address of next node
// temp: temporary pointer variable used for traversing.
// item: variable which holds the value of item to be searched.
// c : Counter variable used for counting the no of node.
// nd-node : variable which will hold the no. of node at which item
is present.

Step1 : temp = start

Step2 : do

a. if $\text{temp} \rightarrow \text{info} = \text{item}$ then :

i) $c = c + 1$

ii) $\text{nd-node} = \text{nd-node} + 1$

iii) print item is found in ($\text{nd-node} + 1$).
[End if]

Step3 : $\text{temp} = \text{temp} \rightarrow \text{link}$

Step4 : while ($\text{temp} \neq \text{start}$); [End of do...while loop]

Step5 : if ($c = 0$) then

print searching is unsuccessful.

Step6 : Exit .

C-procedure :

```
void search()
{
    // start, link and info are declared as global variable.
    int item, c=0, nd-node=0 ;
    struct node * temp ;
    printf ("Enter the item for searching");
    scanf ("%d", &item);
```

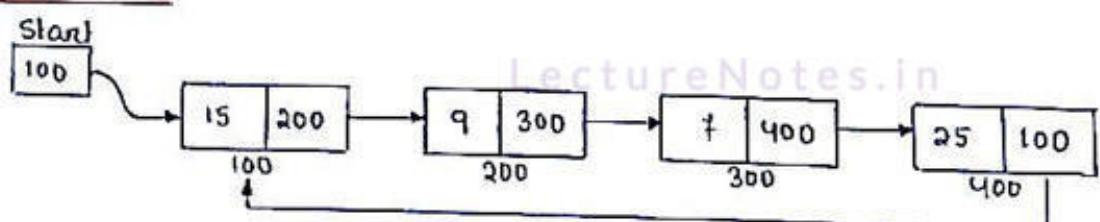
```

if ( start == NULL )
{
    printf (" Underflow");
    exit();
}
else
{
    temp = start;
    do
    {
        nd-node = nd-node + 1;
        if ( temp->info == item )
        {
            c = c + 1;
            printf (" Item is found in node %d", nd-node);
        }
        temp = temp->link;
    } while ( temp != start );
}

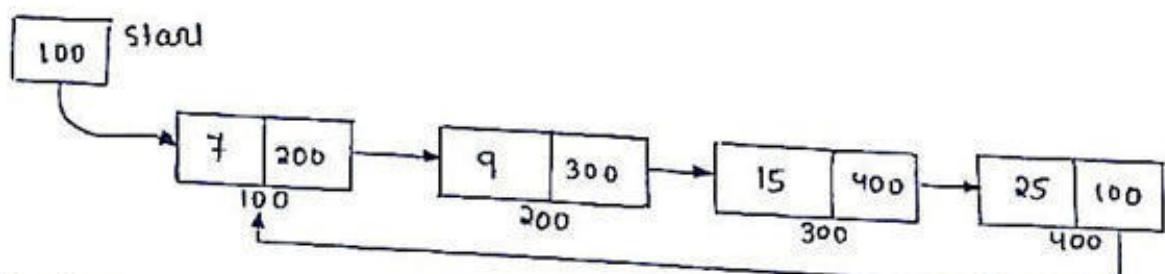
if ( c == 0 )
    printf (" Searching is unsuccessful");
}

```

SORTING :



After sorting, the list will be ;



Algorithm:

PROCEDURE sort (start, link, info, temp1, temp2, num)

// start : pointer variable

// link : pointer variable which holds the address of next node.

// info : variable which holds the information part of the node.

// temp1 : temporary pointer variable which holds the address of 1st node (initially) & used for traversing.

// temp2 temporary pointer variable which holds the address of 2nd node initially & also used for traversing.

// num : variable used for swapping two numbers

Step1 : if (start = null) then :

 1) print the list is empty.

 2) exit

Step2 : else

 1) for(temp1 = start , temp1 \neq start , temp1 = temp1 \rightarrow link)

 ii) for(temp2 = temp1 \rightarrow link ; temp2 \neq start , temp2 = temp2 \rightarrow link)

 a. if (temp1 \rightarrow info $>$ temp2 \rightarrow info)

 1) num = temp1 \rightarrow info

 2) temp1 \rightarrow info = temp2 \rightarrow info

 3) temp2 \rightarrow info = num

 [End if]

 [End of for loop ii]

 [End of for loop i]

Step3 : Exit

C - Procedure :

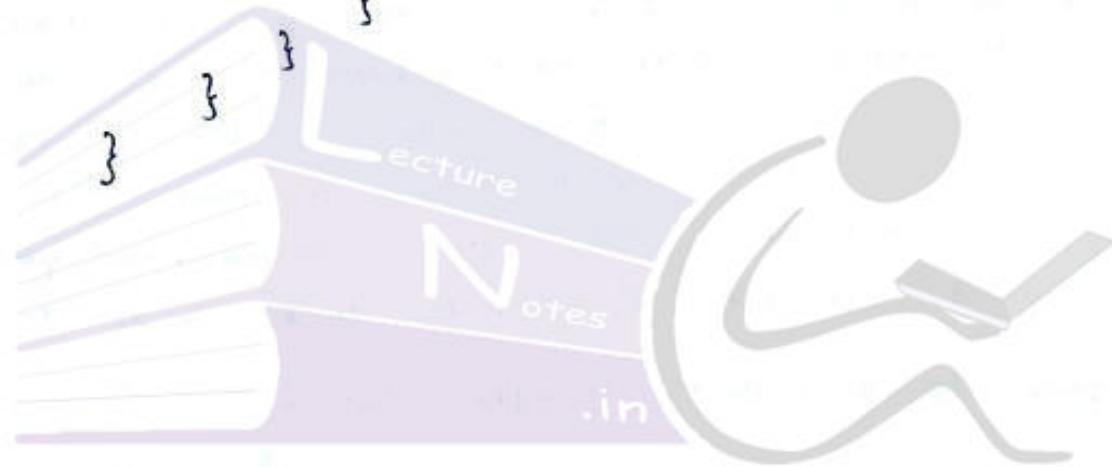
void sort()

{

// start, link, info are declared as global variables.

int num;

```
struct node *temp1, *temp2 ;  
for( temp1 = start ; temp1->link != start ; temp1 = temp1->link )  
{  
    for( temp2 = temp1->link ; temp2 != start ; temp2 = temp2->link )  
    {  
        if( temp1->info > temp2->info )  
        {  
            num = temp1->info ;  
            temp1->info = temp2->info ;  
            temp2->info = num ;  
        }  
    }  
}
```



LectureNotes.in

LectureNotes.in



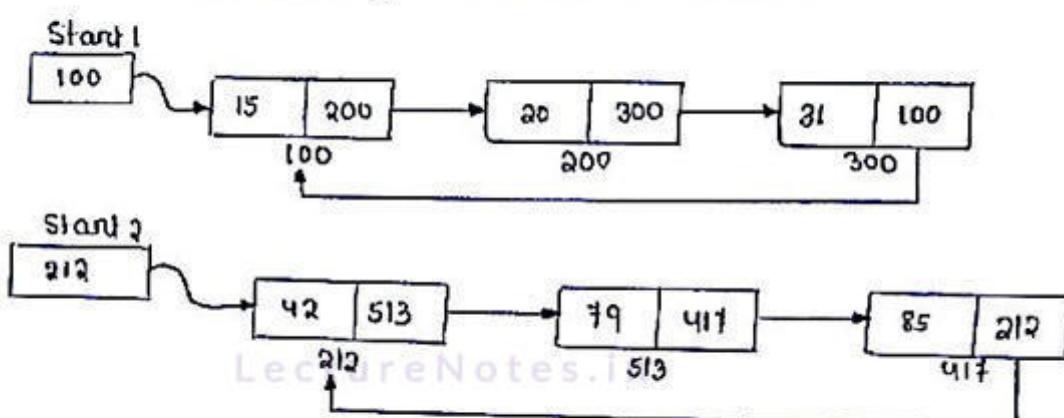
Data Structure Using C

Topic:

Concatenation In Single Circular Link List

Contributed By:

Mamata Garanayak

Concatenation in Single Circular Link list :Algorithm:

```
PROCEDURE Concatenate ( start1, start2, list1, list2, temp, link )
```

- // Start1 : pointer variable which holds the address of first node of list1.
- // Start2 : Pointer variable which holds the address of first node of list2 .
- # list1 : Existing 1st circular link list .
- // list2 : Existing 2nd circular link list .
- // temp : temporary pointer variable used for traversing .
- // link : pointer variable which holds the address of next node .

Step1 :

```
for ( temp = start1 ; temp->link != start1; temp = temp->link );
```


[End for].

Step2 :

```
temp->link = start2
```

Step3 :

```
for( temp = start1 ; temp->link != start2 ; temp = temp->link );
```


[End for]
i>

```
temp->link = start1
```


ii>

```
start2 = 'NULL'
```

.

Step4 : exit.

C - procedure :

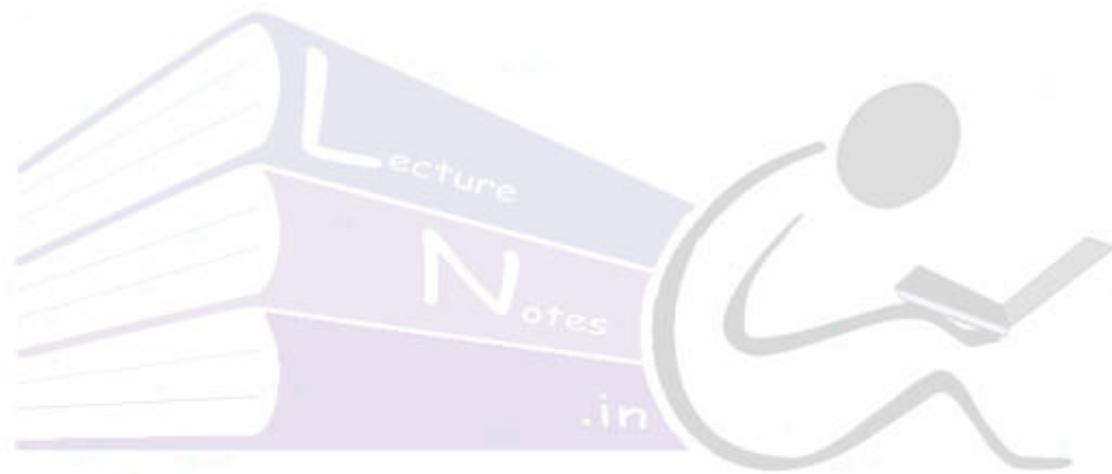
```
void Concatenation()
```

```
{
```

// start, link, list1, list2, temp are global variables.

```
for( temp = start1 ; temp->link != start1 ; temp = temp->link ) ;  
    temp->link = start2 ;  
  
for( temp = start1 ; temp->link != start2 ; temp = temp->link ) ;  
    temp->link = start1 ;  
    start2 = NULL ;  
}
```

LectureNotes.in



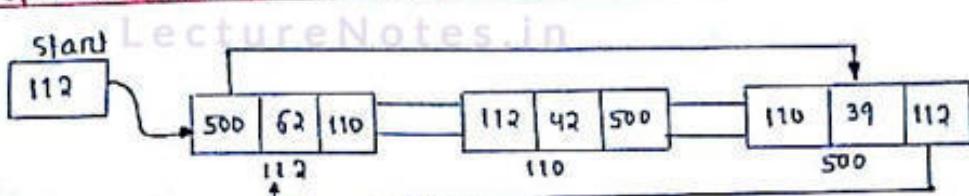
LectureNotes.in

LectureNotes.in

2) Double Circular link list :

In a double circular link list, the next pointer field of last node points to the first node instead of containing NULL pointer and previous pointer field of first node points to the last node instead of containing NULL pointer.

Representation of Double Circular link list :



Condition for overflow:

if ($nw == \text{NULL}$) then overflow.

Condition for Underflow:

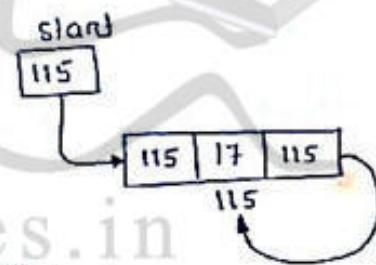
if ($start == \text{NULL}$) then underflow.

Single node condition:

if ($start == start \rightarrow \text{next}$) or

if ($start == start \rightarrow \text{prev}$) then

list contain a single node.



Traversing of Double circular link list:

Traversing of double circular link list is the process of visiting each node in the list exactly once. Traversing in double circular link list is classified into two types.

1) Forward Display | Traverse

2) Backward Display | Traverse.

- Forward display will display the information part of the list in forward manner i.e. from node 1 to last node sequentially.
- Backward display will display the information part of the list in backward manner i.e. from last node to 1st node sequentially.

Forward Display :

Algorithm :

PROCEDURE display-forward (start, info, next , temp, list)
// start : pointer variable which holds the address of 1st node .
// info : variable which holds the information part of the node .
// next : pointer variable which holds the address of next node .
// temp : temporary pointer variable used for traversing .
// list : existing double circular link list .

Step1 : if (start = null) then :

- 1) print the list is empty .
- 2) exit .

Step2 : else

A) temp = start

B) do

- i) print temp → info
- ii) temp = temp → next

c) while temp ≠ start repeat step B(i) to step B(ii)

[End of do-while loop]

[End of Step-1 if--else]

Step3 : Exit .

C - procedure :

```
void display-forward( )
{
    // start, info, next are declared as global variable .
    struct node * temp ;
    if ( start == NULL )
    {
        printf(" List is empty");
        exit();
    }
}
```

```

else {
    temp = start;
    do {
        printf ("%d", temp->info);
        temp = temp->next;
    } while (temp != start);
}

```

Backward Display:

Algorithm:

PROCEDURE display-Backward (start, info, next, list, temp, prev, temp1)

// start : pointer variable which holds the address of 1st node in the list.

// info : variable which holds the information part of node .

// next : pointer variable which holds the address of next node .

// list : Existing double circular link list .

// temp: temporary pointer variable used for traversing .

// temp1: temporary pointer variable used for traversing .

// prev : pointer variable which holds the address of previous node .

// temp1 : if (start = null) then :

- print the list is empty .
- exit .

Step2 : else

a) temp = start

b) while (temp->next ≠ start)

c) temp = temp->next .

[End of while]

c) temp1 = temp

d) do

- i) print temp → info
- ii) temp = temp → prev.

e) while temp ≠ temp₁ repeat step 2(d).

[end of do-while]

[end of if-else]

step 3: Exit

C-procedure:

```
void display-backward()
{
    // start, info, next and prev are declared as global variable.
    struct node *temp, *temp1;
    if (start == NULL)
    {
        printf ("List is empty");
        exit();
    }
    else
    {
        for (temp = start; temp->next != start; temp = temp->next)
            temp1 = temp;
        do
        {
            printf ("%d", temp->info);
            temp = temp->prev;
        } while (temp != temp1);
    }
}
```

NOTE: In circular link list, the last node can be accessed directly if we have the address of 1st node or the value of start. Also we can access the 1st node if the address of last node is known.



Data Structure Using C

Topic:
Header Link List

Contributed By:
Mamata Garanayak

HEADER LINK LIST :

- Definition: Header link list is a link list where a special node 'HEAD' is there, which keep information about the entire list.
- This header node is the 1st node of the list. Start keeps the address of head node.
- The informations kept by the head nodes are :
- How many nodes are there
 - Which type of list it is
 - What type of data are there in the list.

Types of Header Link List :

- 1) Single header link list
- 2) Double header link list.
- 3) Single header Circular link list
- 4) Double header Circular link list.

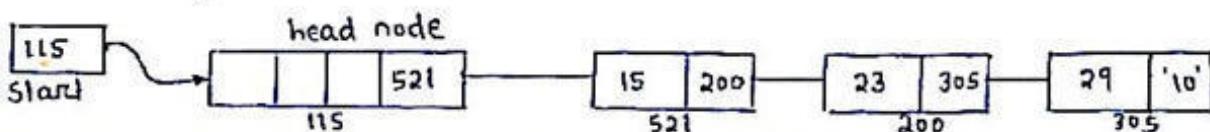
Declaration of Head node :

```
struct node
{
    int info;
    struct node *link;
} *start;
```

```
struct head
{
    int node-no;
    char type[30];
    char list-type[25];
    struct node *link;
};
```

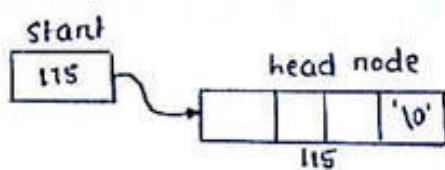
1) SINGLE HEADER LINK LIST :

The single link list containing head node as its first node is called single header link list.

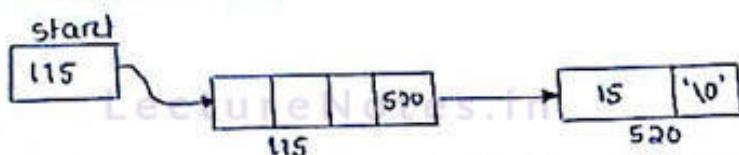


Condition for underflow:

if (`start->link == '\0'`)
then underflow on the list
is empty.



Condition for single node:



if (`start->link->link == '\0'`) then single node is there.

NOTE: To perform all the operations in single header link list, change the underflow condition, single node checking condition and set `temp = start->link` in place of `start`.

Creation of single header link list:

C - procedure :

```
void create-single-header-link-list()
{
    struct node *nw;
    struct head *h;
    int ch, c=0;
    do
    {
        nw = (struct node *) malloc ( sizeof ( struct node ) );
        if ( nw == '\0' )
            printf ("overflow");
            exit();
    }
    else
    {
        nw->link = NULL;
        printf ("Enter the information part");
        scanf ("%d", &nw->info);
        nw->link = start;
        start = nw;
    }
    c = c + 1;
}
```

```

printf (" Enter your choice \n 1. creation \n 2. stop \n");
scanf (" %d ", & ch);
} while ( ch != 0 );
h = ( struct head * ) malloc ( sizeof ( struct head ) );
if ( h == '10' )
{
    printf (" Malloc cannot allocate memory for head node ");
    exit ();
}
else
{
    h->type = "integer";
    h->node-no = c;
    printf (" Enter the type of the list ");
    scanf (" %s ", & h->list-type );
    h->link = start;
    start = h;
}
}

```

Algorithm:

PROCEDURE create-SHLL (nw, h, link, info, start, c, type, node-no, list-type)

- // nw : newly created node . or pointer variable which holds the address of newly created node.
- // h : pointer variable which holds the address of header node .
- // link : pointer variable which holds the address of next node in the list.
- // start : pointer variable which holds the address of 1st node in the list .
- // c : Counter variable used for counting the no. of nodes .
- // type : variable which holds the type of the list (int , float or character)
- // node-no : variable which holds the no. of nodes in the list .
- // list-type : variable which holds the types of list (single header link list or single circular header link list etc.) .

Step 1 : Set $c = 0$.

Step 2 : do

i) $nw = \text{allocate memory}$

ii) if ($nw = \text{null}$) then :

a) print overflow

b) exit .

iii) else

1) $nw \rightarrow \text{link} = \text{null}$

2) Read $nw \rightarrow \text{info}$

3) $nw \rightarrow \text{link} = \text{start}$

4) $\text{start} = nw$

5) $c = c + 1$.

Step 3 : print Enter your choice .

1. Creation

2. Stop .

Step 4 : while ($ch \neq 0$) .

[End of step-2 do...while loop]

Step 5 : $h = \text{allocate a memory}$.

Step 6 : if ($h = \text{null}$) then :

i) print malloc can't allocate memory

ii) exit .

Step 7 : else

i) $h \rightarrow \text{type} = \text{"integer"}$

ii) $h \rightarrow \text{node-no} = c$

iii) Read $h \rightarrow \text{list-type}$

iv) $h \rightarrow \text{link} = \text{start}$

v) $\text{start} = h$

[End of step-6 if...else structure].

Step 8 : Exit

Traversal of single header link list :

Visiting all the elements in the list exactly once is called traverse.

Algorithm :

PROCEDURE Traversal (start , link , list , temp , info)

// start : pointer variable which holds the address of first node .

// link : pointer variable which holds the address of next node .

// list : Existing single header link list .

// temp : Temporary pointer variable used for traversing .

// info : variable which holds the information part of each node .

Step1 : if (start → link = null) then :

i) print the list is empty .

ii) exit .

Step2 : else

i) temp = start → link .

ii) while temp ≠ null

a) print temp → info

b) temp = temp → link

[End of while]

[End of if..else loop]

Step3 : exit .

C - procedure :

void traversal()

{

// start , link and info are global variables .

struct node * temp ;

if (start → link == NULL)

{

printf(" list is empty ");

exit();

}

else

{

for (temp = start → link ; temp != '10' ; temp = temp → link)

```

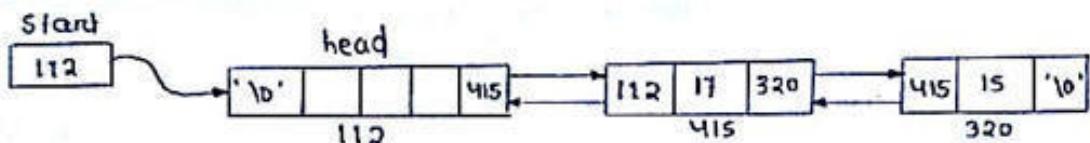
        printf ("%d", temp->info);
    }
}

```

> DOUBLE HEADER LINK LIST :

If the double link list contain the head node as its first node then it is called as double header link list.

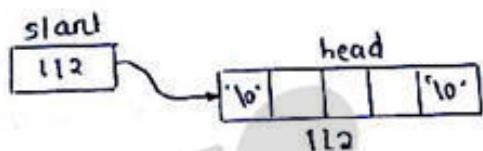
Representation of double header link list :



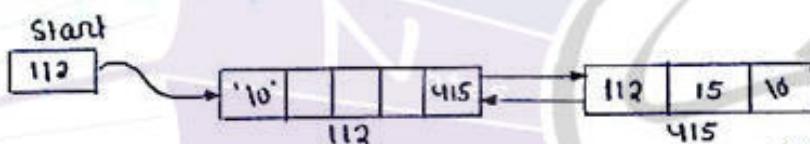
⑦

Underflow Condition :

if ($start \rightarrow next = null$) then underflow or list is empty.



Single node condition :



if ($start \rightarrow next \rightarrow next = null$) then single node is present in the list.

⑧

NOTE: For traversal operation change the underflow condition and replace the $temp = start$ by $temp = start \rightarrow next$.

Traverse :

Traversing means visiting all the data elements of the list exactly once.

→ Forward traversal is same as that of single header link list.

Backward Traversal :

Algorithm :

```

PROCEDURE traverse-backward (start, prev, next, list, info, temp)
// start : pointer variable which holds the address of 1st node .
// prev : pointer variable which holds the address of previous node .
// next : pointer variable which holds the address of next node .
// list : Existing double link list .

```

// info : information part of each node
// temp : temporary pointer variable used for traversing .

Step1 : if ($\text{start} \rightarrow \text{next} = \text{null}$) then :
 i) print the list is empty
 ii) exit .

Step2 : Else

 1) $\text{temp} = \text{start} \rightarrow \text{next}$
 2) while ($\text{temp} \rightarrow \text{next} \neq \text{null}$)
 i) $\text{temp} = \text{temp} \rightarrow \text{next}$
 [End of while]
 3) while ($\text{temp} \neq \text{start}$)
 i) print $\text{temp} \rightarrow \text{info}$
 ii) $\text{temp} = \text{temp} \rightarrow \text{prev}$
 [End of while]
 [End of if--else]

Step3 : Exit .

C - procedure :

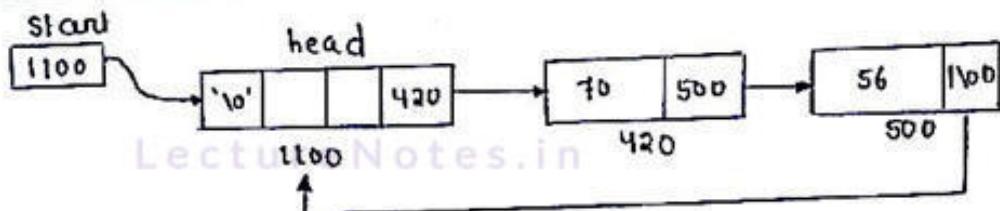
```
void traverse-backward()
{
    // start, next, prev, temp and info are global variables.

    if ( $\text{start} \rightarrow \text{next} == \text{NULL}$ )
    {
        printf("List is empty");
        exit();
    }
    else
    {
        for (temp = start  $\rightarrow$  next ; temp  $\rightarrow$  next != NULL ; temp = temp  $\rightarrow$  next)
            ;
        for ( ; temp != start ; temp = temp  $\rightarrow$  prev)
            printf(".i.d. temp  $\rightarrow$  info");
    }
}
```

3) SINGLE CIRCULAR HEADER LINK LIST :

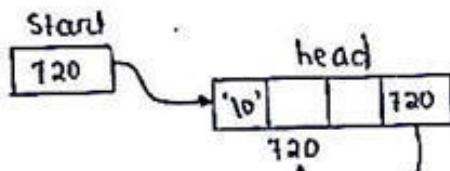
The single header link list in which the last node points to the first node is called single circular header link list.

Representation:

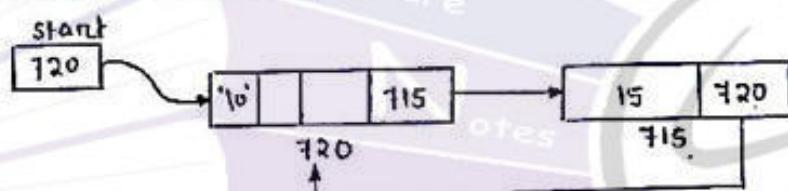


Underflow Condition :

if (start → link = start) then
underflow on the list is
empty .



Single node Condition :



if (start → link → link = start) then single node is present in the list .

Traverse :

Algorithm : PROCEDURE traverse (start , link , info , temp , list)
 // start : pointer variable which holds the address of 1st node .
 // link : pointer variable which holds the address of next node .
 // info : variable which holds the information part of each node .
 // temp : temporary pointer variable which is used for display .
 // list : Existing single circular link list .

Step 1 : if (start → link = start) then :
 i) printf the list is empty .
 ii) exit .

Step 2 : Else
 a) temp = start → link

b) while ($\text{temp} \neq \text{start}$)
 i) Print $\text{temp} \rightarrow \text{info}$
 ii) $\text{temp} = \text{temp} \rightarrow \text{link}$
 [End while]
 c) print the address of 1st node i.e. start .
 [End of if..else]

Step 3: Exit

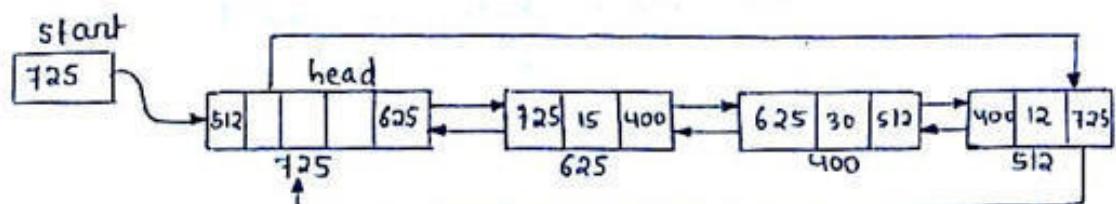
C- procedure :

```
void traverse()
{
    // start, link, temp, info are global variables
    if (start → link == start)
    {
        printf("The list is empty");
        exit();
    }
    else
    {
        for (temp = start → link; temp != start; temp = temp → link)
            printf("I.D", temp → info);
        printf("I.U", start);
    }
}
```

4) DOUBLE HEADER CIRCULAR LINK LIST :

If in double header link list last node's next part contain the address of head node and the previous part of the head node contain the address of last node then it is called a double header circular link list.

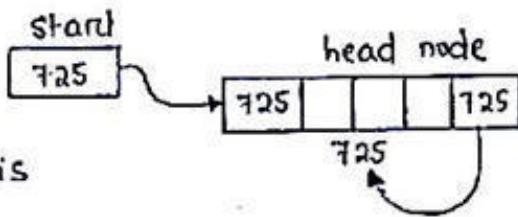
Representation :



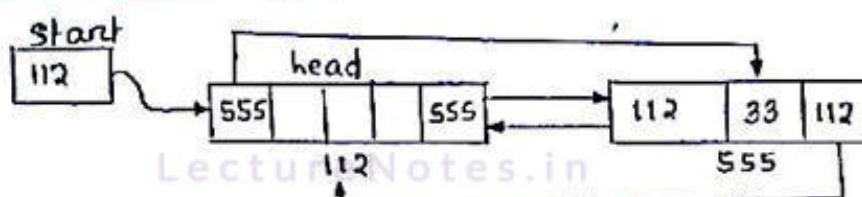
Underflow Condition :

if ($\text{start} \rightarrow \text{next} = \text{start}$)

then underflow on the list is
empty.



Single node Condition :



if ($\text{start} \rightarrow \text{next} \rightarrow \text{next} = \text{start}$) then a single node is present in the list.
or if ($\text{start} \rightarrow \text{next} = \text{start} \rightarrow \text{prev}$) then a single node is present in the list.

Traverse :

Forward traverse /

Algorithm : PROCEDURE Forward-traverse (list, start, next, temp, info)

// start : pointer variable
which holds the address of 1st node.

// list : Existing double header circular link list .

// next : pointer variable which holds the address of next node .

// temp : temporary pointer variable used for traversing .

// info : variable , which holds the information part of each node .

Step1: if ($\text{start} \rightarrow \text{next} = \text{start}$) then

i> print list is empty

ii> exit

Step2 : else

i> for (temp = start \rightarrow next ; temp != start ; temp = temp \rightarrow next)
 a> print temp \rightarrow info .

[End for]

Step3 : exit

C - procedure :

```
void forward-traverse()
{
    // start, info, next, temp are global variables.

    if (start → next == NULL)
    {
        printf ("The list is empty");
        exit();
    }

    else
    {
        for( temp = start → next ; temp != start ; temp = temp → next )
            printf ("%d", temp → info);
    }
}
```

Backward Traversal :

Algorithm :

```
PROCEDURE Backward-traverse (start, next, pprev, info, list, temp)

// start : pointer variable which holds the address of 1st node .
// next : pointer variable which holds the address of next node .
// pprev : pointer variable which holds the address of previous node .
// info : variable which holds the information part of each node
// list : existing double header circular link list .
// temp : temporary pointer variable used for traversing .

step1 : if (start → next == start) then :
        i) print the list is empty .
        ii) exit

step2 : else
        i) temp = start → next
        ii) while (temp → next ≠ start)
                temp = temp → next
                [End while]
        iii) while (temp ≠ start)
```

a) print temp → info
b) temp = temp → prev
[End of while]
[End of if .. else structure]

Step 3 : Exit

C - procedure :

```
void backward_traverse()
{
    // start, next, prev and temp are global.

    if (start → next == start)
    {
        printf("The list is empty");
        exit();
    }
    else
    {
        for (temp = start → next; temp → next != start; temp = temp → next)
            ;
        for ( ; temp != start; temp = temp → prev)
            printf(" + d", temp → info);
    }
}
```



Data Structure Using C

Topic:

Tree

Contributed By:

Mamata Garanayak

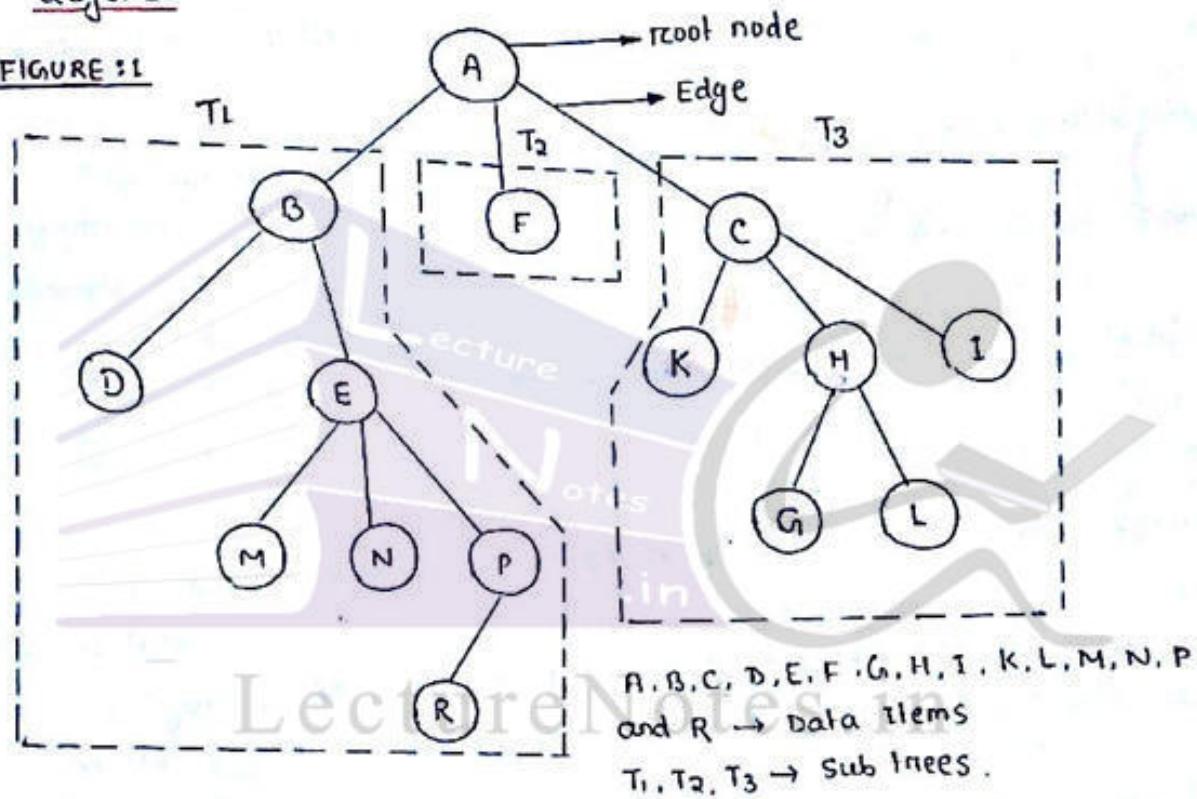
Lesson Number: 30

TREE :

Definition: Tree is a non-linear data structure where data items are arranged in hierarchical manner such that the first data item in this arrangement is called as root node and other remaining data items are divided into different subsets which are disjoint. (Disjoint means $A \cap B = \emptyset$). Those subsets are called as subtree.

→ In tree, all the data item should be connected but must be acyclic.

FIGURE 3.1



NOTE : Let F is connected to G, then

$T_3 \cap T_2 = G$ (not satisfied by definition of tree)

But $T_3 \cap T_2 = \emptyset$ must be satisfied by the definition of tree.

So tree must be acyclic.

TERMINOLOGY : A tree consists of ;

- | | | |
|-------------------|----------------------|--------------------|
| 1) Node | 6) Non-terminal node | 10) Level |
| 2) Edge | 7) Siblings | 11) Depth Height |
| 3) Degree of node | 8) Ancestors | 12) path |
| 4) Degree of tree | 9) Descendent | 13) Forest |
| 5) Terminal node | | 14) Root |

Node :

The individual data items in a tree is called node. In FIGURE:1 A, B, C, D, E, F, G, H, I, K, L, M, N, P, R are called nodes.

Root :

In an hierarchical arrangement, the first data item is called the root node. In FIGURE:1 (A) is the root node.

EDGE :

By means of which one data item is connected with its successive data item is called edge.

Degree of Node :

The number of child node present for a given node is called the degree of node.

Example : In FIGURE:1 if (B) is the parent node then (E) and (D) are 2 children node of (B). So the degree of node (B) is 2. similarly the degree of node (R) is 0 and the degree of node (H) is 2.

Degree of Tree :

It is the maximum or highest no. of degree of any node in a given tree.

In figure 1 the degree of tree is 3. because here the maximum or the highest no. of degree of any node (i.e. node A and node E) is 3.

Terminal Node :

The nodes having '0' degree are called as terminal or external node.

In figure 1, nodes D, F, G, I, K, L, M, N and R are terminal or external nodes.

Non-Terminal Node :

The nodes having degree > 0 are called as non-terminal nodes or internal nodes or intermediate nodes.

In figure 1 nodes B, E, P, C, H are non-terminal nodes.

NOTE: Root node is not considered as an non-terminal node or internal node or as an intermediate node even if it has degree > 0 .

Sibling :

For a given parent node, the relationship among the children nodes is called sibling relation that means they are the sibling of each other.

In figure 1 consider the node C.

For node C the node K, H, I are the children nodes. So,

K is the sibling of H and I.

I is the sibling of K and H.

H is the sibling of K and I.

Ancestor and descendant :

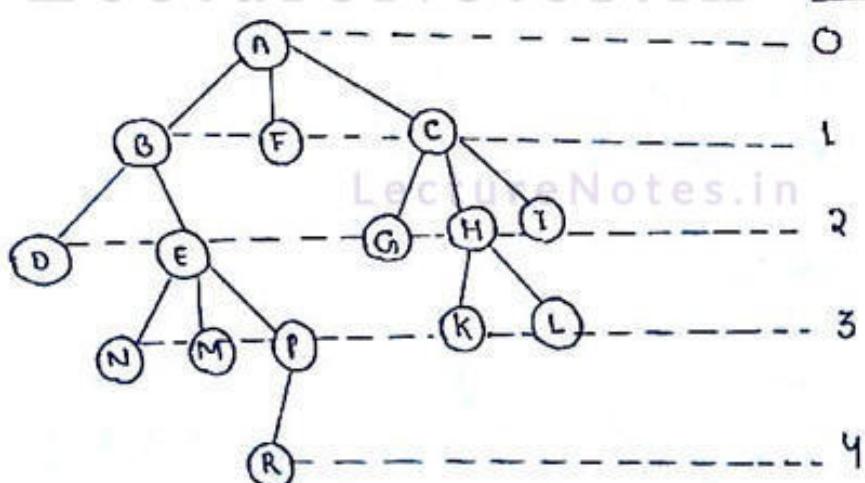
If two nodes say n_1 and n_2 are connected indirectly as given below then n_1 is called as ancestor of n_2 and n_2 is called descendant.



Level :

In an hierarchical arrangement, the root node is leveled as 0 and its descendant nodes are leveled as 1, 2, 3,

Level



NOTE: If the parent node is in the level n , then its children will be in the level $(n+1)$.

The level of node E is 2. So its children are - N, M, P will be in the level $(2+1) = 3$.

Depth Or Height :

Depth = (highest level + 1)

It is also called the height of the tree.

In figure 1 the depth or height = $4+1=5$,

Path :

Collection of edges sequentially from source node to destination node is called path.

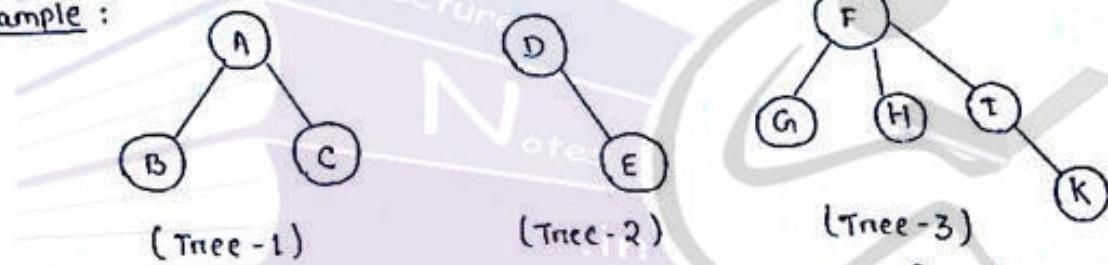
For example in figure 1 let \textcircled{A} is the source node.
 \textcircled{M} is the destination node.

\therefore path is $(A,B), (B,E), (E,M)$.

Forest :

Forest is the collection of disjoint trees.

Example :

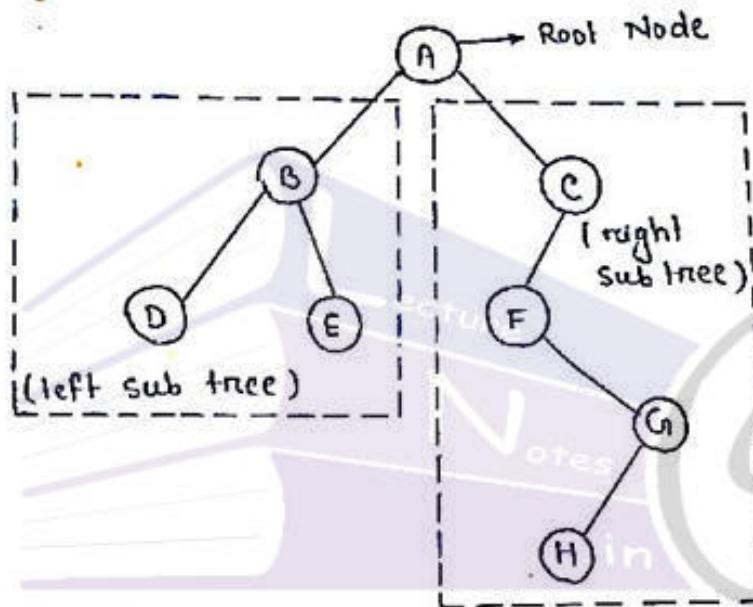


Tree-1, Tree-2 and Tree-3 are collectively called as forest.

Binary Tree :

Definition: It is a tree comprises of data item called as nodes are arranged such that :-

- 1) The tree 'T' may be empty or
- 2) There exist a root node 'R' and except this root node R all other nodes are divided into two sub trees called as left sub tree and right sub tree. Left sub tree may be empty or right sub tree may be empty.



[Binary Tree]

NOTE: In a binary tree, the degree of any node is at most 2.

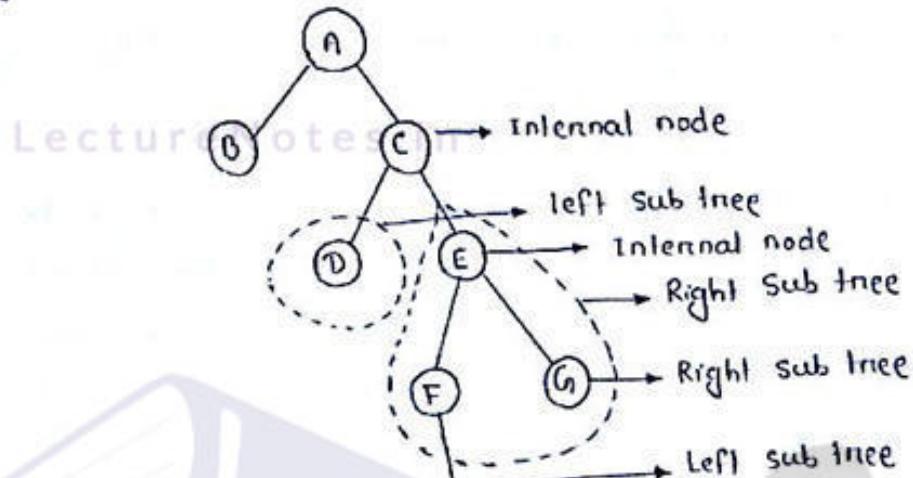
TYPES OF BINARY TREE :

Binary tree is classified into 4 types.

- Strictly binary tree .
- Complete binary tree .
- Almost complete binary tree .
- Extended binary tree or 2-tree .

Strictly Binary Tree :

In a binary tree, if each internal node have non-empty left sub tree as well as non-empty right sub tree then it is called strictly binary tree.



[Strictly Binary Tree] [Degree 2 or 0]

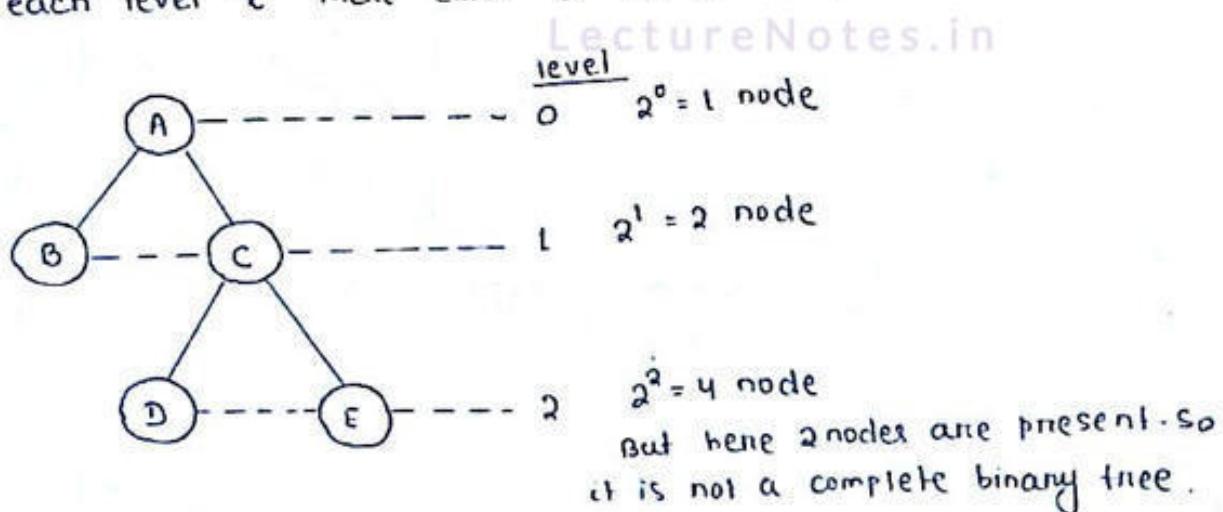
NOTE : If a strictly binary tree have ' n ' number of external nodes then the total no. of nodes for that tree is $2n-1$.

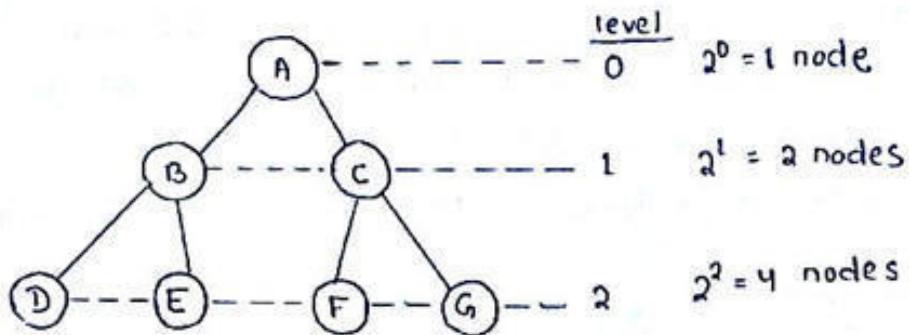
In the above figure no. of external nodes = 4

$$\begin{aligned} \text{So the total no. of nodes for that tree} &= 2 \times 4 - 1 \\ &= 8 - 1 \\ &= 7 \quad " \end{aligned}$$

Complete Binary Tree :

A binary tree will be called as complete binary tree if in each level ' l ' there exist 2^l no. of nodes.





so it is a complete binary tree.

NOTE : If a complete binary tree have highest level = l, then the total no. of nodes will be :

$$2^0 + 2^1 + 2^2 + \dots + 2^l \\ = \sum_{i=0}^l 2^i = \boxed{2^{l+1}-1}$$

[We know $S_n = \frac{a(r^n - 1)}{r-1}$]

where $a = 1st$ term

$n = no. of total terms$

$r = ratio$ between 2nd term & 1st term .

So in $2^0 + 2^1 + 2^2 + \dots + 2^l$

$a = 2^0 = 1$

$n = l+1$

$r = \frac{2^1}{2^0} = 2$ or $\frac{2^2}{2^1} = 2$ etc .

$\therefore S = \frac{a(r^n - 1)}{r-1}$ should become

$$\Rightarrow S = \frac{l(2^{l+1}-1)}{2-1}$$

$$\Rightarrow S = 2^{l+1}-1]$$

So in the above figure highest level = 2 .

.. The total no. of nodes = $2^{l+1}-1$

$$= 2^{2+1}-1$$

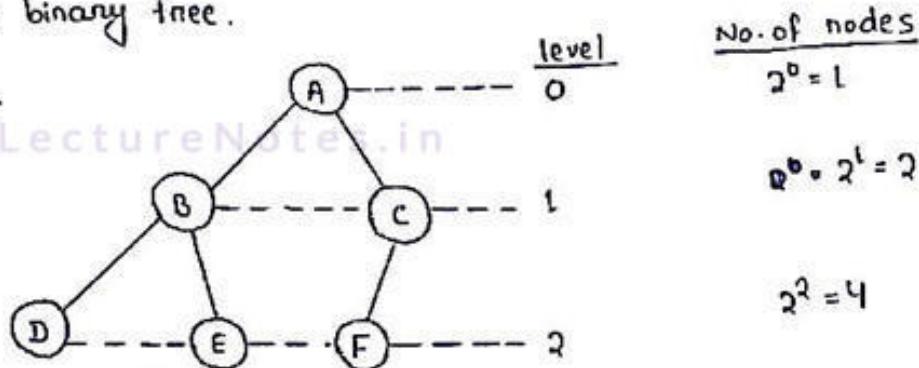
$$= 8-1$$

$$= 7 //$$

Almost Complete Binary Tree :

A Binary tree where $(\text{highest level} - 1)$ level is completed (i.e. it is having 2^{l-1} no. of nodes) and highest level may not be completed but it must be leftmost completed is called an almost complete binary tree.

Example :



In level 2, 4 nodes should be there, but it contains 3 nodes with leftmost completed.

$(\text{highest level} - 1) = 2 - 1 = 1$ is completed having 2 no. of nodes.
so it is an almost complete binary tree.

NOTE:

- For an almost complete binary tree

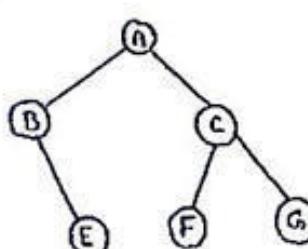
$$\text{Depth / Height} = \lfloor \log_2 t_n + 1 \rfloor$$

where $t_n = \text{total no. of nodes} = 2^{\lfloor \log_2 t_n \rfloor + 1}$

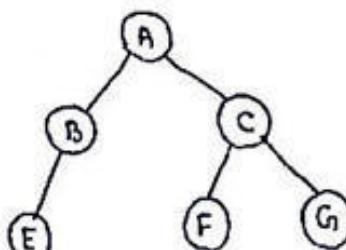
- For a complete binary tree

$$\text{Depth / Height} = \log_2 (t_n + 1)$$

where $t_n = \text{total no. of nodes} = 2^{l+1} - 1$.



[not an almost complete binary tree]



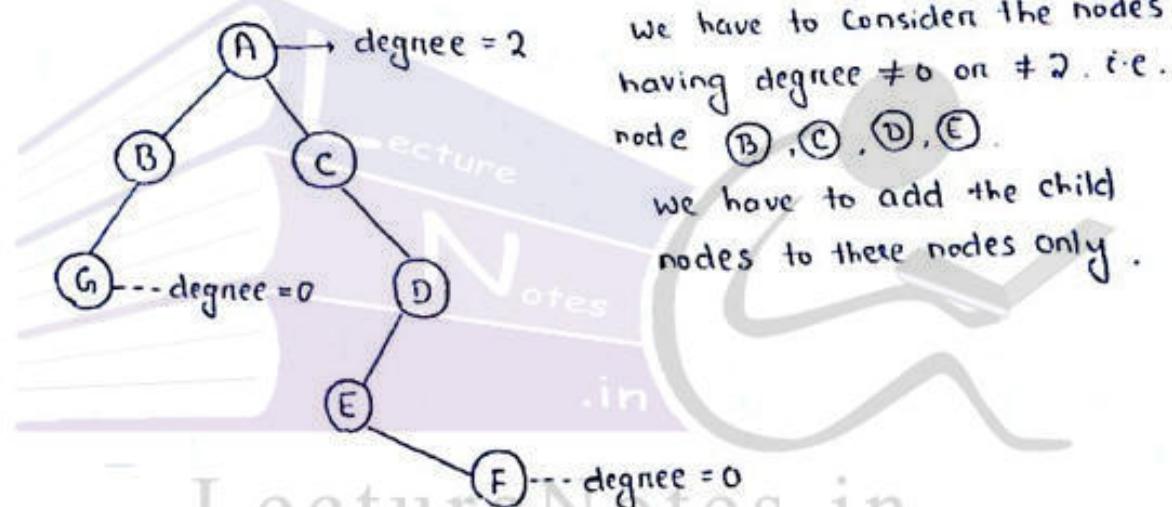
[not an almost complete binary tree]

Extended Binary Tree : (or 2-tree)

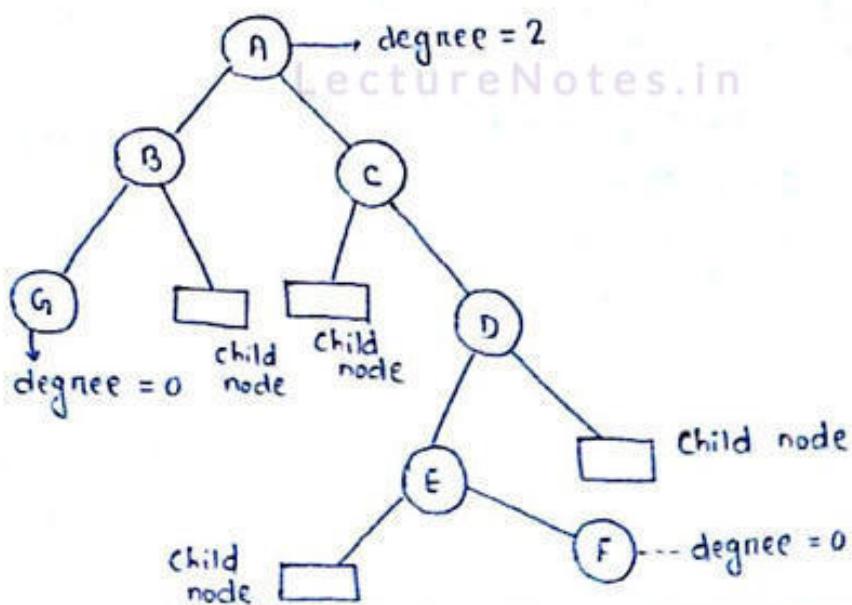
A binary tree where each node having degree 0 or 2 is called as extended binary tree or 2-tree.

Any binary tree can be converted into 2-tree through the following operations.

- 1) The nodes which are having degree 1 will be taken into account and a node as a child node will be added to this node.
- 2) The child node that has been added will be represented by a square (\square).



So the extended binary tree or 2-tree is ;





Data Structure Using C

Topic:
Representation Of Tree

Contributed By:
Mamata Garanayak

Lesson Number : 31

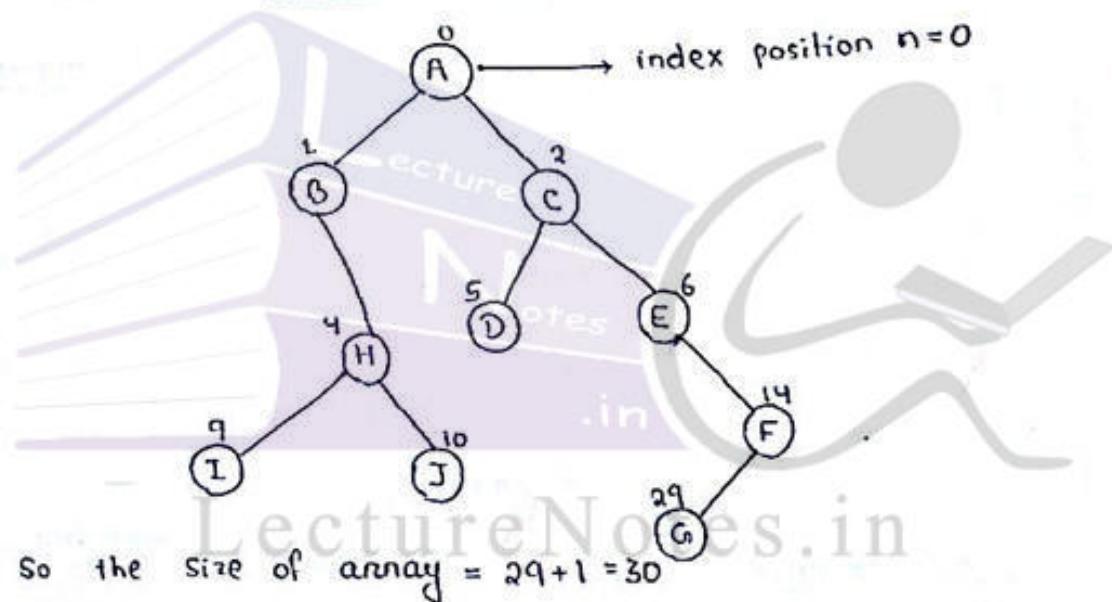
Representation of Tree :

Tree can be represented through

- i) Array representation or static representation .
- ii) Link list representation or dynamic representation .

i) Array Representation :

- i) In an array representation the data item in the root node will be stored in the index '0' in an array .
- ii) If a parent node have an index 'n' , then its left child will be stored in the index $n+1$ and the right child will be stored in the index $n+2$.



a	A	B	C		H	D	E		I	J	F	G
	0	1	2	3	4	5	6	7	8	9	10	11	12	29

Approximate Approximate Size of the array = $2^d - 1$
 $= 2^5 - 1 = 31$ approx.

Q. If the index of the right child is 30 then what is the index of parent node ?

Solⁿ. Index of right child = 30

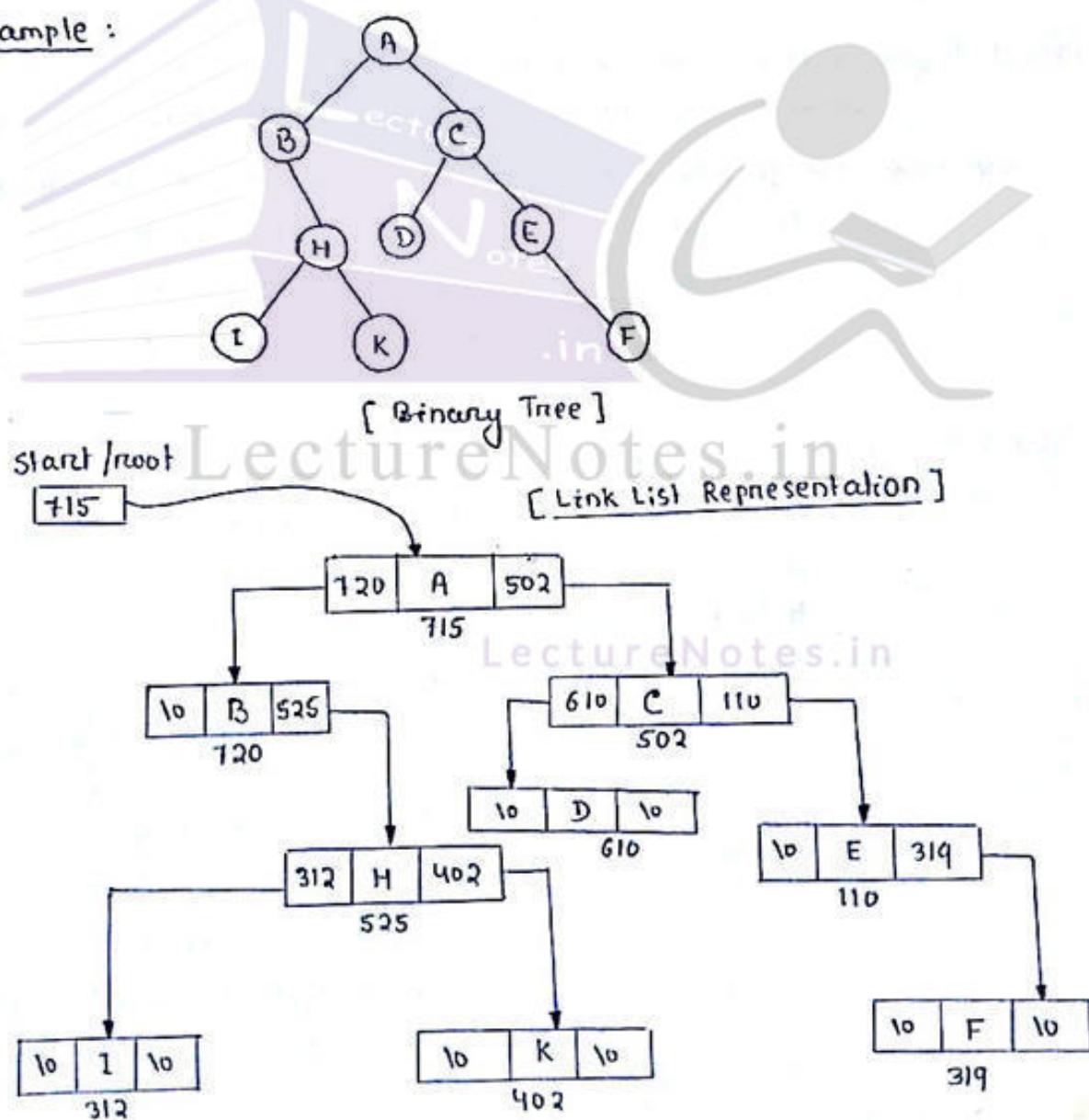
Given $2n+2 = 30$

$\Rightarrow n = 14$,

ii) Link List Representation :

- In a Link list representation, each data item will be stored in the form of nodes.
- Here each node is divided into 3 parts
 - 1) left-child → gt holds the address of left child of the parent node .
 - 2) Right-child → gt holds the address of right child of the parent node (next pointer field).
 - 3) Info → gt holds the information .
- start → gt holds the address of root node .

Example :



NOTE : If the tree is empty, then start will hold NULL.

Applications of Tree :

- Trees are used to implement file system of several operating systems (OS).
- Trees are used to evaluate arithmetic expressions.
- Trees are used to support searching operations in $O(\log n)$. average times and to refine ideas to obtain $O(\log n)$ worst case bounds and how to implement these operations when data are stored on a disk.

TRAVERSAL IN BINARY TREE :

The process of going through a tree in such a way that each node is visited once and only once is called tree traversal.

3 types of traversal can be performed in a binary tree. These are -

- 1) preorder (depth-first order) traversal
- 2) inorder (symmetric order) traversal
- 3) postorder traversal

1) Preorder Traversal :

If the root node will be traversed first followed by the left sub-tree and right subtree then it is called preorder traversal.

- Here
- 1) visit the root
 - 2) Traverse the left subtree in preorder
 - 3) Traverse the right subtree in preorder.

Recursive Algorithm :

A binary tree is in memory.
node holds the address of the root node of the tree.

Preorder (node)

Step 1. if (node ≠ NULL)

Step 2. process (node)

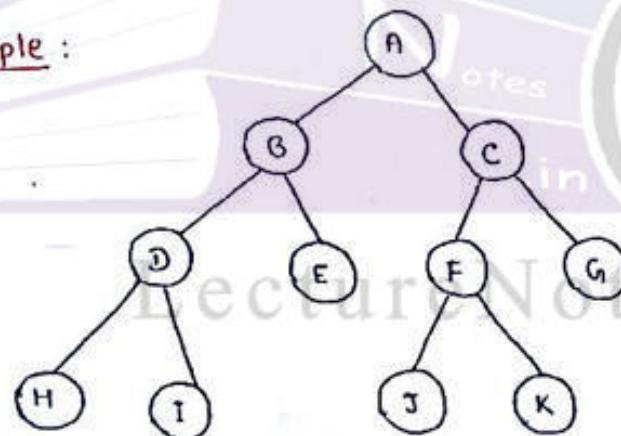
Step 3. preorden (left [node])
Step 4. preorden (Right [node])
[End of step 1 of structure]

Step 5. Exit.

Recursive C - Procedure :

```
void preorden ( struct node *P )
{
    // info, left, right are global variable .
    if ( P != NULL )
    {
        printf ("%d", p->info) ;
        preorden ( p->left ) ;
        preorden ( p->right ) ;
    }
}
```

Example :



preorden traversal = ABDHI_EC_FJKG

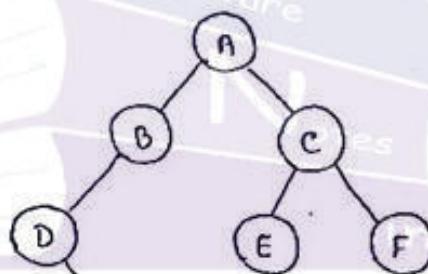
Non - Recursive Algorithm :

```
PROCEDURE preorden - traversal ( )
// .....
Step1 : set top = 0
stack[0] = NULL
node = root
Step2 : repeat the step 3 to step 5 while node ≠ NULL
```

Step3 : process (info[Node])
 [process the node]
 Step4 : if (right [node] ≠ NULL) then:
 i> top = top + 1
 ii> stack[top] = right [node]
 [End if]
 Step5 : if (left [node] ≠ NULL) then:
 i> set node = left [node]
 else
 i> node = stack[top]
 ii> top = top - 1
 [End of if .. else]
 [End of step-2 loop]

Step6 : Exit.

Example :

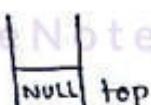


[Binary Tree]

- Steps
- 1> Initially push NULL on to stack.
set node = A

Node processed

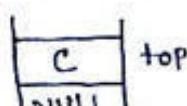
Stack



- 2> i> process A and push its right child on to stack .

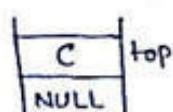
A

LectureNotes.in



- iii> process B

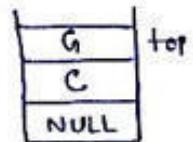
B



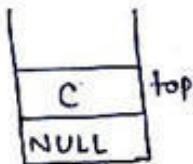
StepsNode processedStack

iii) process D and push its right child onto the stack

D

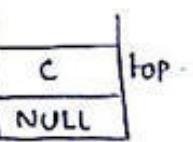


3) pop the top element from stack and set node = G.



4) since node ≠ NULL
process G.

G



No other node is processed. Since G has no child.

5) node = C



6) since node ≠ NULL

i) process C and push its right child onto stack.

C

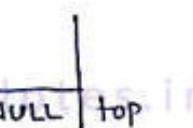


ii) process E (no right child exists) No other node is present.

E



7) node = F



8) since node ≠ NULL

F

i) process F
No other node is processed
since F has no child.



9) pop top element from stack and set node = NULL

stack empty

10) since node = NULL the algorithm terminated.

∴ preorder traversal = ABDGCEF

Non recursive C-procedure :

```
void preorden( Struct node *P )
{
    // info, left, right, node are global variable . top is a global variable ,
    // which is initialized to 0. stack is a global variable .

    stack[0] = NULL ;
    node = root ;
    while ( node != NULL )
    {
        printf ("f.d", p->info) ;
        if ( right[node] != NULL )
        {
            top = top + 1 ;
            stack[top] = right[node] ;
        }
        if ( left[node] != NULL )
        {
            node = left[node] ;
        }
        else
        {
            node = stack[top] ;
            top = top - 1 ;
        }
    }
}
```

2) Inorder Traversal :

If the root node will be traversed after the left subtree and before the right subtree then it is called inorder traversal.

- Here 1) Traverse the left subtree in inorder
- 2) visit the root
- 3) Traverse the right subtree in inorder

Recursive Algorithm:

A binary tree is in memory.

node holds the address of the root node of the tree.

inorder (node)

Step1 : if (node ≠ NULL)

Step2 : inorder (left [node])

Step3 : process (node)

Step4 : inorder (right [node])

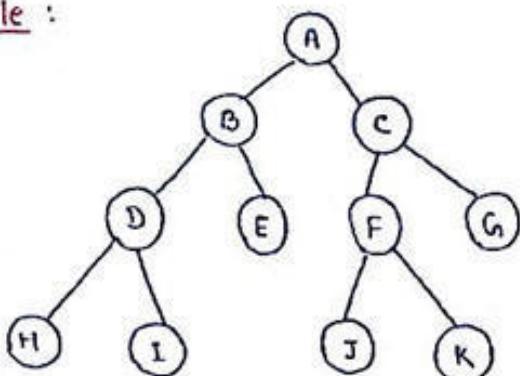
[End of step1 if]

Step5 : Exit

Recursive C - procedure :

```
void inorder ( struct node *P )
{
    if ( P != NULL )
    {
        inorder ( P->left );
        printf ( "%d", P->info );
        inorder ( P->right );
    }
}
```

Example :



Inorder traversal = HDIBEAJFKCG

Non-Recursive Algorithm :

PROCEDURE Inorder-traversal()

// ...

Step1 : set top = 0

stack[0] = NULL

node = root.

Step2 : while (node ≠ NULL)

i) top = top + 1

ii) stack[top] = node

iii) node = left[node]

[End while]

Step3 : node = stack[top]

top = top - 1

Step4 : Repeat the steps 5 to 7 while node ≠ NULL .

Step5 : process(info[node])

Step6 : if (right[node] ≠ NULL)

i) node = right[node]

ii) Go to step 2 .

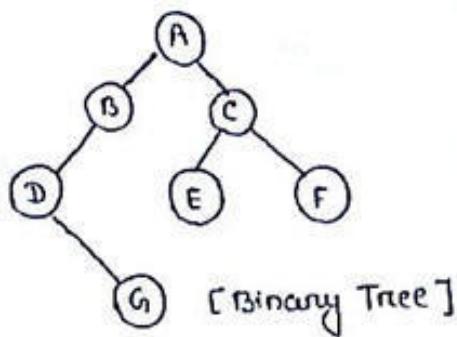
[End if]

Step7 : node = stack[top]

top = top - 1

Step8 : Exit [End of step-4 while]

Example :



Steps

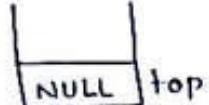
Node processed

Stack

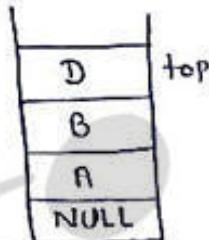
1) Initially push NULL

on to stack

set node = A

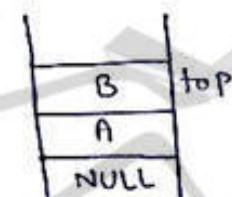


2) push node A,B,D
on to stack

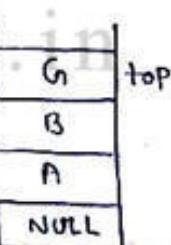


3) node = D
process D

set node = C

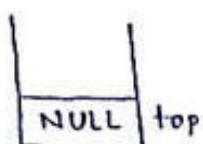


4) node = C, push the
node C onto the stack



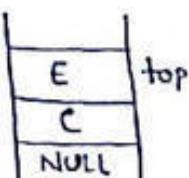
5) pop C,B,A and
process C,B,A.
(we stop at A since A
has right child)

G
B
A



set node = C

6) node = C . push node
C and E onto stack.



- | <u>steps</u> | <u>Node processed</u> | <u>stack</u> |
|--|-----------------------|---------------------|
| 1) pop E and C and process E and C.
set node = F. | E
C | |
| 8) node = F.
push the node F onto the stack. | | |
| 9) node = F , POP F and process F. | F | |
| 10) The next element NULL is popped and since node = NULL, the algorithm is terminated | |
stack is empty. |

Inorder traversal = DGBAECF

Non-recursive C - Procedure :

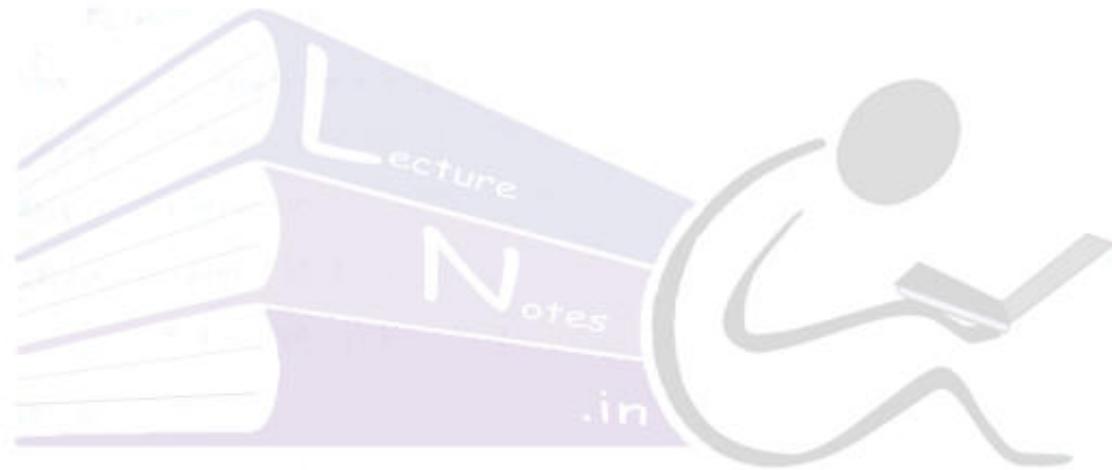
```

void inorder_traverse (struct node *P)
{
  // info, left, right, node, top=0, stack are global variable.
  stack[0] = NULL;
  node = root;
  while (node != NULL)
  {
    top = top + 1;
    stack[top] = node;
    node = left[node];
  }
  node = stack[top];
  top = top - 1;
}

```

```
while (node != NULL)
{
    printf ("%d", p->info);
    if ( right [node] != NULL)
    {
        node = right [node];
    }
}
```

LectureNotes.in



LectureNotes.in

LectureNotes.in



Data Structure Using C

Topic:
Post Order Traversal

Contributed By:
Mamata Garanayak

3) Postorder Traversal :

If the root node will be traversed at last, first left child, then right child then root node then it is called postorder traversal.

- Here
- 1) Traverse the left subtree in postorder
- 2) Traverse the right subtree in postorder
- 3) visit the root.

Recursive Algorithm:

A binary tree is in memory.

node holds the address of the root node of the tree.

postorder (node)

Step1 : if (node != null)

Step2 : postorder (left [node])

Step3 : postorder (right [node])

Step4 : process (node)

[End of Step 1 if]

Step5 : Exit.

Recursive C-procedure :

```
void postorder (struct node *P)
```

{

if (P != NULL)

{

postorder (P->left);

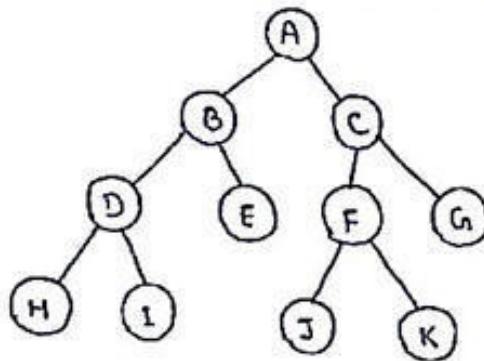
postorder (P->right);

printf ("%d", P->info);

}

}

Example :



postorder traversal = HIDEBJFKFGCA

Non-Recursive Algorithm :

PROCEDURE postorder-traversal()

// ...

Step1 : top = 0

stack[0] = NULL

node = root .

Step2 : Repeat the step3 to 5 while node ≠ null .

Step3 : i) top = top + 1

ii) stack[top] = node

Step4 : if (right [node] ≠ null)

i) top = top + 1

ii) stack[top] = -(right [node])

[End of if]

Step5 : node = left [node]

[End of Step2 while loop]

Step6 : i) node = stack [top]

ii) top = top - 1

Step7 : while (node > 0)

i) process (info[node])

ii) node = stack [top]

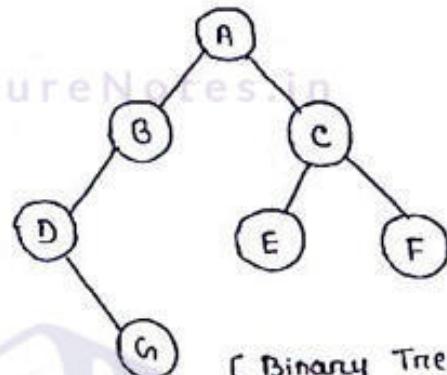
iii) top = top - 1

[End while]

Step 8 : if (node < 0)
 i) node = -node
 ii) Goto Step 2.
 [End of if]

Step 9 : Exit.

Example :



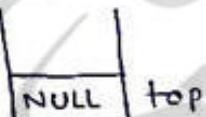
Steps

Node processed

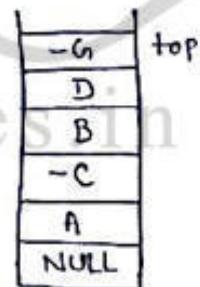
stack

1) Initially push NULL on to stack .

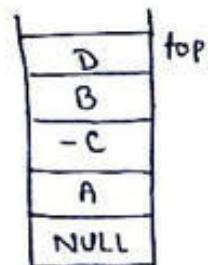
set node = A



2) By taking node = A ! proceed to left most path , on the way if right child is obtained push negative of that on to stack .



3) pop and process positive node if negative node is popped make it positive and repeat the step
 $-G$ is popped so
 node = $-G$
 Now reset node = G



Steps

- 4) By taking node = G proceed to left most path. push G on to stack.

Node processed

- 5) pop and process

G, D, and B. -C is popped so node = -C now reset node = C

G
D
B

stack

G	top
D	
B	
-C	
A	
NULL	

- 6) By taking node = C,

proceed to left most path if right child is obtained push negative of that on to stack. push C, -F and E.

- 7) pop and process E.

-F is popped. So, node = -F now reset node = F

E

E	top
-F	
C	
A	
NULL	

- 8) By taking node = F, popped proceed to left most path. push F onto stack.

- 9) pop and process F, C and A

F
C
A

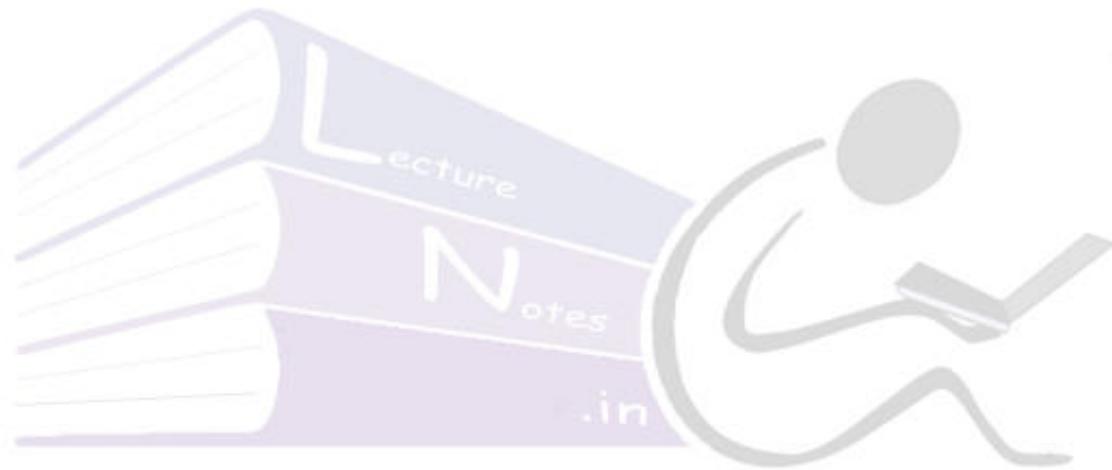
F	top
C	
A	
NULL	

- 10) The next element NULL is popped since node = NULL algorithm is terminated

stack empty.
postorder traversal = GDBEFCAB

Non- recursive C-procedure:

LectureNotes.in



LectureNotes.in

LectureNotes.in

Construction Of Binary Tree

A Binary tree can be constructed if

- 1) preorder and inorder traversal is given
- 2) postorder and inorder traversal is given
- 3) preorder and postorder traversal is given.

I) Construction of Binary tree from preorder and inorder traversal

Step1 : The first node in the preorder traversal is the root node of the tree.

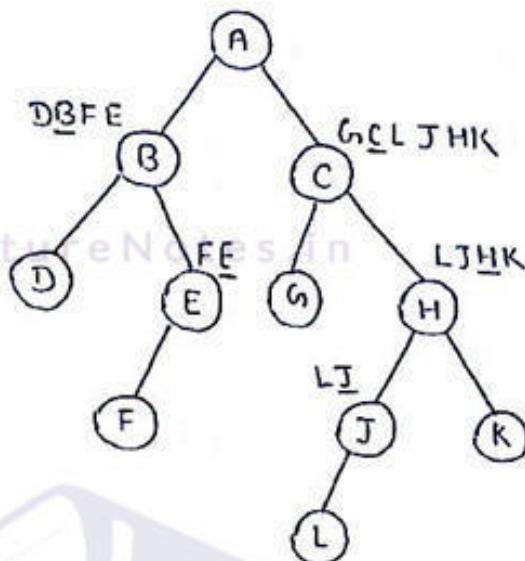
Step2 : Find the position of the root node in inorder traversal . The nodes precede to root node in the inorder traversal are the nodes of the left subtree of the root node, and the nodes in the right subtree succeed to the root node are the nodes in the right subtree of the root node.

Step3 : Now Consider two sets of inorder and preorder traversals of the left and right subtrees of the root . The first set is the nodes appear precede to the root node in inorder traversal and the combination of those nodes only appear in preorder traversal just after the root node . The second set is the nodes appear after the root node in inorder traversal and the nodes in the preorder traversal except the root node and the nodes considered in the first set .

Step4 : Taking these two sets of preorder and inorder traversals for left and right subtrees , repeat the steps 2 and 3 till the entire tree is constructed .

Example: preorder: $\xrightarrow{\text{ABDEF CGHJLK}}$

inorder: DBFEG \underline{A} GCLJHK



>> Construction of Binary tree from postorder and inorder traversal :

Step1: The last node in the postorder traversal is the root node in the tree.

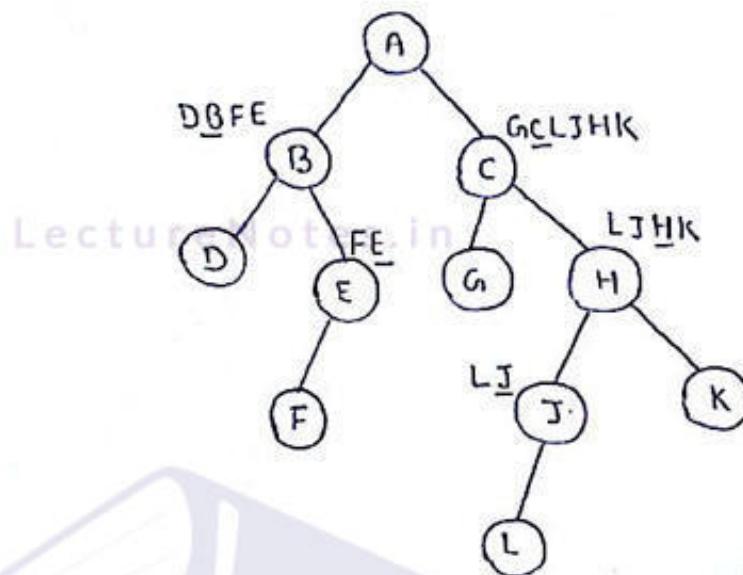
Step2: Find the position of the root node. The nodes precede to root node in the inorder traversal are the nodes in the left subtree of the root node and the nodes succeed to the root nodes are the nodes in the right subtree of the root node.

Step3: Now consider the two sets of inorder and postorder traversals of the left and right subtrees of the root. The first set is the nodes appear precede to the root node in inorder traversal and the sequence of same nodes that are in postorder traversal just before the root node. The second set is the nodes appear after the root node in inorder traversal and the sequence of nodes that are in the postorder traversal except the root node and the nodes considered in the first set.

Step4: Taking these two sets of postorder and inorder traversals for left and right subtrees repeat the Step 2 and 3 till the entire list is empty or tree is constructed.

Example : postorder : D F E B G L J K H C A

inorder : D B F E A G C L J H K



3) Construction of Binary tree from preoder and postorder traversal

Step1 : From preoder and postorder find the root node.

Step2 : Find the successor of root node from preoder traversal sequence .

Step3 : Find the predecessor of root node from postorder traversal sequence .

Step4 : If the successor of root node from preoder traversal is not equal to the predecessor of root node from postorder traversal then

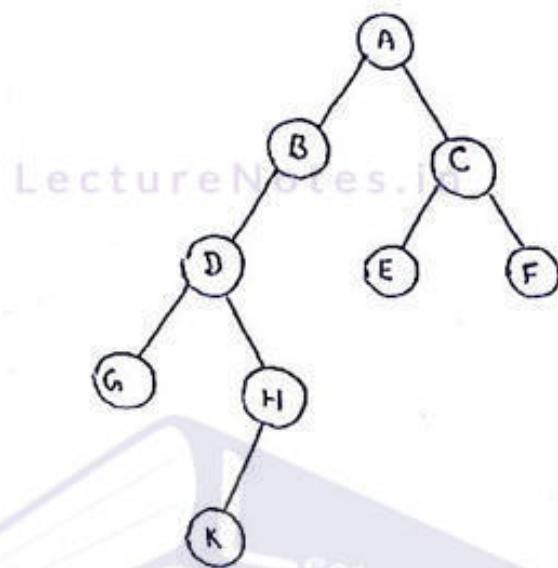
left child = successor of root node .

right child = predecessor of root node .

Step5 : If the successor of root node from preoder traversal is not equal to the predecessor of root node from postorder traversal then successor of root node may be a left child or right child and the predecessor of root node may also be a left child or right child .

Step 6 : Repeat the steps 2 to 5 until the entire tree is constructed.

Example : Preorder : A B D G H K C E F
Postorder : G H D B E F C A



Expression Tree :

Expression tree is a binary tree where each internal node is an operator and each external node is an operand.

→ Expression tree can be constructed from

- 1) Infix expression
- 2) Postfix expression
- 3) Prefix expression

I) Construction of an expression tree from an infix expression :

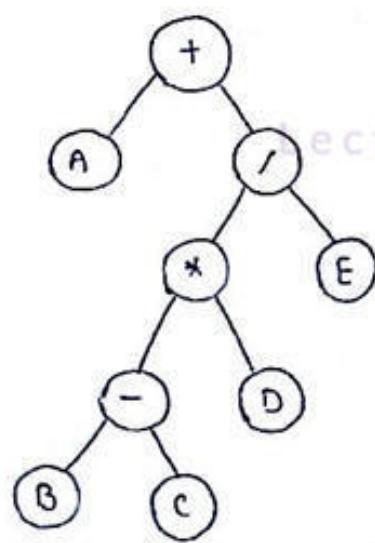
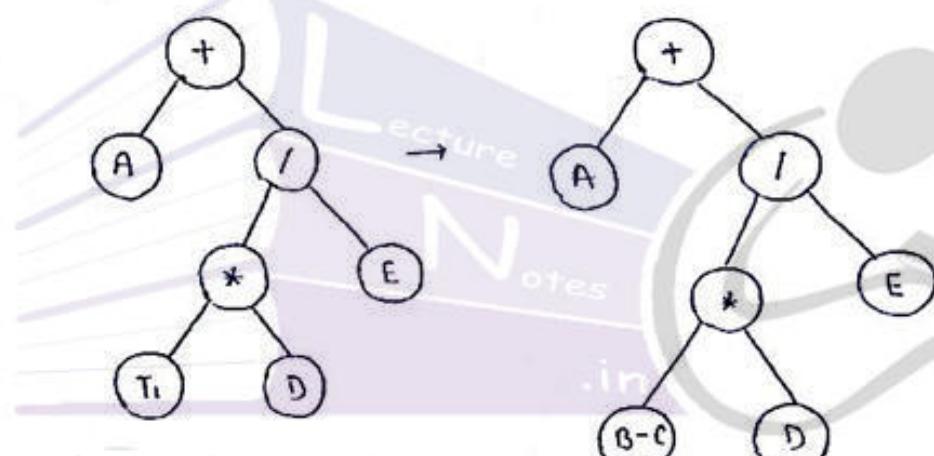
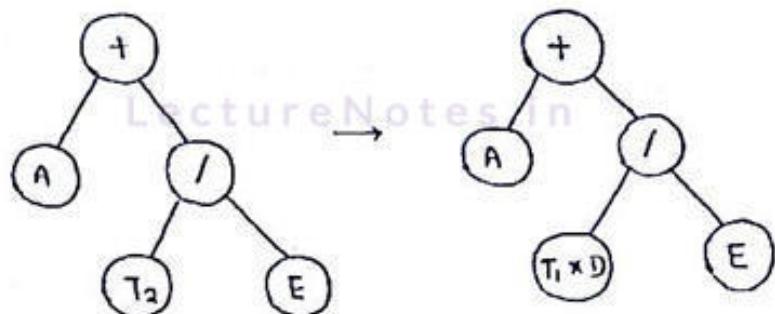
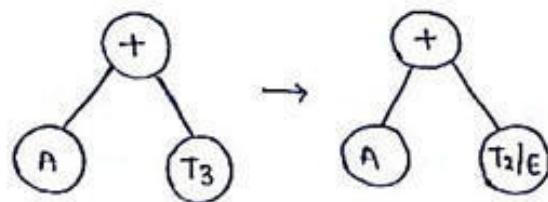
→ In infix expression, first find the operator. Its left operand is left subtree and right operand is right sub tree.

Example : $A + (B - C) * D / E \rightarrow$ Infix expression

$$= A + T_1 * D / E \quad (T_1 = B - C)$$

$$= A + T_2 / E \quad (T_2 = T_1 * D)$$

$$= A + T_3 \quad (T_3 = T_2 / E)$$

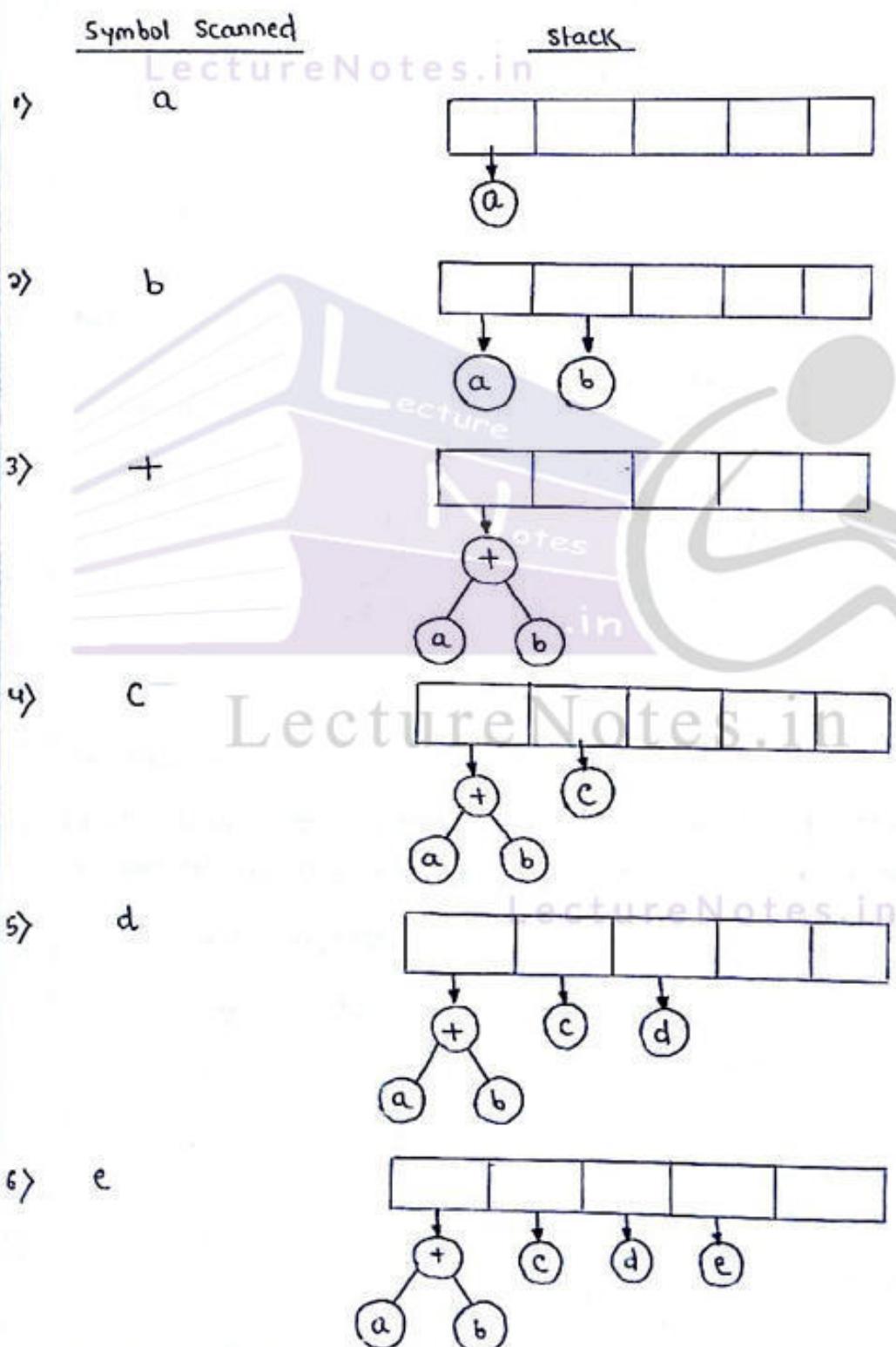


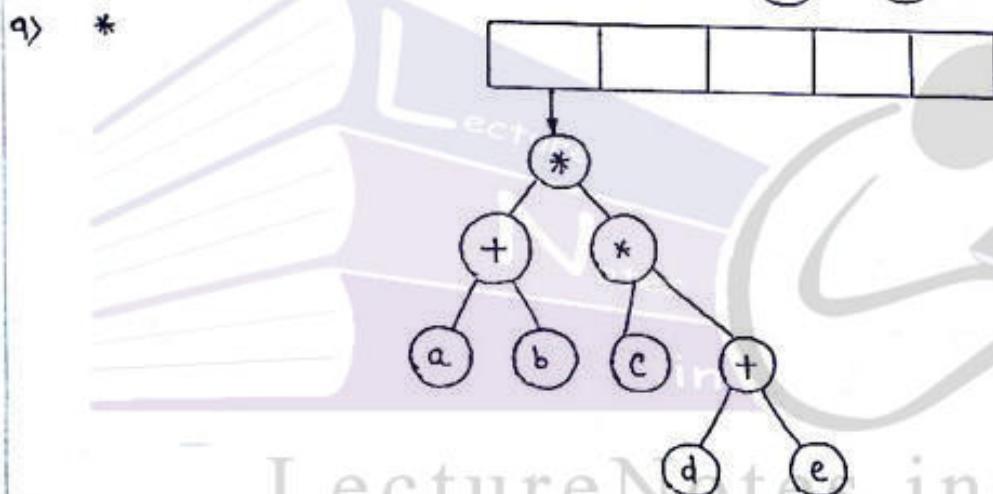
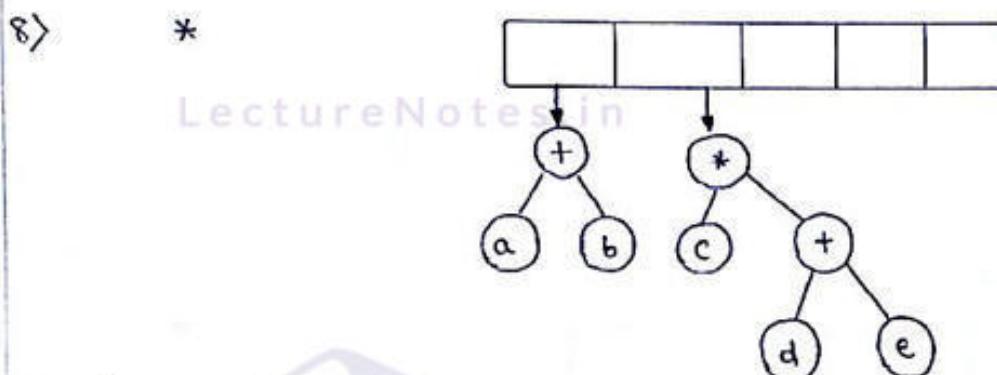
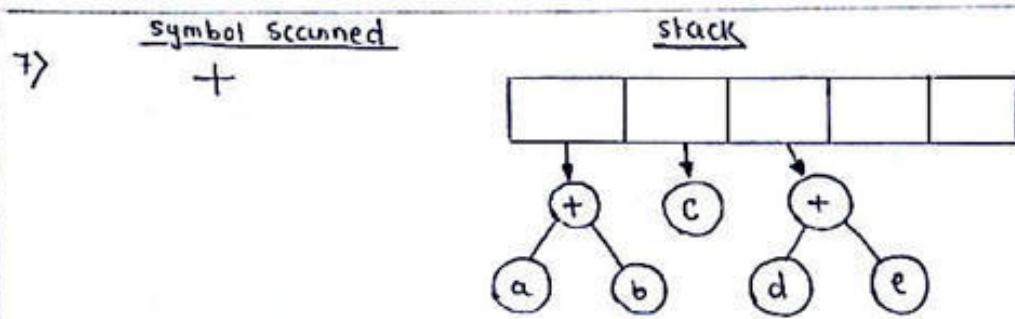
(Expression Tree)

2) Construction of an expression tree from postfix expression :

→ The algorithm for the construction of an expression tree from postfix expression is same as an algorithm to convert the infix to postfix . evaluate the postfix expression .

Example: ab + cde + * * → postfix Expression

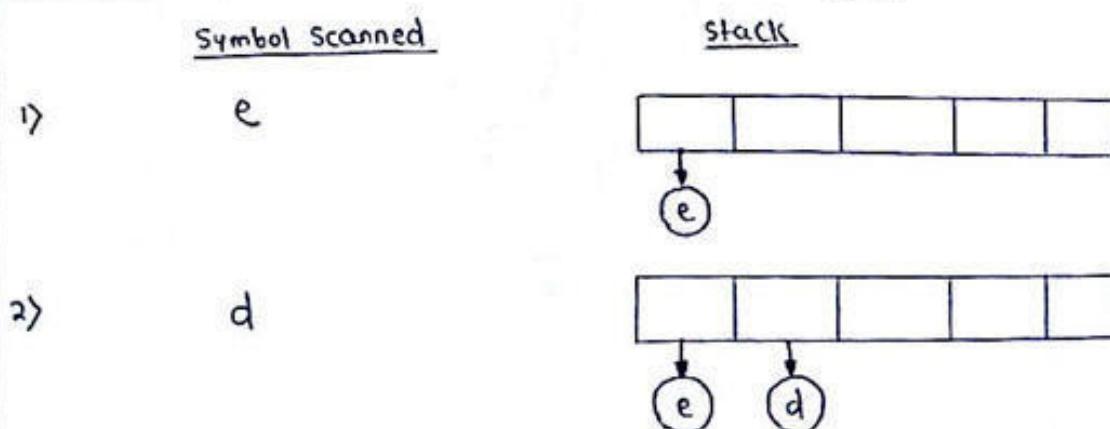




3) Construction of an expression tree from a prefix expression :

→ Construction of an expression tree from a prefix expression algorithm is same as the algorithm for evaluation of prefix expression.

Example : prefix expression = * + ab * c + de

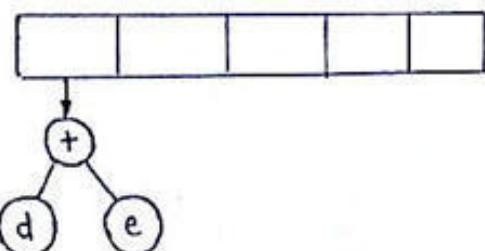


Symbol Scanned

Stack

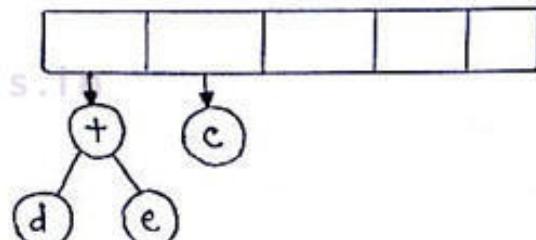
3>

+



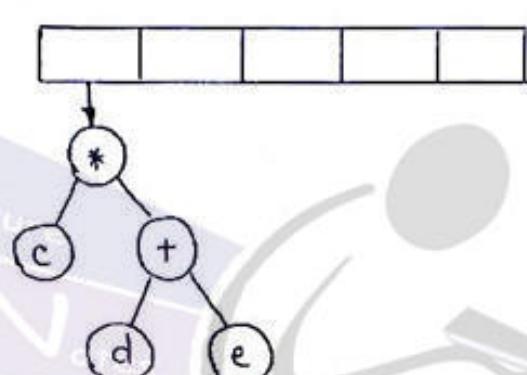
4>

c



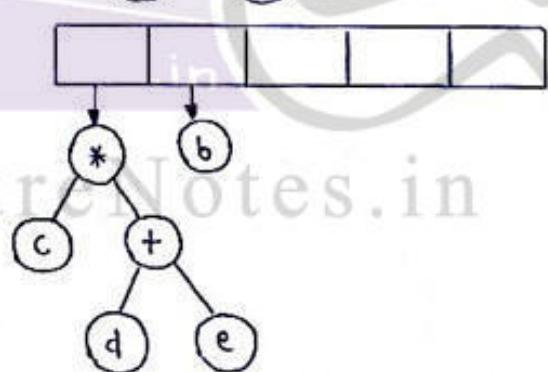
5>

*



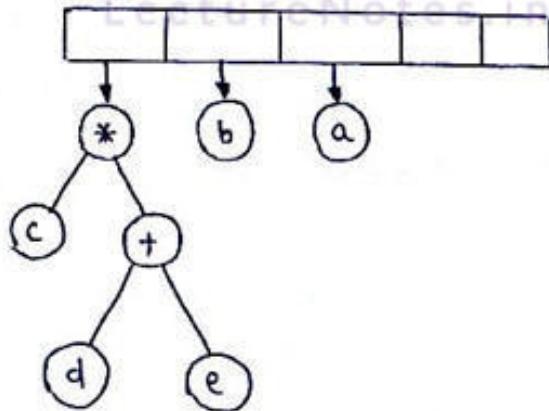
6>

b



7>

a

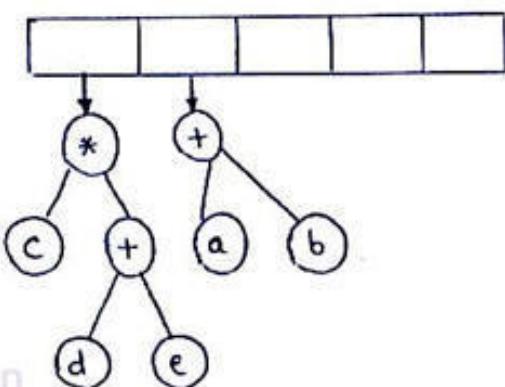


symbol scanned

8)

+

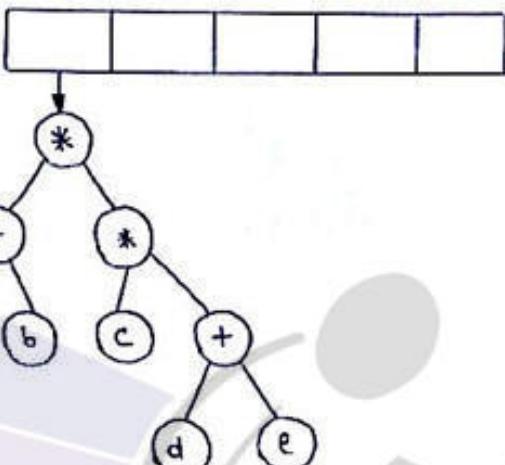
stack



LectureNotes.in

9)

*



NOTE :

- In expression tree if it will be traversed in preorder sequence, then its corresponding prefix expression will be obtained.
- If it will be traversed in postorder sequence, then its corresponding postfix expression will be obtained.

General Tree to Binary Tree :

Step1 : Consider the root node and findout its siblings.

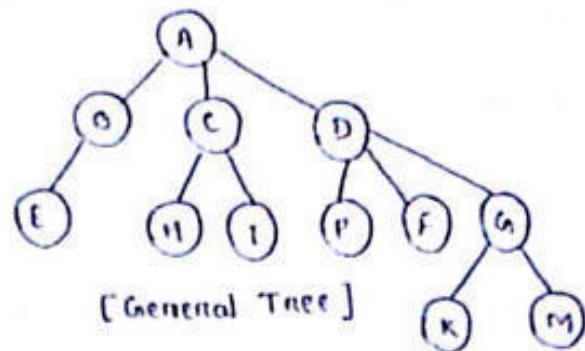
Step2 : Connect the siblings through edge.

Step3 : Cut all the parental edges which connects to its child except the leftmost child.

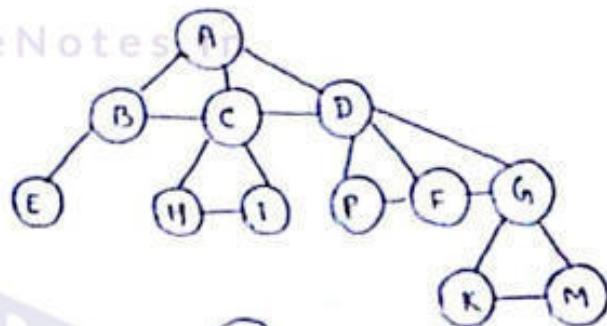
Step4 : Rotate all the nodes at an angle of 45° .

Example:

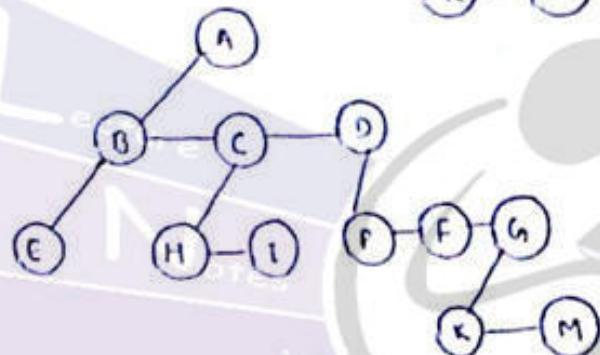
Step1 :



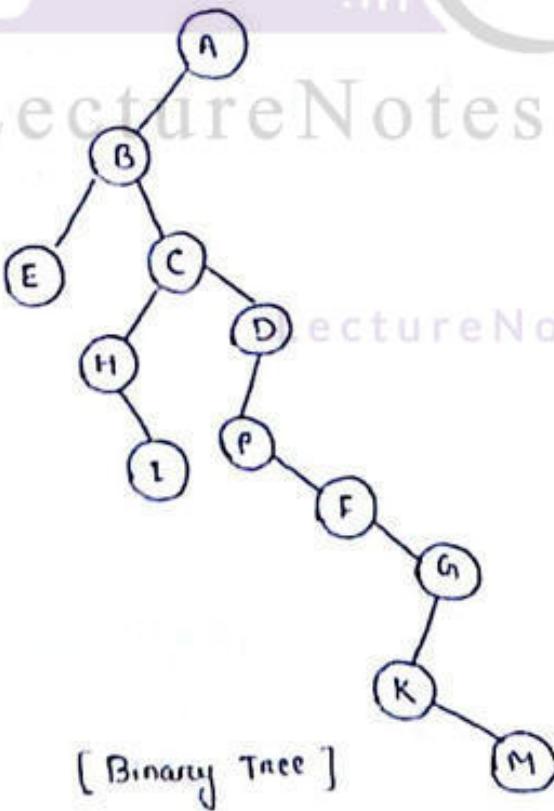
Step2 :



Step3 :



Step4 :





Data Structure Using C

Topic:
Binary Search Tree

Contributed By:
Mamata Garanayak

NOTE :

If the number of nodes in the tree is n then the number of binary tree constructed from the given number of node n is :

$$2^n - n$$

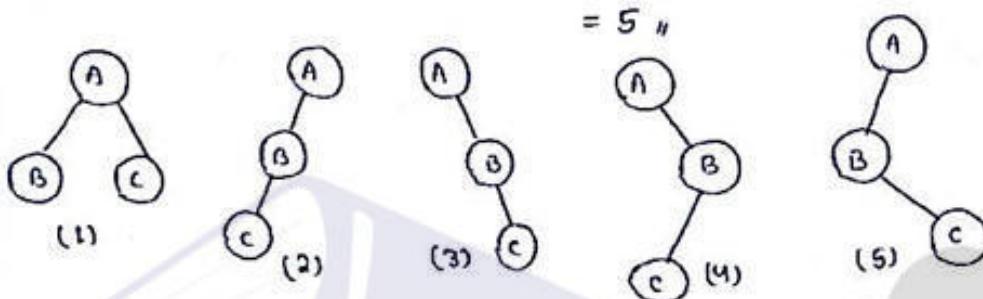
Example : no. of nodes = $3 = n$

$$\text{no. of possible binary tree} = 2^n - n$$

$$= 2^3 - 3$$

$$= 8 - 3$$

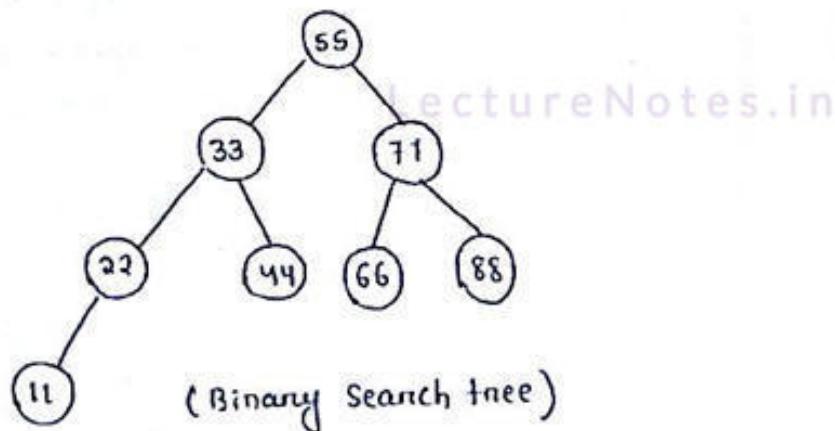
$$= 5 \text{ "}$$

Binary Search Tree : (BST)

Definition : A binary tree T is termed as binary search tree (BST) or binary sorted tree if each node n of T satisfies the following property.

- 1) The value at n is greater than the values of all nodes in its left subtree and
- 2) The value at n is less than the values of all nodes in its right subtree.

Example :



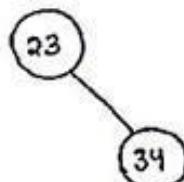
Creation of BST from the given nodes:

Nodes are : 23 34 35 100 49 53 36 L

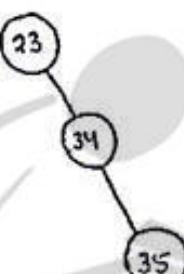
Step1: if root = NULL then
root = 23

23

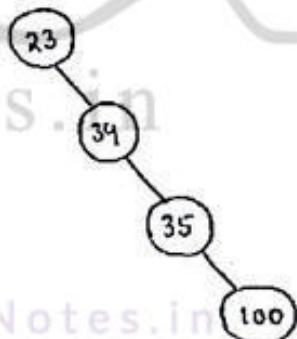
Step2: if ($34 > 23$) and
if $\text{root} \rightarrow \text{right} = \text{NULL}$ then
 $\text{root} \rightarrow \text{right} = 34$
else if $\text{root} \rightarrow \text{left} = \text{NULL}$ then
 $\text{root} \rightarrow \text{left} = 34$.



Step3: $35 > 23$
if $\text{root} \rightarrow \text{right} = '10'$
No.
if $35 > 34$ and
if $34 \rightarrow \text{right} = \text{NULL}$
 $34 \rightarrow \text{right} = 35$



Step4: $100 > 23$
 $\text{root} \rightarrow \text{right} = '10'.$ (No)



$100 > 35$
 $35 \rightarrow \text{right} = \text{NULL}$
 $\therefore 35 \rightarrow \text{right} = 100$

Step5: $49 > 100$ 23
 $\text{root} \rightarrow \text{right} = \text{NULL}$ (No)

$49 > 34$
 $34 \rightarrow \text{right} = \text{NULL}$ (No)

$49 > 35$
 $35 \rightarrow \text{right} = \text{NULL}$ (No)

$49 > 100$ (No)
 $100 \rightarrow \text{left} = 49$

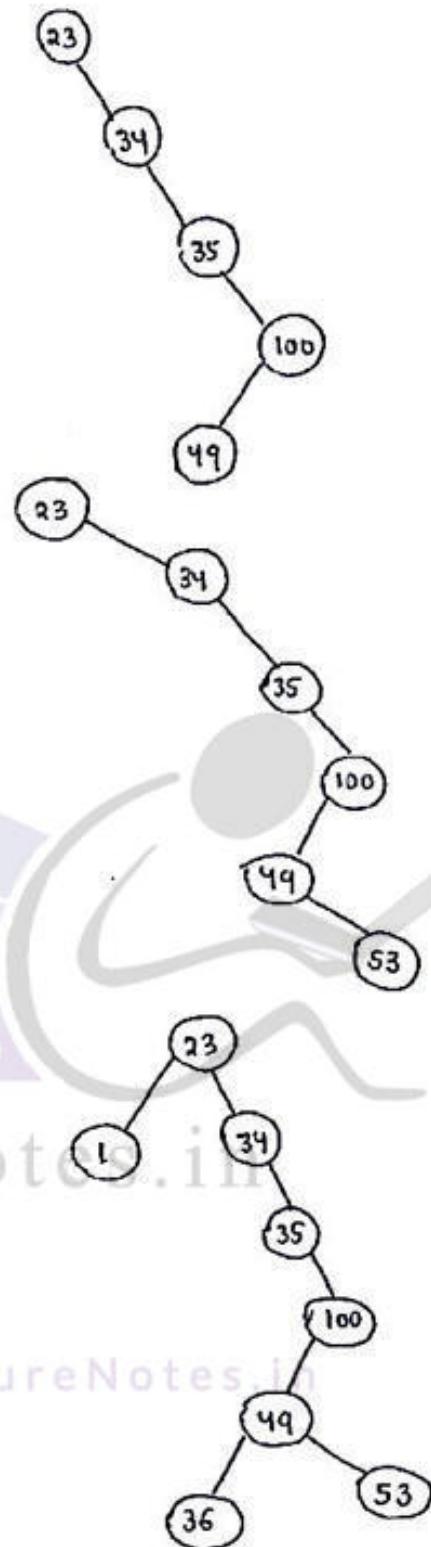
Step 6: $53 > 23$
 $23 \rightarrow \text{right} \neq \text{NULL}$

$53 > 34$
 $34 \rightarrow \text{right} \neq \text{NULL}$
 $53 > 35, 35 \rightarrow \text{right} \neq \text{NULL}$

$53 > 100$ (No)
 $100 \rightarrow \text{left} \neq \text{NULL}$
 $53 > 49, 49 \rightarrow \text{right} = 53$

Step 7: $36 > 23$
 $23 \rightarrow \text{right} \neq \text{null}$
 $36 > 34, 34 \rightarrow \text{right} \neq \text{null}$
 $36 > 35, 35 \rightarrow \text{right} \neq \text{null}$
 $36 > 100$ (No)
 $100 \rightarrow \text{left} \neq \text{null}$
 $36 > 49$ (No), $49 \rightarrow \text{left} = 36$

Step 8: $1 > 23$ (No)
 $23 \rightarrow \text{left} = \text{NULL}, \text{So}$
 $23 \rightarrow \text{left} = 1$



NOTE: If we allow the repetition of data elements in BST, then it should satisfy the following condition.

- i) Each node $n > \text{left}(n)$
- ii) Each node $n \leq \text{Right}(n)$.

Construct the Binary Search Tree from the following sequence of integers.

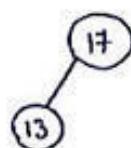
17, 13, 14, 18, 30, 6, 5, 35, 12, 8

Soln: Integers are : 17, 13, 14, 18, 30, 6, 5, 35, 12, 8

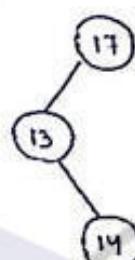
1) root = 17



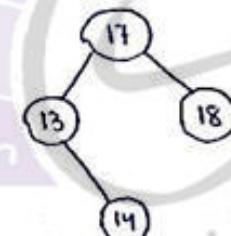
2) $13 > 17$ (No)
 $17 \rightarrow \text{left} = 13$



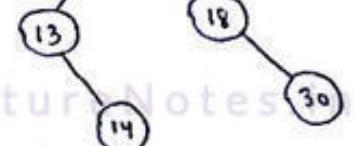
3) $14 > 17$ (No)
 $17 \rightarrow \text{left} \neq \text{null}$
 $17 \rightarrow \text{left} = 13$
so $14 > 13$
 $13 \rightarrow \text{right} = 14$



4) $18 > 17$ (Yes)
 $17 \rightarrow \text{right} = \text{NULL}$ so
 $17 \rightarrow \text{right} = 18$

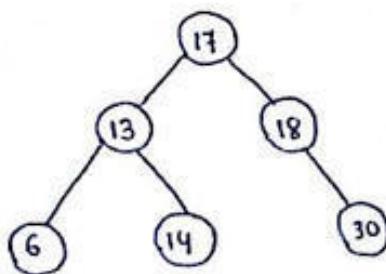


5) $30 > 17$ (Yes)
 $17 \rightarrow \text{right} \neq \text{null}$ (18)
 $30 > 18$ (Yes)
 $18 \rightarrow \text{right} = 30$

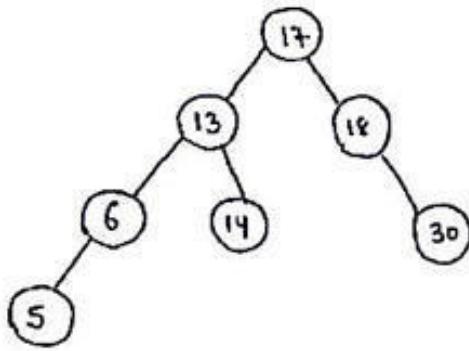


6) $6 > 17$ (No).
 $17 \rightarrow \text{left} \neq \text{null}$ (13)

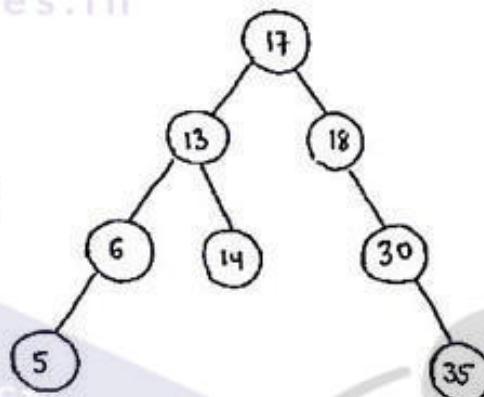
$6 > 13$ (No)
 $13 \rightarrow \text{left} \neq \text{null} = 6$



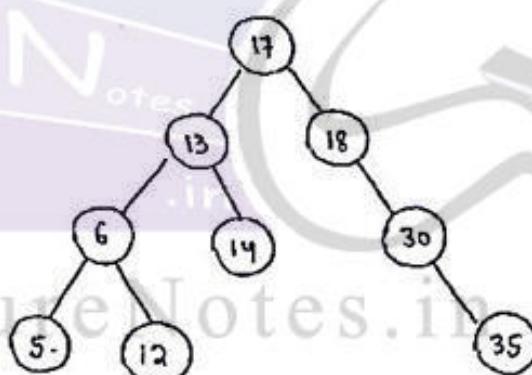
- 7) $5 > 17$ (No)
 $17 \rightarrow \text{left} \neq \text{null}$ (13)
 $5 > 13$ (No)
 $13 \rightarrow \text{left} \neq \text{null}$ (6)
 $5 > 6$ (No)
 $6 \rightarrow \text{left} = 5$



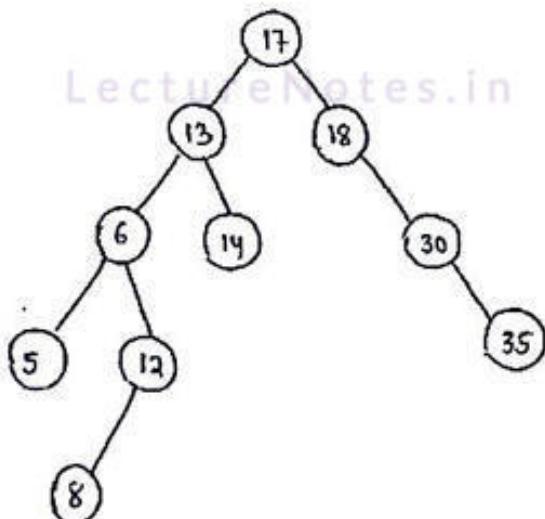
- 8) $35 > 17$ (Yes)
 $17 \rightarrow \text{right} \neq \text{null}$ (18)
 $35 > 18$ (No Yes)
 $18 \rightarrow \text{right} \neq \text{null}$ (30)
 $35 > 30$ (Yes)
 $30 \rightarrow \text{right} = 35$



- 9) $12 > 17$ (No)
 $17 \rightarrow \text{left} \neq \text{null}$ (13)
 $12 > 13$ (No)
 $13 \rightarrow \text{left} \neq \text{null}$ (6)
 $12 > 6$ (Yes)
 $6 \rightarrow \text{right} = 12$



- 10) $8 > 17$ (No)
 $17 \rightarrow \text{left} \neq \text{null}$ (13)
 $8 > 13$ (No)
 $13 \rightarrow \text{left} \neq \text{null}$ (6)
 $8 > 6$ (Yes)
 $6 \rightarrow \text{right} \neq \text{null}$ (12)
 $8 > 12$ (No)
 $12 \rightarrow \text{left} = 8$



Operations In BST :

Operations in Binary search tree includes

- i) insertion
- ii) Deletion .
- iii) searching.

Insertion :

Algorithm:

PROCEDURE Insertion - in-BST (root, nw, info, R-child, L-child, ptr)

// root : pointer variable which holds the address of 1st node in BST.

// nw : pointer variable which holds the address of newly created node .

// info : variable for holding the information part of each node .

// R-child : pointer variable which holds the address of right child of a given node .

// L-child : pointer variable which holds the address of left child of a given node .

// ptr : temporary pointer variable .

steps: if (root = null) then

i) root = nw .

ii) goto step 5

step2: else

ptr = root .

step3: if (ptr → info > nw → info) then

i) move to left child of ptr and check if (ptr → left = null)

then set ptr → left = nw .

ii) else move to begining of step 3 .

step4: if (ptr → info < nw → info) then

i) move to right child of ptr and check

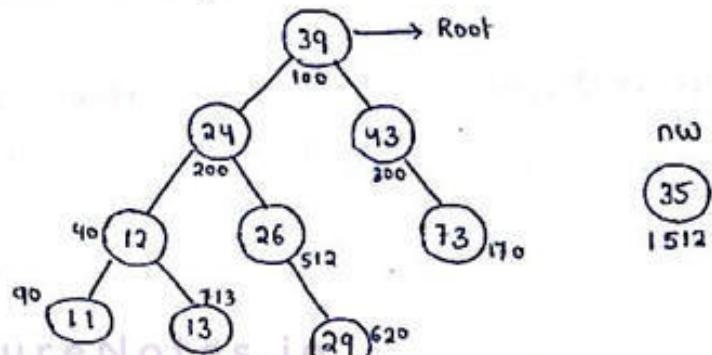
if (ptr → right = null) then set ptr → right = nw .

ii) else move to begining of step 3 .

step5: return root .

Step 6 : Exit

Example: Given a BST



In the above BST, we want to insert 35

1) Root = null (No)

so ptn = 100

2) if ($39 > 35$)

ptn = 200.

if ($200 = \text{null}$) (No).

if ($24 > 35$) (No)

if ($24 < 35$) then

ptn = 512

if ($512 = \text{null}$) (no)

if ($26 < 35$) (Yes)

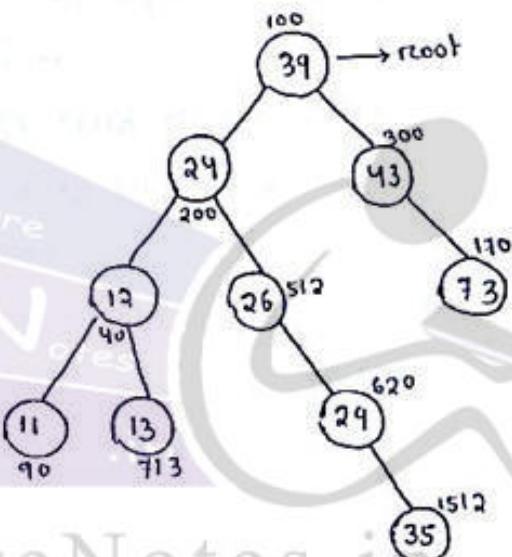
ptn = 620

if ($620 = \text{null}$) (no)

if ($29 < 35$) then

ptn = null

if ($\text{ptn} \rightarrow \text{right} = 35$



Deletion :

Algorithm :

PROCEDURE Delete-node-in-BST (N, L(N), R(N), P(N))

// N : gt is the node to be deleted.

// L(N) : left child of node N.

// R(N) : Right child of node N.

// P(N) : parent node of N.

Step1: If the node 'N' is not having any child node then

a) If 'N' is the right child of P(N) then

i) Assign null to right part of P(N).

ii) delete node N.

b) if 'N' is the Left child of P(N) then

i) Assign null to left part of P(N)

ii) delete node N.

Step2: If there exist only one child node of N then

a) If 'N' is the right child R(N) then

P(N) → right = child node of N.

b) else

P(N) → left = child node of N.

c) delete (N).

Step3: If node 'N' having 2 child node then

a) goto the right subtree of N and findout the smallest value in right subtree.

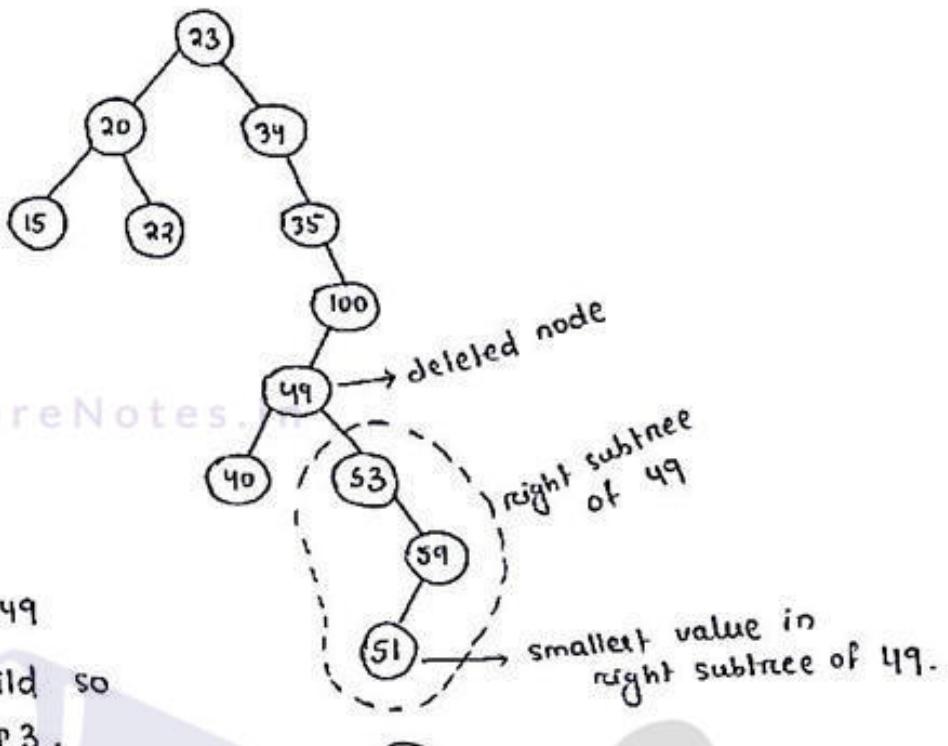
b) By the help of step1 and step2 the node having smallest value will be deleted.

c) replace N by the smallest node

d) delete (N)

Step4: Exit

Example :



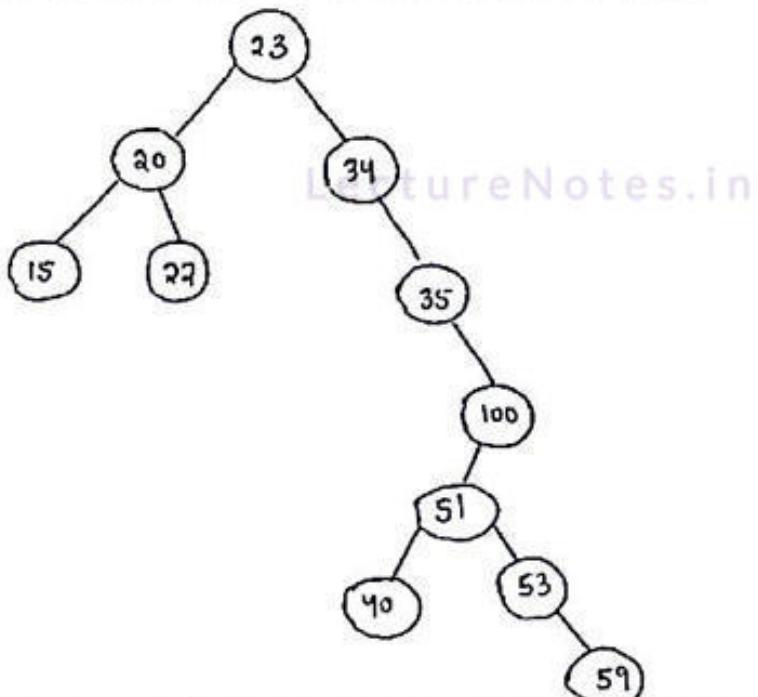
deleted node = 49

49 have 2 child so
we go for step 3.

The right subtree of 49 is →
The smallest value in the right
subtree of 49 = 51

So 51 is deleted from that
position and 49 is replaced by 51

Now the tree becomes :



Searching :

Algorithm :

PROCEDURE Search-in-BST (t, item, left, right, info)
// t : Given Binary search tree's root node
// item : searched item
// left : left child of a given node .
// right : right child of a given node .
// info : information part of each node .

Step 1 : if (t == null) then

- i) print the tree is empty
- ii) exit .

Step 2 : else

while (t->info != item && t != null)

i) if (item < t->info)

t = t->left .

ii) else

t = t->right

[End of while]

Step 3 : if (t->info == item)

i) print item is found having address t .

else

print item is not found .

Step 4 : exit .

C - procedure :

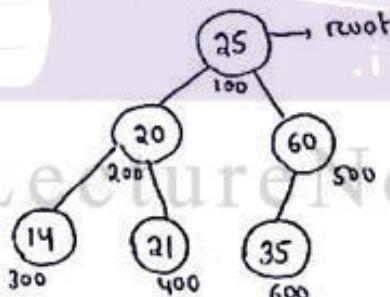
```
void search-BST ( struct node *t, int item )
{
    // right, left, info are global variable . temp is also a global variable
    printf (" Enter the item to be searched " );
    scanf ("%d", &item );
    exit
}
```

```

if ( t == NULL)
{
    printf("Tree is empty");
    exit();
}
else {
    while ( t->info != item && t != NULL)
    {
        // temp = t;
        if (item < t->info)
            t = t->left;
        else
            t = t->right;
    }
    if ( t->info == item)
        printf("Item is found in node having the address %u", t);
    else
        printf("Item is not found");
}

```

Example :



Search item = 14

$t = 100$.

while ($25 \neq \text{item} \&\& 100 \neq \text{NULL}$)

$\text{temp} = 100$.

if ($14 < 25$)

$t = 200$

while ($20 \neq \text{item} \&\& 200 \neq \text{NULL}$)

$\text{temp} = 200$

if ($14 < 20$)

$t = 300$

while ($14 \neq \text{item} \&\& 300 \neq \text{NULL}$)

(false)

∴ item is found in node having

address 300.

C - procedure for Finding the Maximum Value in BST :

```
Void Find-Max ( struct node * t )
{
    // right , left , and info are global variable
    if ( t == NULL )
    {
        printf (" Tree is empty ");
        exit();
    }
    else
    {
        while ( t ->right != '10' )
            t = t ->right ;
        printf (" The maxm value is %d ", t ->info );
    }
}
```

C - procedure for Finding the Minimum Value in BST :

```
Void Find-Min ( struct node * t )
{
    // right , left and info are global variable .
    if ( t == NULL )
    {
        printf (" Tree is empty ");
        exit();
    }
    else
    {
        while ( t ->left != NULL )
            t = t ->left ;
        printf (" The minm value is %d ", t ->info );
    }
}
```

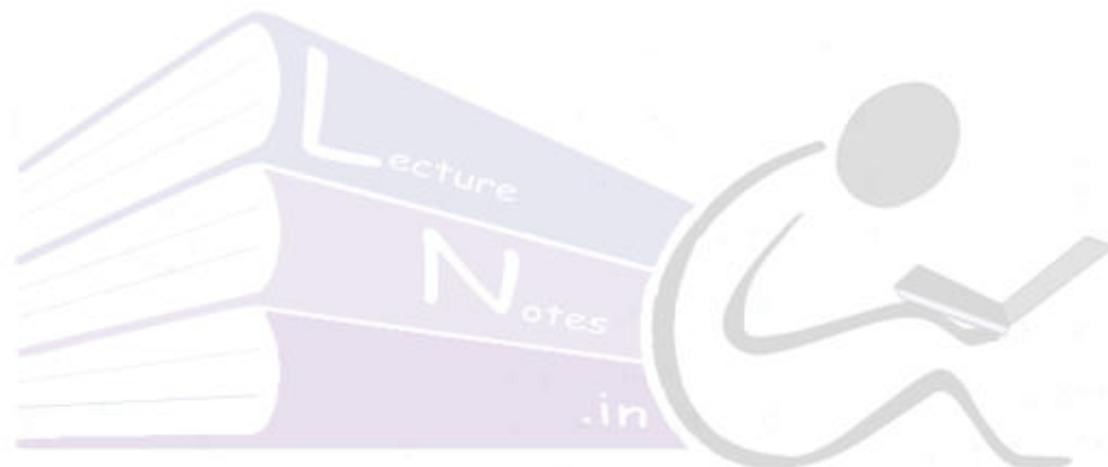
Advantages of BST :

- Average time complexity for searching operation is $O(\log_2 n)$.
- Insertion and deletion operation is quite easy.

Disadvantage of BST :

- If a binary search tree is right skewed or left skewed then its time complexity for searching operation in worst case is $O(n)$.
(\because It is same as sequential search).

LectureNotes.in



LectureNotes.in

LectureNotes.in



Data Structure Using C

Topic:
AVI Tree

Contributed By:
Mamata Garanayak

AVL Tree : [Height Balanced Tree] [Adesom - Velskii - Landis]

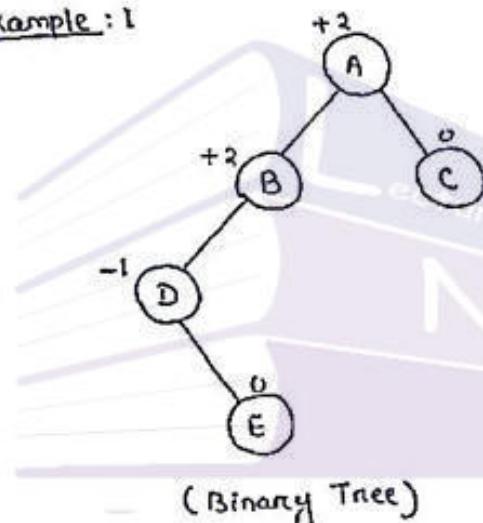
Definition: A binary tree can be called as an AVL tree or a height balanced tree if its left subtree and right subtree is balanced and balance factor of each node is -1, 0 or 1.

Balance Factor :

The balance factor for any node N is denoted as $BF(N)$.

$$BF(N) = h_L - h_R$$

where h_L = height of left subtree
 h_R = height of Right subtree.

Example: I

$$BF(A) = 3 - 1 = 2$$

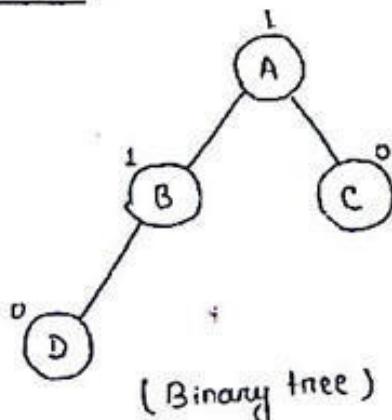
$$BF(B) = 2 - 0 = 2$$

$$BF(C) = 0 - 0 = 0$$

$$BF(D) = 0 - 1 = -1$$

$$BF(E) = 0 - 0 = 0$$

thus this tree is not an AVL tree,
because left subtree is not balanced
and also the root node is not balanced.

Example: II

$$BF(A) = 2 - 1 = 1$$

$$BF(B) = 1 - 0 = 1$$

$$BF(C) = 0 - 0 = 0$$

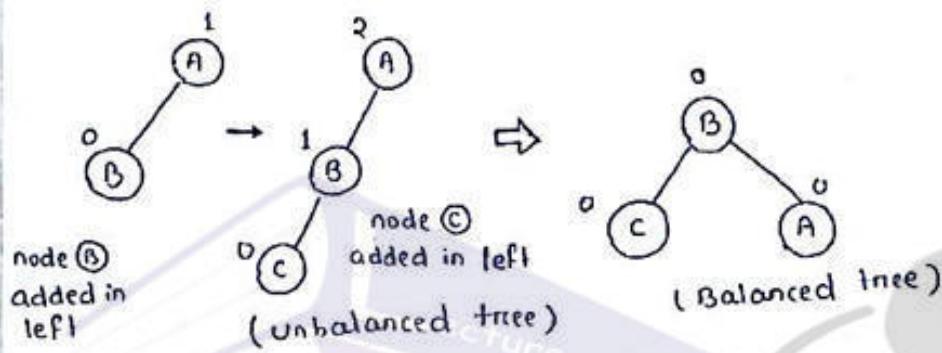
$$BF(D) = 0 - 0 = 0$$

thus the above binary tree is an AVL tree.

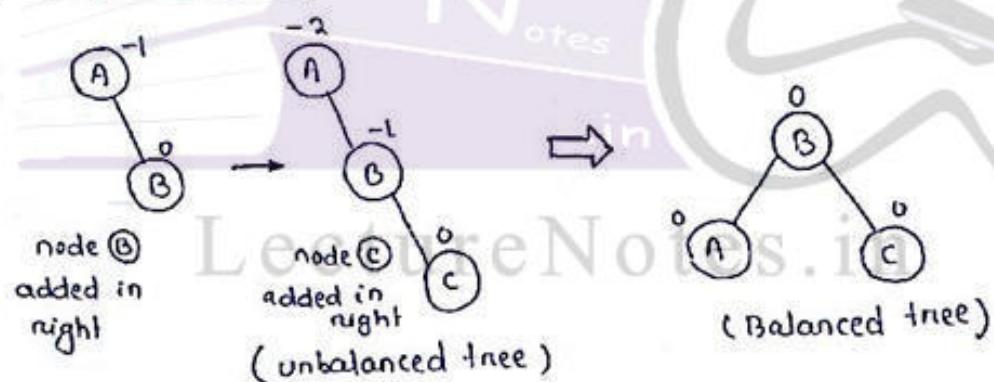
If a binary tree is not balanced then it can be balanced by the help of any one of the rotation methods. These rotations are :

- 1) LL rotation
- 2) RR rotation
- 3) LR rotation
- 4) RL rotation.

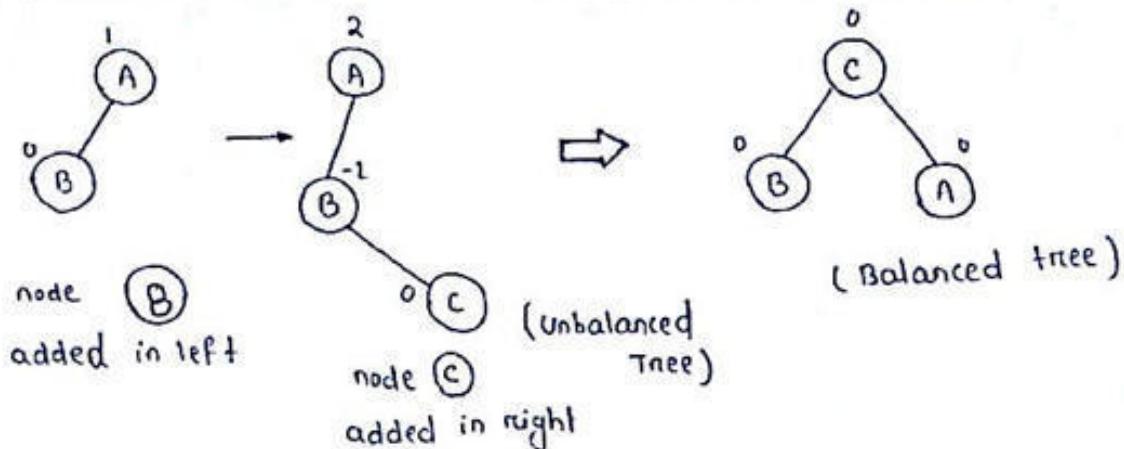
1) LL rotation :



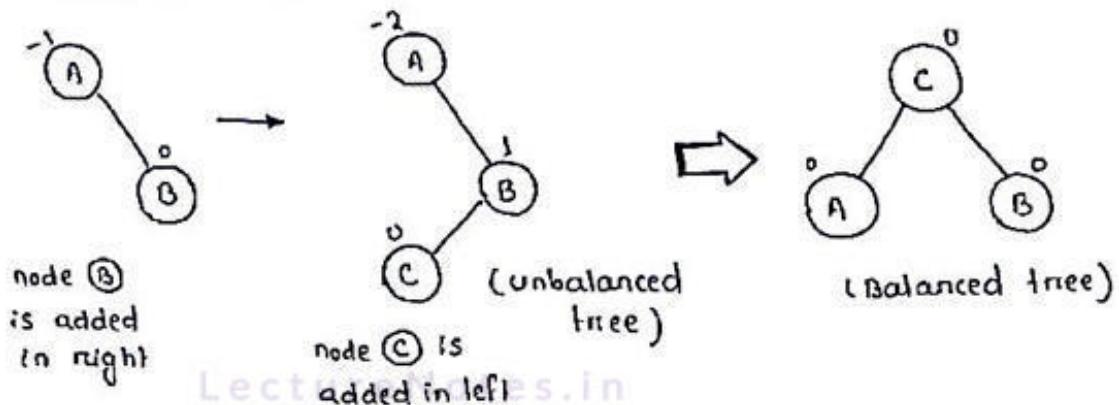
2) RR rotation :



3) LR rotation :



↳ RL rotation:



Example:

Create an AVL tree for the data items given below.

56, 69, 80, 18, 14, 36, 25, 38, 28, 50, 87, 98

Insertion of data items in an AVL tree is same as insertion in BST. Here only we have to check the balanced factor and balanced the tree after insertion of one item, then another item and so on.

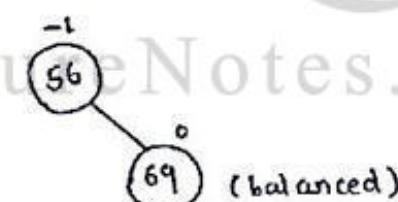
56

56

69

$69 > 56$ (yes)

$56[\text{right}] = 69$



80

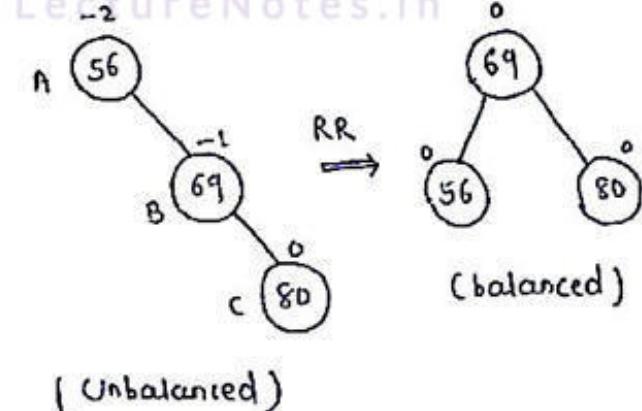
$80 > 56$ (yes)

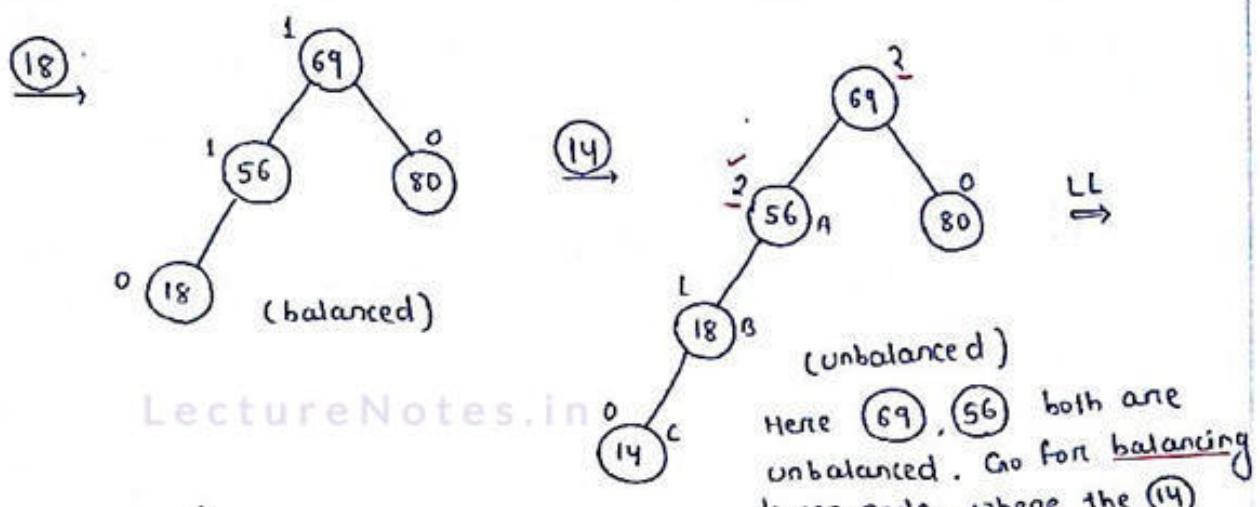
$56 \rightarrow \text{right} = 69$

$80 > 69$ (yes)

$69 \rightarrow \text{right} = 80$

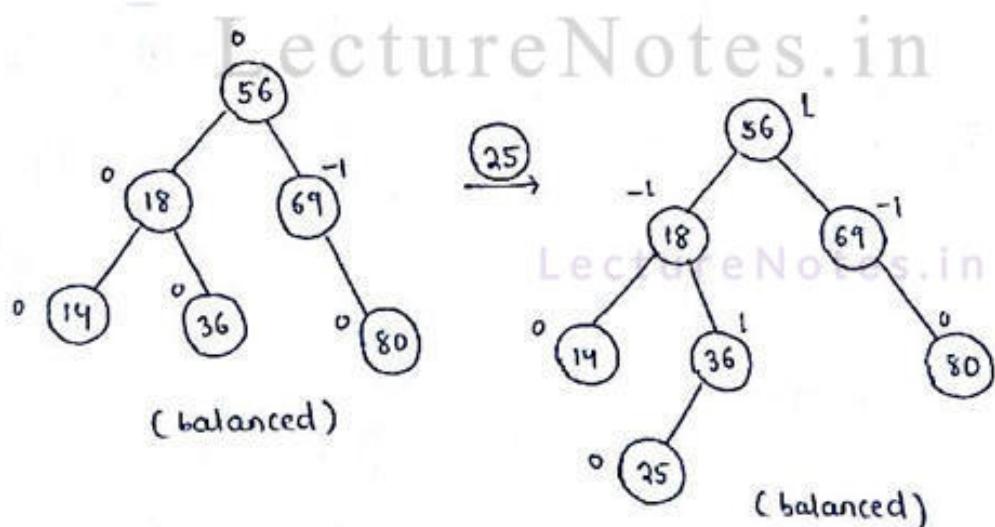
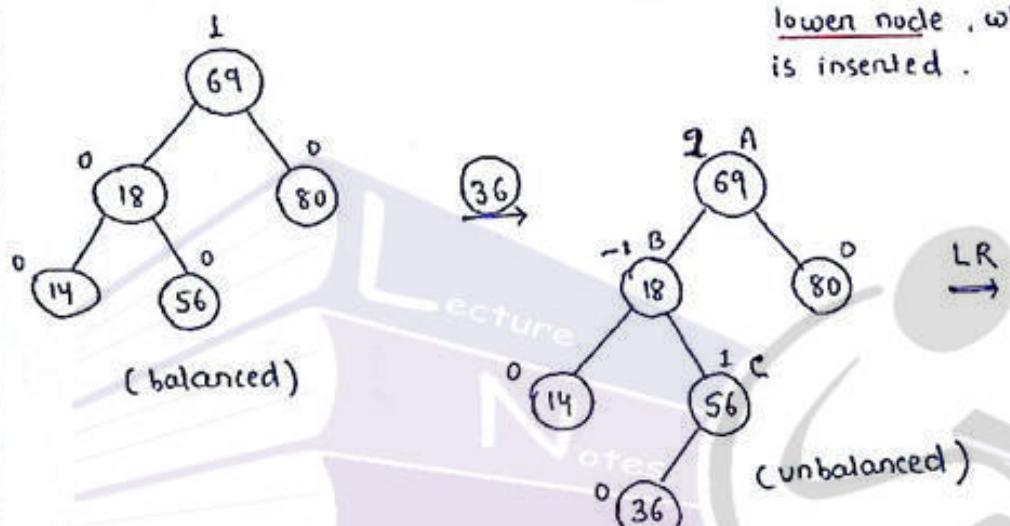
LectureNotes.in

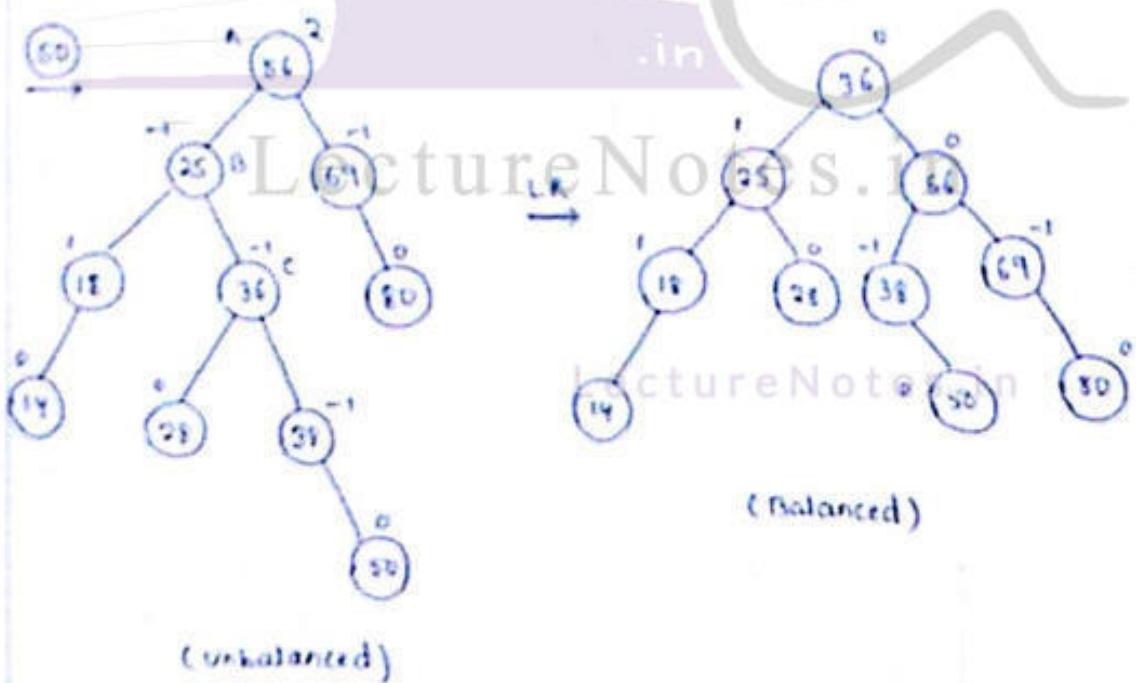
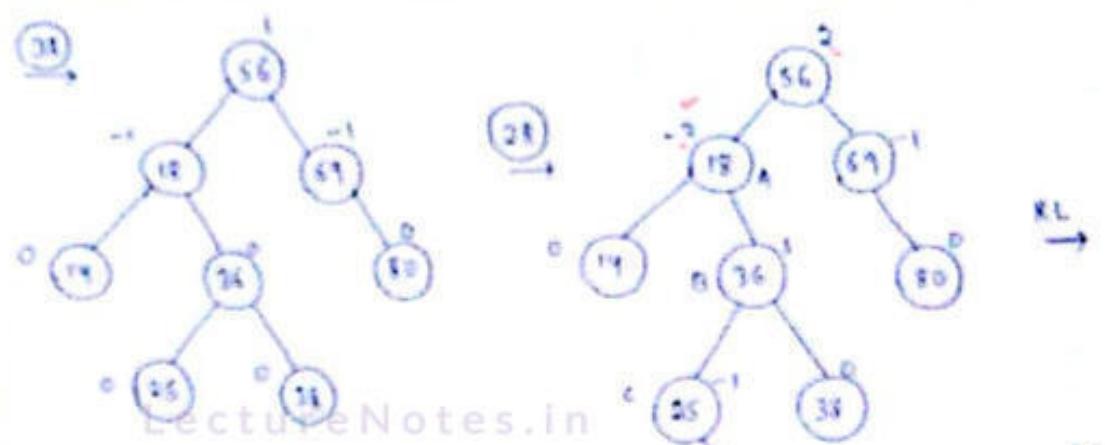


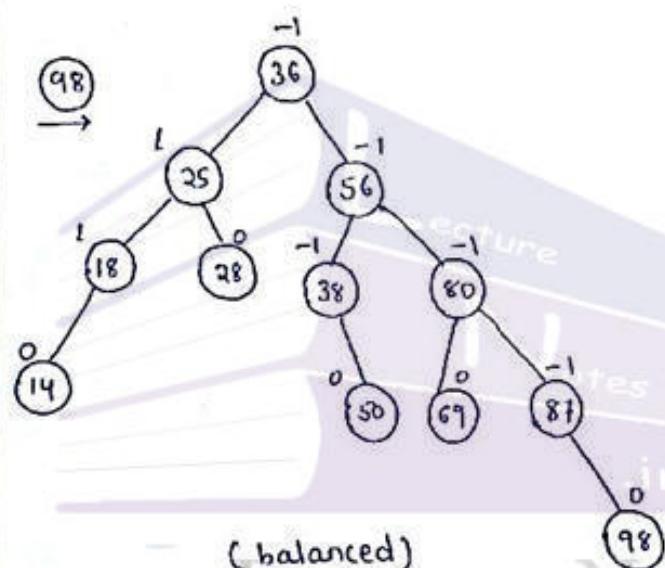
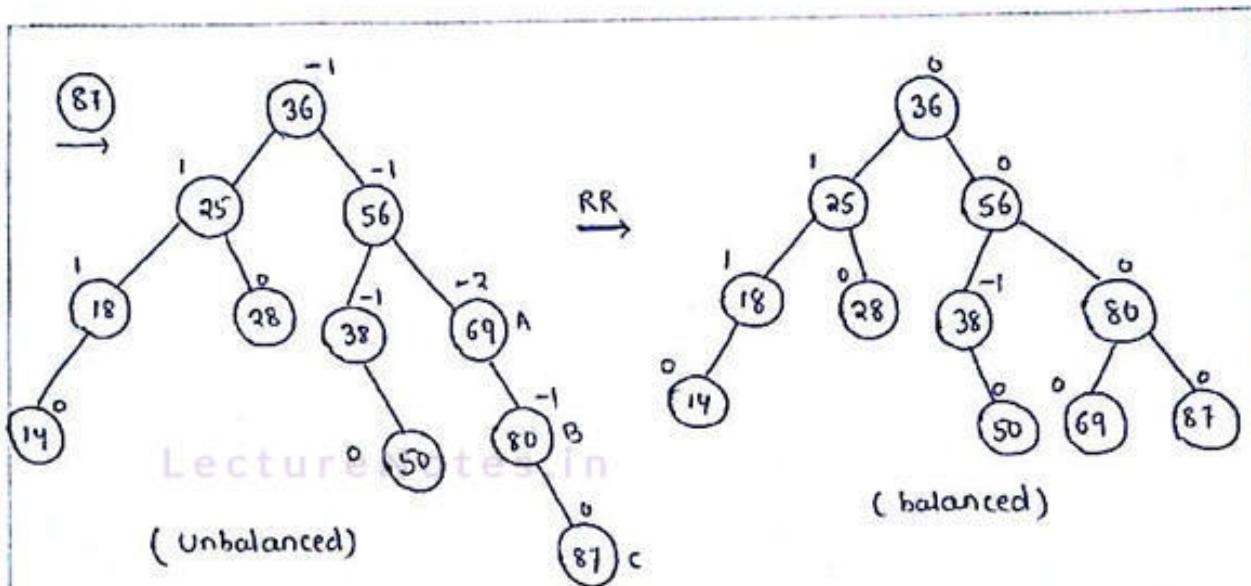


LectureNotes.in

Here 69, 56 both are unbalanced. Go for balancing lower node, where the 14 is inserted.



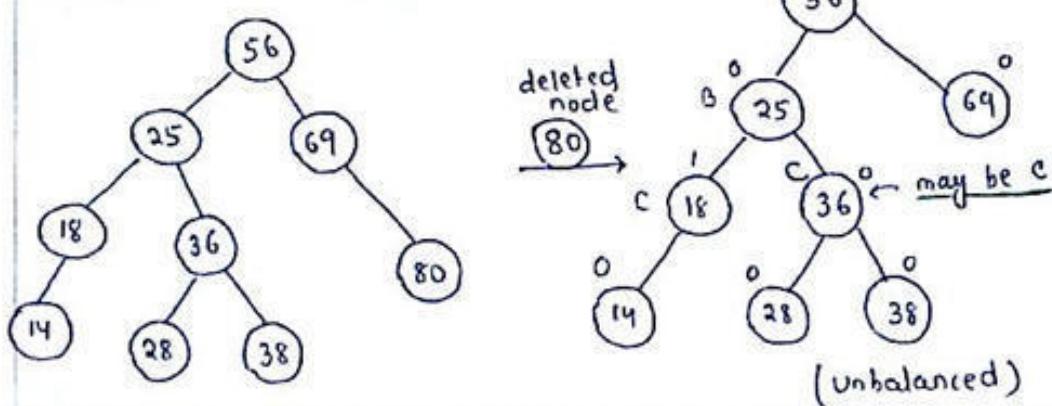


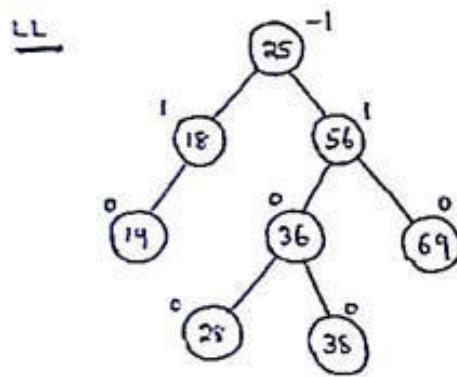


Deletion in AVL :

Deletion in AVL : Deletion in AVL tree is same as deletion in BST. But here after deleting one node we have to balance the tree.

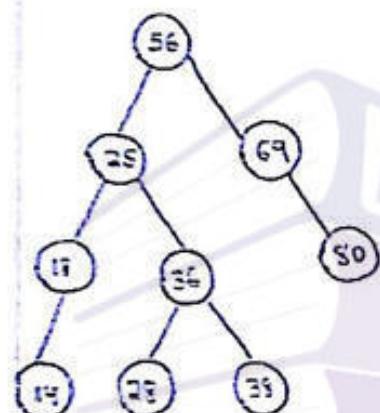
case1 : No child node



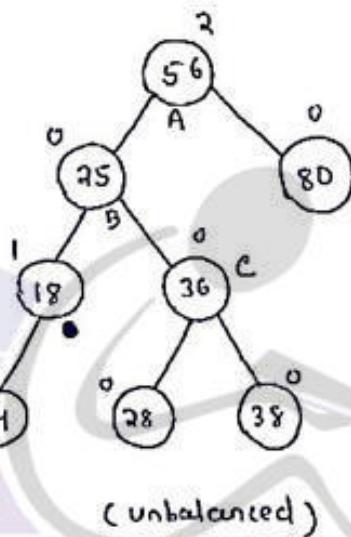


LectureNotes.in
(Balanced)

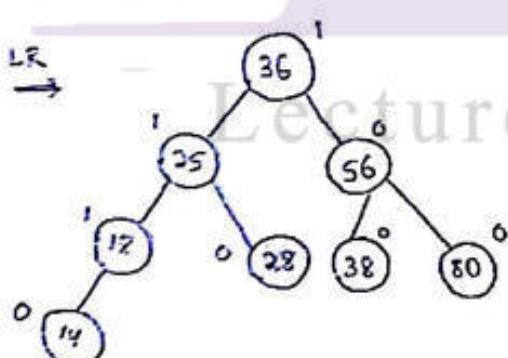
Case 2: One child (may be left or may be Right)



deleted node
69
69 has
one child
(right child)

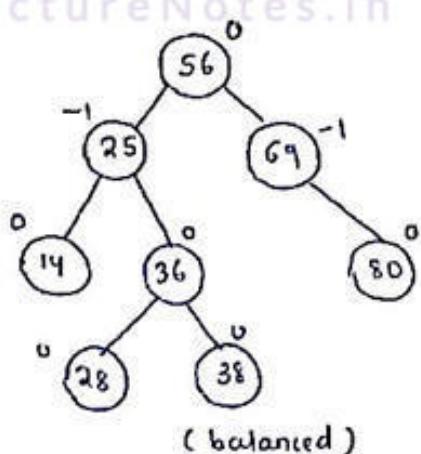
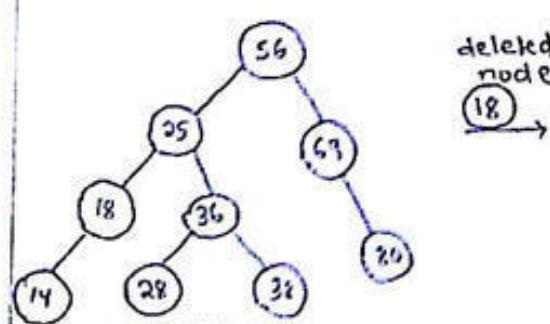


LectureNotes.in
(unbalanced)



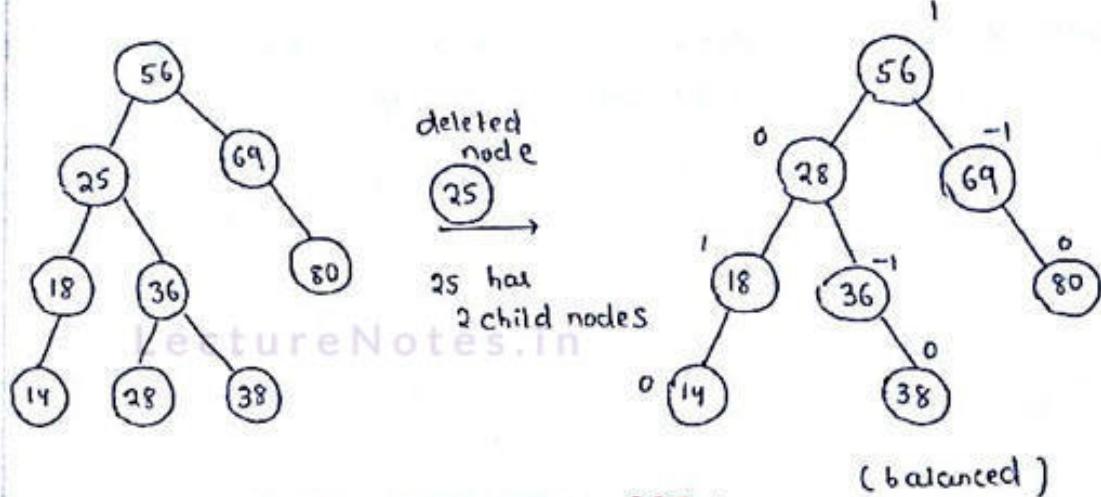
LectureNotes.in
(Balanced)

if the deleted node is
18 then it has one child
i.e. left child.



LectureNotes.in
(balanced)

3) Case 3: Two child node



Advantage of AVL tree over BST :

- If a BST is right skewed or left skewed then the worst case time complexity of a search is $O(n)$. If the BST is a balanced height, it will require a worst case time complexity of search equal to $O(\log n)$ which is less than $O(n)$.
- Time complexity for insertion operation in AVL tree is also $O(\text{height}) = O(\log n)$.
- The height of an AVL tree with n nodes can never exceed $1.44 \log n$.

NOTE

AVL = Adelson Velskii & Landis .



Data Structure Using C

Topic:

B-Tree

Contributed By:

Mamata Garanayak

Lesson Number : 35

B-Tree :

- A B-tree is a balanced m-way tree. A node of the tree may contain many records or keys and pointers to children.
- A B-tree is also known as the balanced sort tree. It is not a binary tree. It finds its use in external sorting.
- To reduce disk access, several conditions of the tree must be :
 - 1) The height of the tree must be kept to a minimum.
 - 2) There must be no empty subtrees above the leaves of the tree.
 - 3) The leaves of the tree must all be on the same level.
 - 4) All nodes except the leaves must have at least some minimum number of children.

Properties of B-tree of order m :

- a) It is a m-way search tree in which all leaves are on the same level.
- b) All internal nodes except the root will have maximum m children and minimum $\frac{m}{2}$ children.
- c) The root node has almost m children, but may have as few as 2 if it is not a leaf.
- d) The keys in a node are arranged in the fashion of a search tree.
- e) The tree is allowed to grow towards the root, not towards leaf (if it is allowed to grow towards leaf then all leaf may not be in the same level).

Operations in B-Tree :

There are two major operations are performed in B-tree.

- 1) Insertion into a B-tree
- 2) Deletion from a B-tree

Insertion :

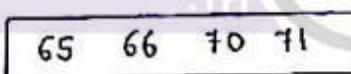
- A search is made to see if the new key is already in the tree, if the search fails the new key is added to the leaf node.
- If the node was not previously full then the insertion is over.
- If the new key is added to a full node then the node is splitted into two nodes on the same level but the median key value is not put in either of these two splitted nodes, rather it is send up the tree to be inserted in the parent node.
- The splitted node will have exactly $m/2$ items. If this insertion also causes an overflow in the parent node then i.e. also splitted and the splitting propagates upwards, in the extreme case, it may propagate upto the root and may cause an increase in the height of the tree.

Example :

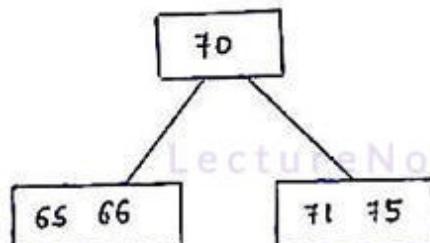
Let us take an example such as growth of the B-tree of order 5.

Keys are : 65, 71, 70, 66, 75, 68, 72, 71, 74, 69, 83, 73, 82, 88, 67, 76, 78, 84, 85, 80

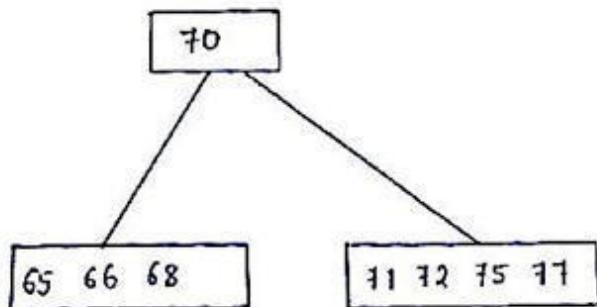
i) 65, 71, 70, 66



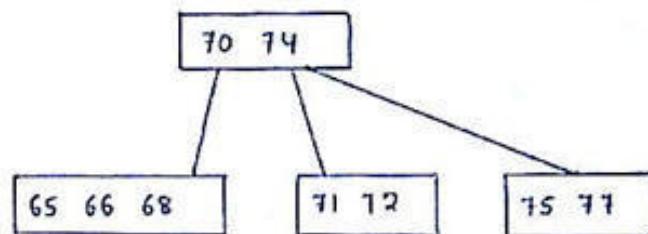
ii) 75



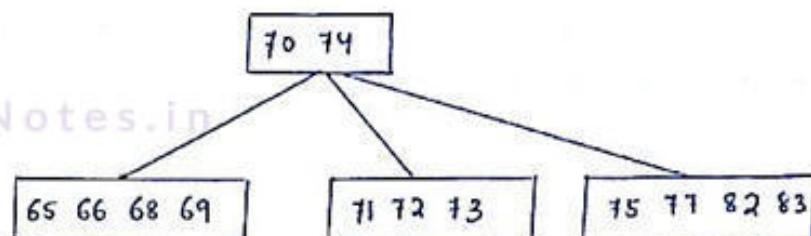
iii) 68, 72, 77



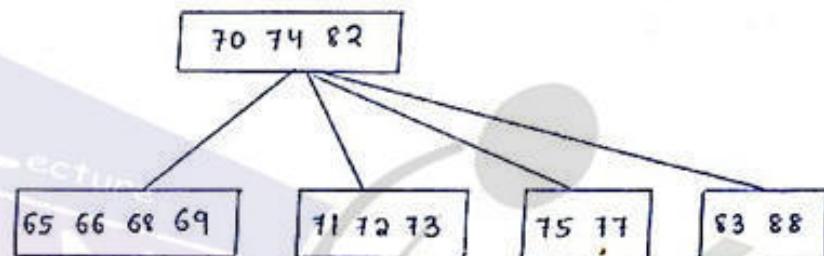
iv) 74



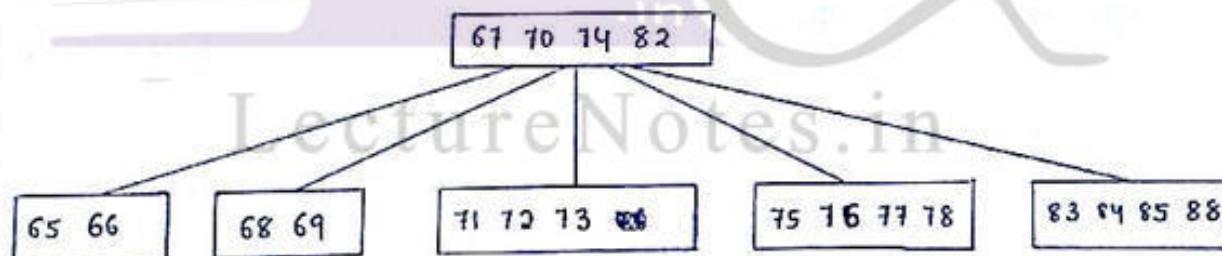
v) 69, 83, 73, 82



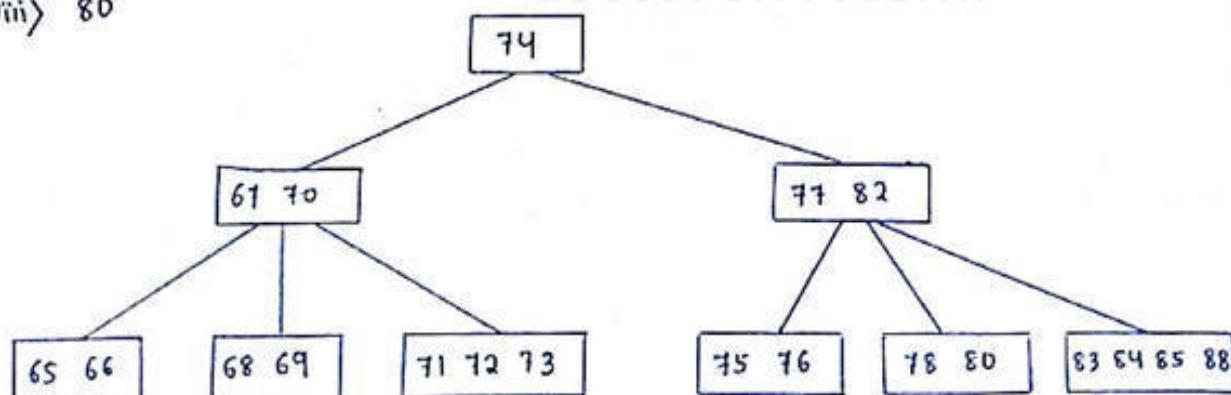
vi) 88



vii) 61, 70, 74, 84, 85



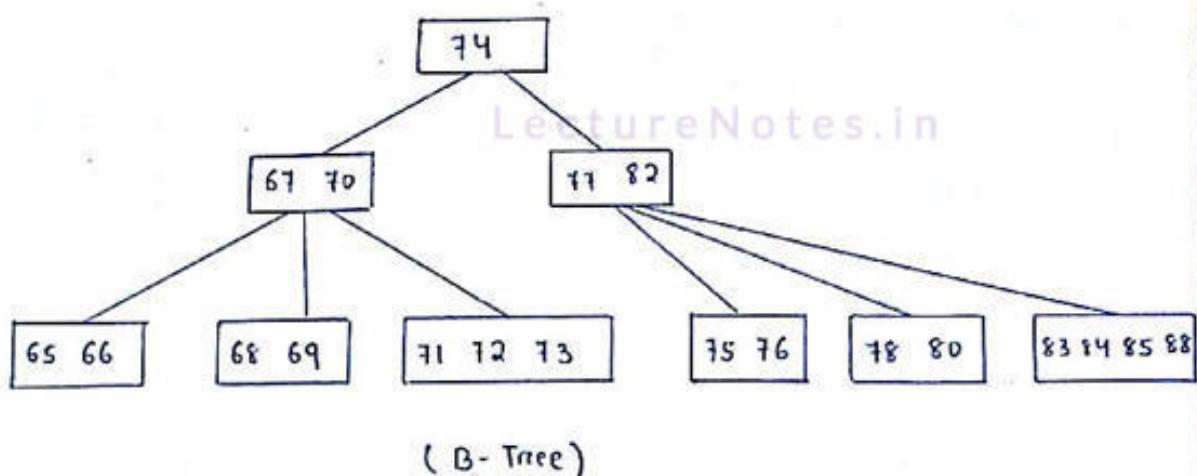
viii) 80



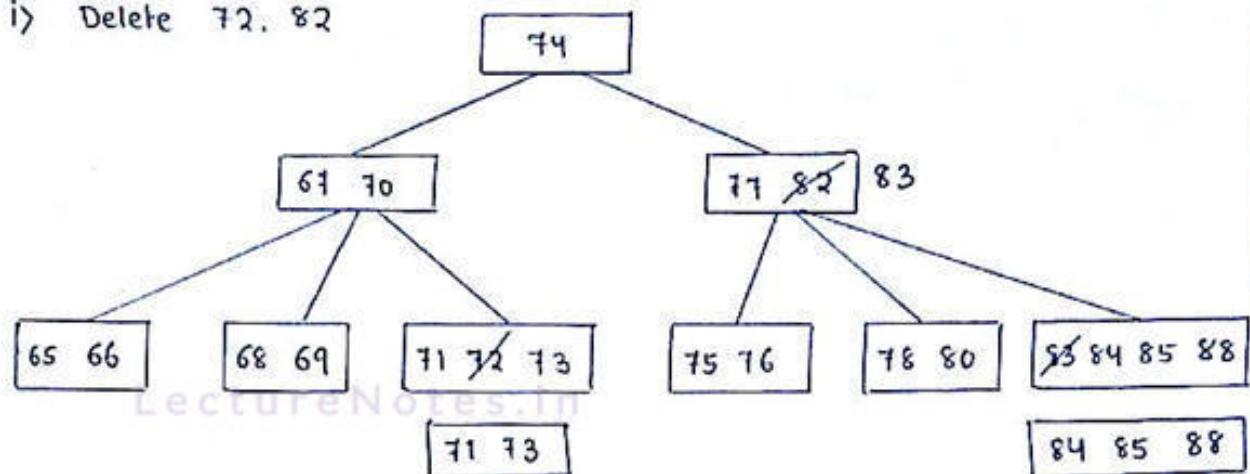
Deletion :

- Deletion from a leaf node
- Deletion from a nonleaf node
- If the node to be deleted is on leaf then it is deleted. If it is not in leaf node it must be replaced by its inorder successor or predecessor which happens to be on the leaf and is deleted easily.
- If the leaf contains more than the minimum then the deletion is made with no further action.
- If the leaf contains the minimum number then we have to look at the two leaves that are immediately adjacent and are children of the same node. If one of these has more than the minimum then one key of them can be moved into the parent node and a key from the parent can be moved down into the leaf where the deletion is occurring.
- If the adjacent leaves has only the minimum number then the two leaves and the median key from the parent can all be combined together into a new leaf node which will contain no more than the maximum key. If this step deletes the parent node with less than the minimum then the process will propagate upwards. In the extreme case it is propagated up to the root and the height of the tree decreases.

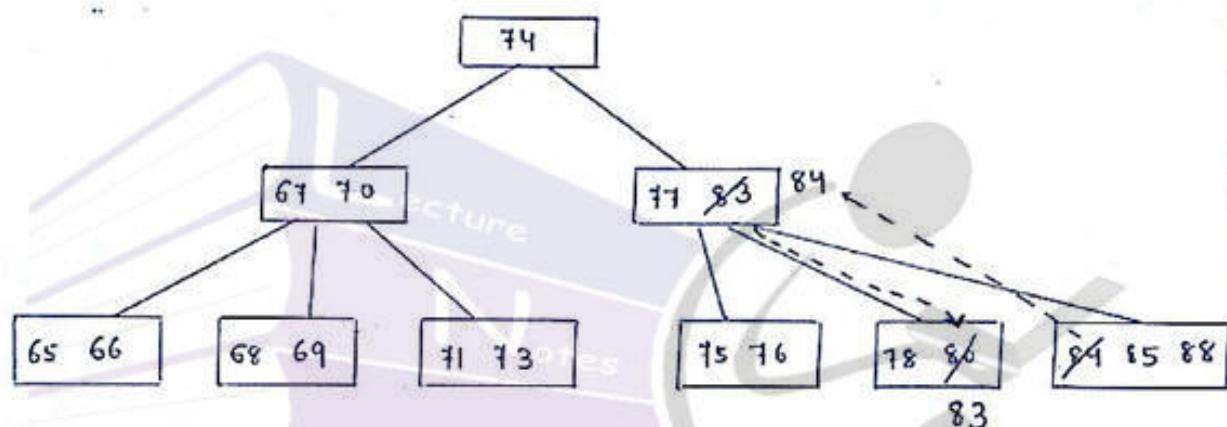
Example: Delete nodes 72, 82, 80, 68 from the following B-tree.



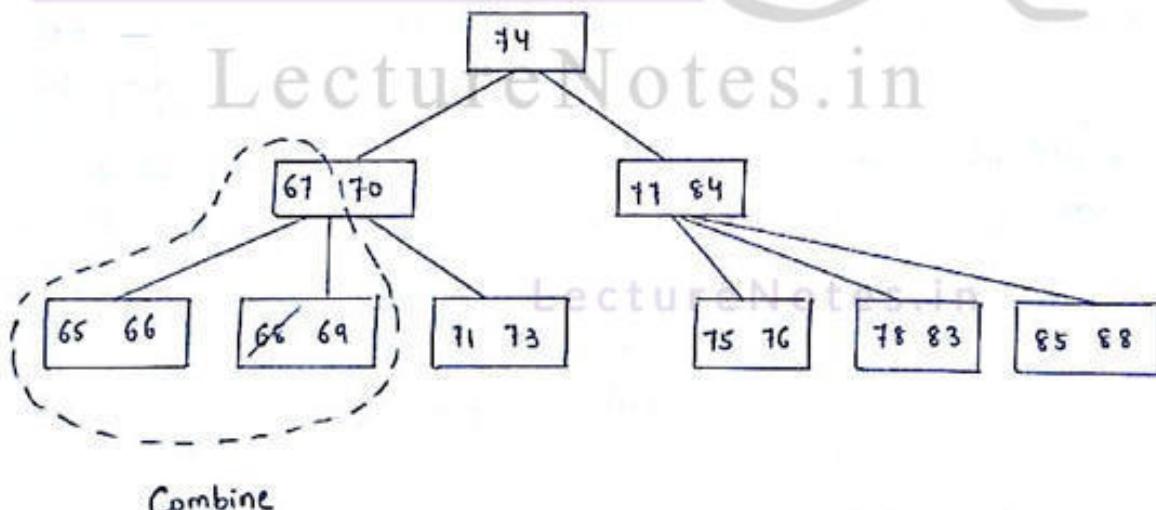
i) Delete 72, 82

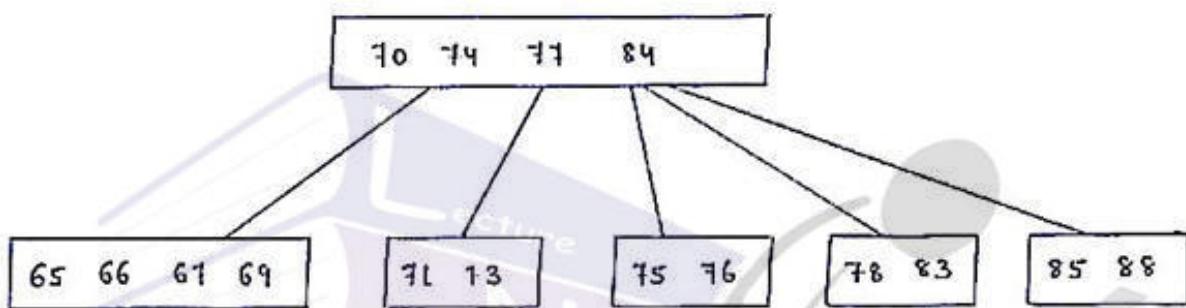
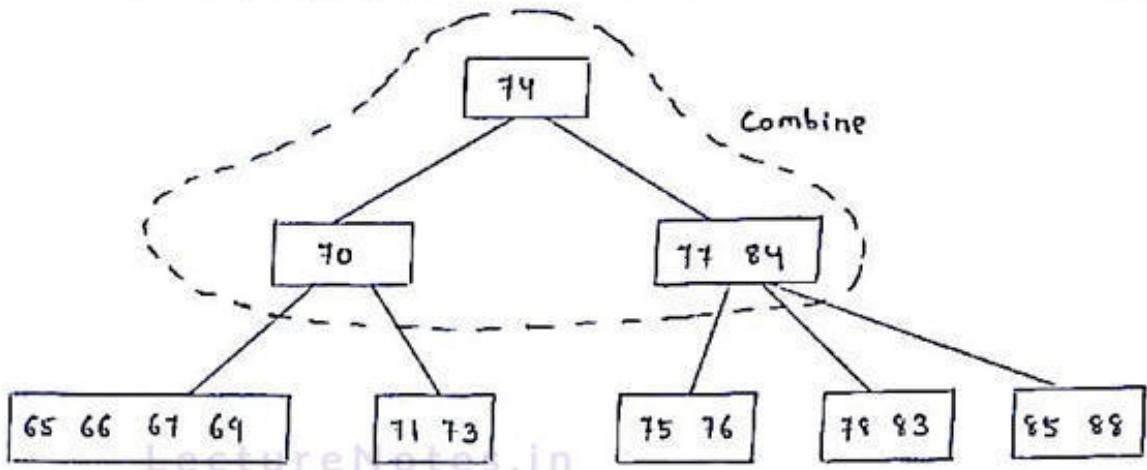


ii) Delete 80



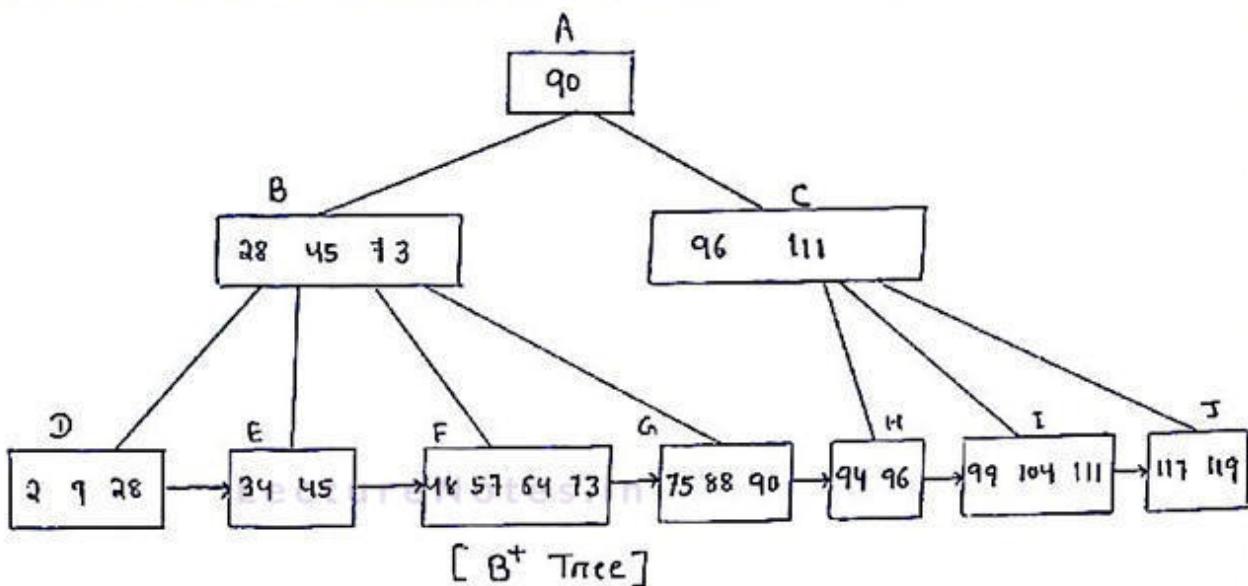
iii) Delete 68





B⁺ Tree :

- One of the drawbacks of B-tree is the difficulty of traversing the keys sequentially.
- A variation of the B-tree is B⁺ tree which retains the rapid random access property of the B-tree while also allowing rapid sequential access.
- In the B⁺ tree, all keys are maintained in leaves, and keys are replicated in non-leaf nodes to define paths for locating individual records.
- The leaves are linked together to provide a sequential path for traversing the keys in the tree.



The above figure depicts a B^+ Tree. To search the record with key 45, the key is first compared with 90 (the key in the root node). Since 45 is less than 90, proceed to node B. 45 is then compared with 28 and then 45 in node B. Since it is less than or equal to 45, proceed to node E.

- Unlike B-tree, the search does not stop when the key is found in B^+ tree.
- In a B -tree, a pointer to the record corresponding to a key is contained with each key in the tree, whether in a leaf or a nonleaf node.
- Thus, once the key is found, the record can be accessed. Consequently the search is not complete until the key is located in a leaf. Therefore when equality is obtained in a nonleaf, the search continues.
- In node E (a leaf), key 45 is located, and from it, the record associated with the key is accessed. If the key in the tree are to be traversed sequentially beginning with key 45, then the pointers in the leaf nodes are to be followed.
- The link list of leaves is called a sequence set.
- The B^+ Tree may be considered to be a natural extension of the indexed sequential file.
- The B^+ tree retains the search and insertion efficiencies of the B-tree but increases the efficiency of finding the next record in the tree from $O(\log n)$ to $O(1)$.



Data Structure Using C

Topic:

Graph

Contributed By:

Mamata Garanayak

Lesson Number : 36

Graph :

→ Graph is a non-linear data structure used to represent the idea of some kind of connection between pairs of objects. A graph consists of:

(a) A collection of "vertices", which we will usually draw as small circles.

(b) A collection of "edges", each connecting two vertices.

→ Tree is also a special kind of graph structure. In tree structure, there is a hierarchical relationship between parent and children i.e. one parent and many children.

→ On the other hand, in a graph, relationship is less restricted. Here relationship is from many parents to many childrens.

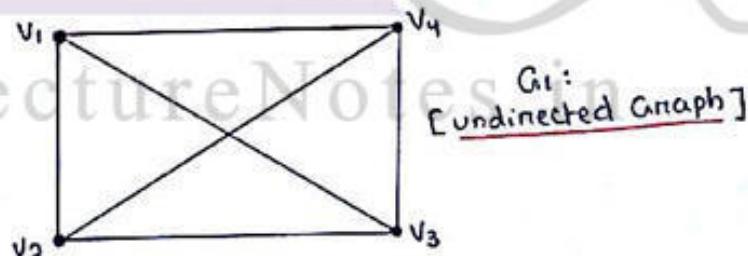
Graph Terminology :

Graph: A Graph $G_1 = (V, E)$ consists of two sets:

i) A set V called set of vertices or nodes and

ii) A set E called set of edges or arcs. This set E is set of pair of elements from V .

Example :



Let us consider the Graph G_1 .

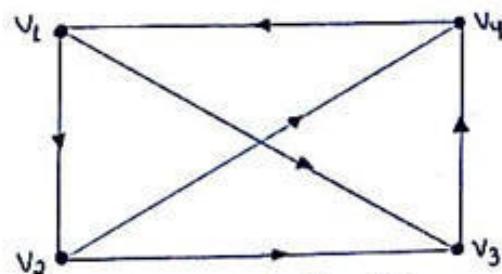
$$V = \{v_1, v_2, v_3, v_4\}$$

$$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$$

Digraph:

A digraph is also called a directed graph. It is a graph G_1 , such that $G_1 = \langle V, E \rangle$, where V is the set of all vertices and E is the set of ordered pair of vertices from V .

Example :



G2 : [Directed Graph]

Graph G2, shown in the above figure is a digraph, where

$$V = \{v_1, v_2, v_3, v_4\}$$

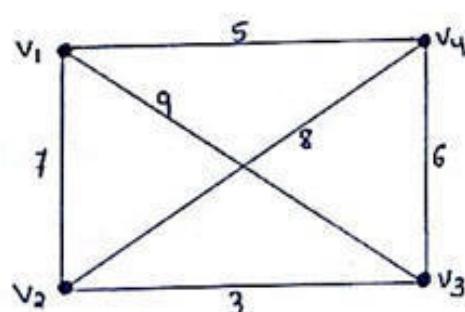
$$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_1), (v_4, v_1), (v_4, v_2)\}$$

Here, if an ordered pair (v_i, v_j) is in E then there is an edge directed from v_i to v_j (indicated by arrow sign).

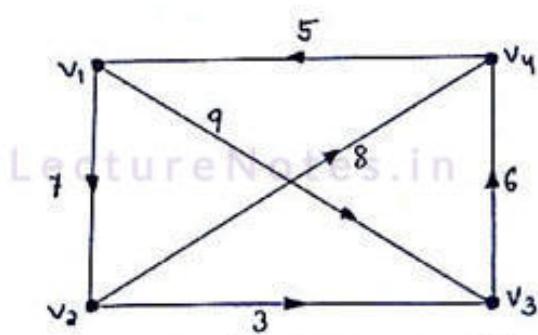
- In case of an undirected graph, pair (v_i, v_j) is unordered i.e. (v_i, v_j) and (v_j, v_i) are the same edge, but in case of a directed graph they correspond to two different edges.
- An undirected graph will be simply termed as graph and directed graph will be termed as digraph.

Weighted Graph:

A graph or digraph is termed as weighted graph if all the edges in it are labelled with some weights.



G3 : [Weighted graph]

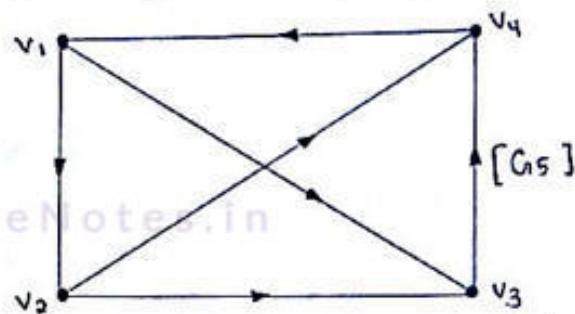


G4 : [Weighted graph]

- The Graph G3 and G4 are two weighted graphs.

Adjacent Vertices:

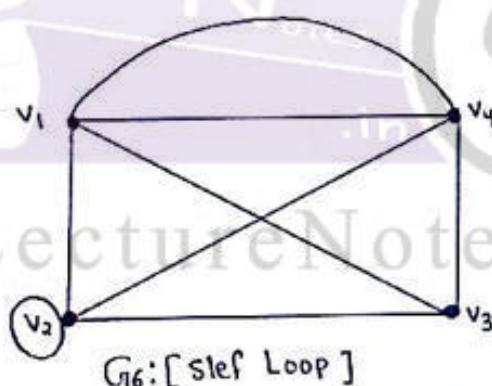
A vertex v_i is adjacent to (or neighbour of) another vertex say v_j , if there is an edge from v_i to v_j .



In the above graph, v_4 is adjacent to v_3 and v_1 , v_1 is not adjacent to v_4 but to v_2 . Similarly the neighbour of v_3 is v_4 but not v_1 and v_2 .

Self Loop:

If there is an edge whose start and end vertices are same, that is (v_i, v_i) is an edge, then it is called a self loop or loop.



In the above graph there is a self loop at vertex v_2 .

Parallel Edges:

If there are more than one edges between the same pair of vertices then they are known as the parallel edges.

For example there are two parallel edges between v_1 and v_4 in graph G6.

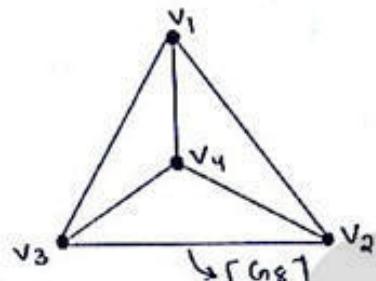
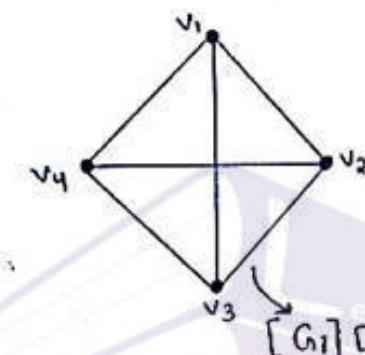
→ A graph which has either self loop or parallel edges or both is called a multigraph. Graph G6 is a multigraph.

Simple Graph (Digraph) :

A graph (digraph) if it does not have any self loop or parallel edges is called a simple graph or digraph .

Complete Graph :

A graph (digraph) G_i is said to be complete , if each vertex v_i is adjacent to every other vertex v_j in G_i . In other words , there are edges from any vertex to all other vertices .

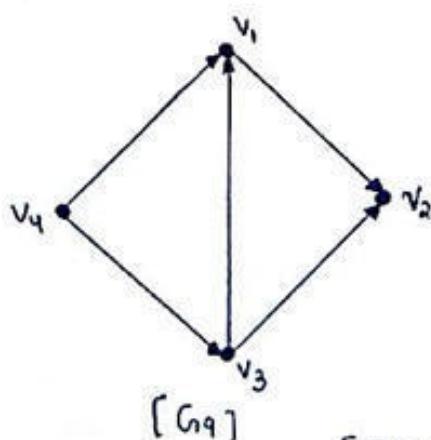


[G_1] [complete Graphs]

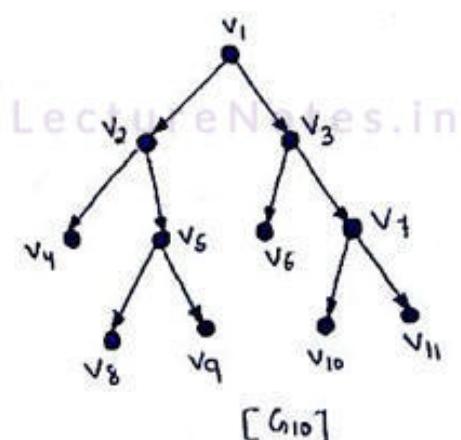
Acyclic Graph :

If there is a path containing one or more edges which starts from a vertex v_i and terminates into the same vertex then the path is known as a cycle . For example there is a cycle in both G_1 and G_2 .

If a graph (digraph) does not have any cycle then it is called acyclic graph .



[G_9]

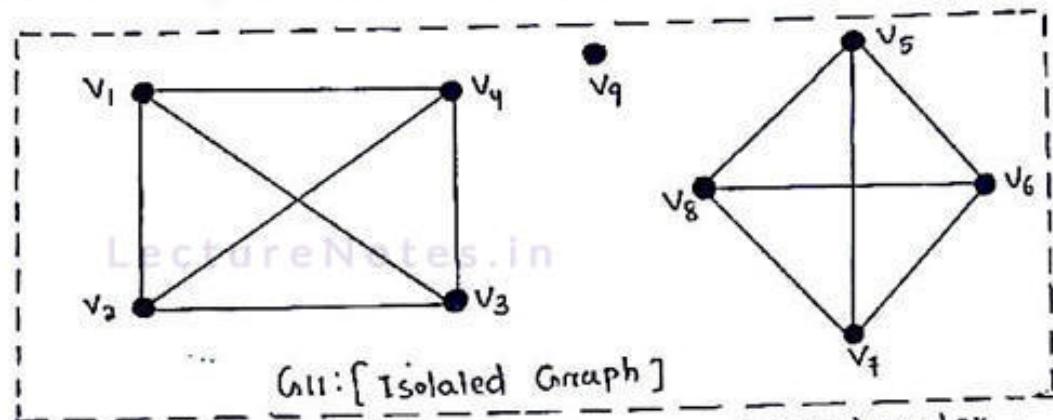


[G_{10}]

[Acyclic Graphs]

Isolated Graph :

A vertex is isolated if there is no edge connected from any other vertex to the another vertex.



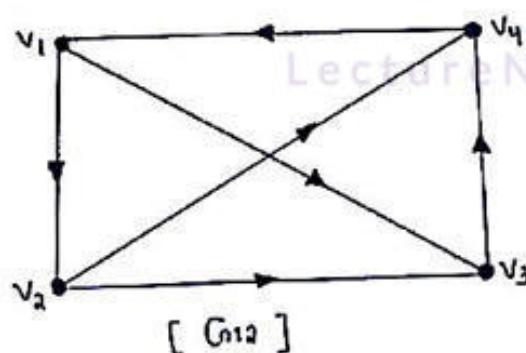
- i) In the above graphs, vertex v_9 is an isolated vertex.

Degree of Vertex :

The number of edges connected with vertex v_i is called degree of vertex v_i and is denoted by $\text{degree}(v_i)$.

Example: $\text{degree}(v_i) = 3$ & in graph G11.

- For a graph, there are two deg. degrees : indegree and outdegree.
 Indegree of v_i is denoted as indegree (v_i) = no. of edges incident to v_i (number of edges ending at v_i).
 Similarly, outdegree (v_i) = no. of edges emanating from v_i (number of edges beginning at v_i).



$$\begin{aligned} \text{indegree}(v_4) &= 2 \\ \text{outdegree}(v_4) &= 1 \end{aligned}$$

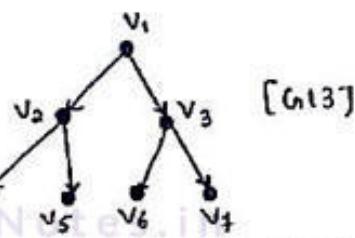
$$\text{indegree}(v_1) = 1, \text{outdegree}(v_1) = 2$$

$$\text{indegree}(v_2) = 1, \text{outdegree}(v_2) = 2$$

$$\text{indegree}(v_3) = 2, \text{outdegree}(v_3) = 1$$

Pendant vertex :

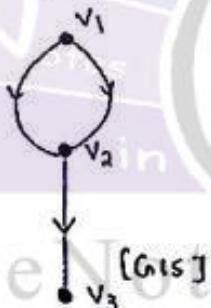
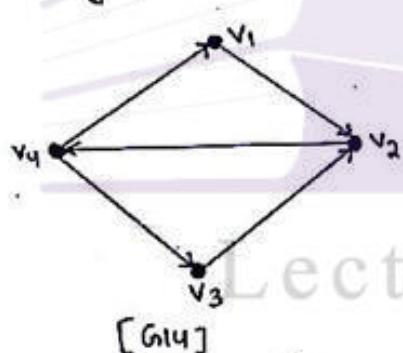
A vertex v_i is pendant if its indegree (v_i) = 1 and outdegree (v_i) = 0.



For example in the above graph, there are four pendant vertices.
These are v_4, v_5, v_6, v_7 .

Connected Graph :

In a graph (not digraph) G_i , two vertices v_i and v_j are said to be connected if there is a path in G_i from v_i to v_j (or v_j to v_i).
→ A graph is said to be connected if there is atleast one path for every pair of distinct vertices v_i, v_j in G_i .



[connected graphs]

→ A directed graph (digraph) with this property is called strongly connected. More precisely, a digraph G_i is said to be strongly connected if for every pair of distinct vertices v_i, v_j in G_i , there is a directed path from v_i to v_j and also from v_j to v_i .

→ Example : G_{14} is a strongly connected graph.

→ If a digraph is not strongly connected but the underlying graph (without direction of the edges) is connected, then the graph is said to be weakly connected. G_{15} is a weakly connected graph.



Data Structure Using C

Topic:

Representation Of Graph

Contributed By:

Mamata Garanayak

Representation Of Graphs:

If we are to write programs for solving problems concerning graphs, then we must first find ways to represent the mathematical structure of a graph as some kind of data structure. There are two standard methods:

- i) sequential or matrix representation
- ii) Link list representation.

i) Matrix Representation:

The graphs can be represented as matrices. There are two most commonly used matrices.

- 1) Adjacency matrix
- 2) Incidence matrix.

i) Adjacency matrix:

The adjacency matrix is a matrix with one row and one column for each vertex. The values of matrix elements are either 0 or 1. The value of 1 for row i and column j implies that the edge (v_i, v_j) exists. A value of 0 indicates that there is no edge between vertices v_i and v_j .

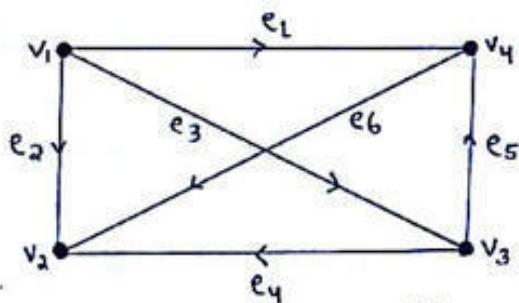
→ Other words we can say that :

if a graph G_1 consists of v_1, v_2, \dots, v_n vertices then the adjacency matrix $A = [a_{ij}]$ of the graph G_1 is the $n \times n$ matrix and can be defined as;

$$a_{ij} = \begin{cases} 1, & \text{if } v_i \text{ is adjacent to } v_j, \text{ therefore it means there} \\ & \text{is an edge between } v_i \text{ and } v_j \\ 0, & \text{if there is no edge between } v_i \text{ and } v_j. \end{cases}$$

Example: Let $V = \{v_1, v_2, v_3, v_4\}$

$$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_3, v_2), (v_3, v_4), (v_4, v_2)\}$$



[A directed graph]

The adjacency matrix of the above graph is :

$$A = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 1 & 1 \\ v_2 & 0 & 0 & 0 \\ v_3 & 0 & 1 & 0 \\ v_4 & 0 & 1 & 0 \end{bmatrix}$$

→ From matrix A, it is clear that number of 1's in an adjacency matrix is equal to the number of edges in the graph. The matrix A contains entries of only 0 and 1, so the matrix is called a bit matrix or Boolean matrix.

→ When one constructing an adjacency matrix for a graph, one must consider the following points.

(a) Adjacency matrix A does not depend on the ordering of the vertices of a graph G, that is different ordering of the vertices may result in a different adjacency matrix. One can obtain the same matrix by interchanging rows and columns.

(b) If a graph G is undirected then the adjacency matrix of G will be symmetric. That is $[a_{ij}] = a[a_{ji}]$ for every i and j.

2) Incidence matrix :

The incidence matrix consists of a row for every vertex and a column for every edge. The values of the matrix are:-
-1, 0, or 1.

→ The R^{th} edge is (v_i, v_j) , R^{th} column has a value 1 in the i^{th} row,
-1 in the j^{th} row and 0 otherwise.

	e_1	e_2	e_3	e_4	e_5	e_6
v_1	1	1	1	0	0	0
v_2	0	-1	0	-1	0	-1
v_3	0	0	-1	1	1	0
v_4	-1	0	0	0	-1	1

Path Matrix :

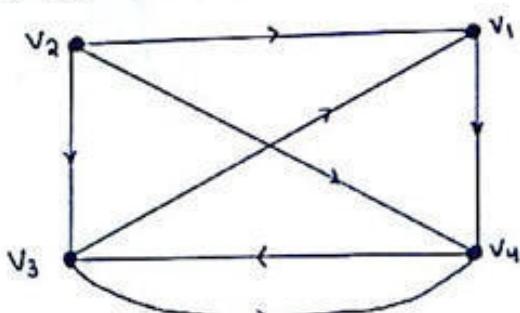
Let G_1 be a simple directed graph with n nodes v_1, v_2, \dots, v_n .
The path matrix or reachability matrix of graph G_1 is the n -square matrix $P = [P_{ij}]$ is defined as:

$P_{ij} = \begin{cases} 1 & \text{if there is a path from node } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$

→ Let A be the adjacency matrix and $P = [P_{ij}]$ be the path matrix of a graph G_1 . Then $P_{ij} = 1$ if and only if there is a non-zero number in the ij entry of the matrix.

$$B_n = A^1 + A^2 + A^3 + \dots + A^n.$$

Example : Find the path matrix of the following graph.



In the above graph no. of nodes $n=4$.
so we have to obtain, A^1, A^2, A^3 and A^4

The adjacent matrix of graph G is :

$$A^1 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 0 & 1 \\ v_2 & 1 & 0 & 1 \\ v_3 & 1 & 0 & 0 \\ v_4 & 0 & 0 & 1 \end{bmatrix}$$

NOTE :

$a_k(i,j)$ = the i,j entry in the matrix A^k

$a_1(i,j) = a_{ij}$ gives the no. of paths of length 1 from node v_i to v_j .

$a_k(i,j) = a_{ij}$ gives the no. of paths of length k from node v_i to v_j .

$$A^2 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 0 & 1 & 0 \\ v_2 & 1 & 0 & 1 & 2 \\ v_3 & 0 & 0 & 1 & 1 \\ v_4 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 1 & 0 & 0 & 1 \\ v_2 & 1 & 0 & 2 & 2 \\ v_3 & 1 & 0 & 1 & 1 \\ v_4 & 0 & 0 & 1 & 1 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 0 & 1 & 1 \\ v_2 & 2 & 0 & 2 & 3 \\ v_3 & 1 & 0 & 1 & 2 \\ v_4 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Now we know $B_n = A^1 + A^2 + A^3 + \dots + A^n$

$$\therefore B_4 = A^1 + A^2 + A^3 + A^4$$

$$B_4 = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 1 & 0 & 2 & 3 \\ v_2 & 5 & 0 & 6 & 8 \\ v_3 & 3 & 0 & 3 & 5 \\ v_4 & 2 & 0 & 3 & 3 \end{bmatrix}$$

By replacing the non-zero entries in B_4 by 1, we get the path matrix P of the graph G .

$$\therefore \text{path Matrix } P = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 1 & 0 & 1 & 1 \\ v_2 & 1 & 0 & 1 & 1 \\ v_3 & 1 & 0 & 1 & 1 \\ v_4 & 1 & 0 & 1 & 1 \end{bmatrix}$$

In the above path matrix, the node v_2 is not reachable from any of the nodes. So the graph G is not strongly connected.

A Graph G is strongly connected iff the path matrix of graph G has no non-zero entries.

Linked Representation Of Graph:

The linked representation of graph contains two lists,

- a node list Node
- an edge list Edge.

Node list: Each element in the list node will correspond to a node in G , and it will be a record of the form:

node	next	adj
------	------	-----

node \rightarrow will be the name or key value of the node.

next \rightarrow will be a pointer to the next node in the list node

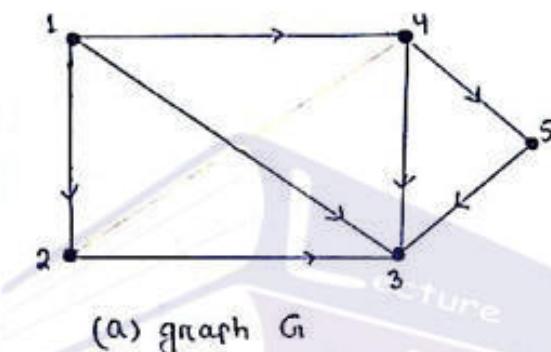
adj \rightarrow will be a pointer to the first element in the adjacency list of the node, which is maintained in the list edge.

Edge list: Each element in the list Edge will correspond to an edge of G_1 and will be a record of the form:



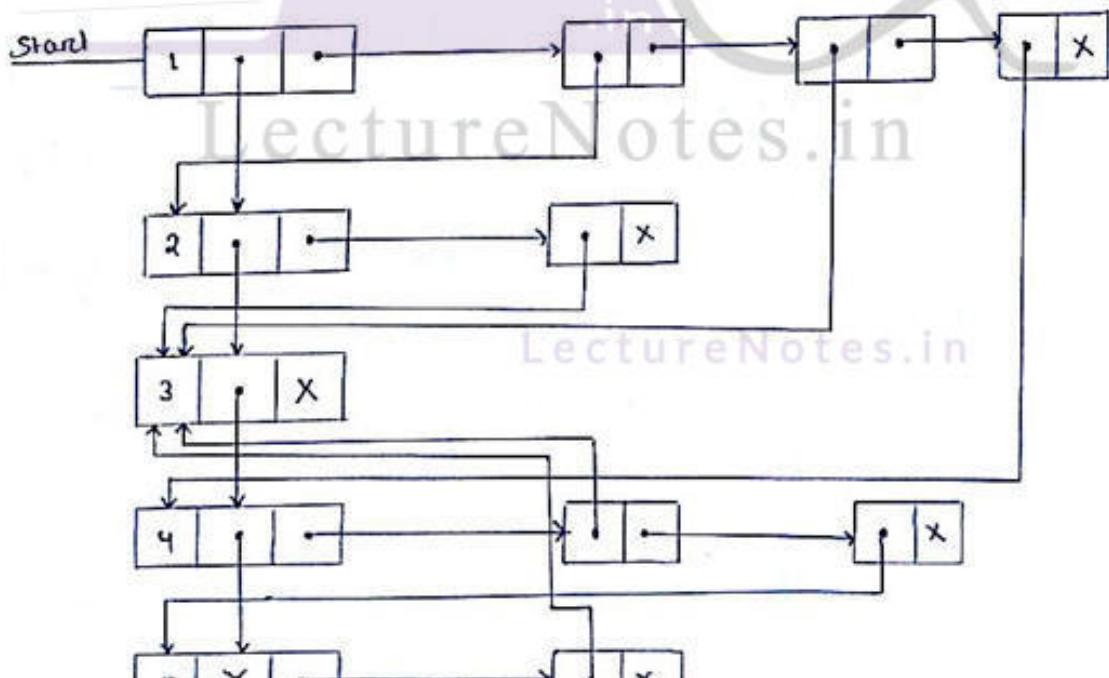
The field dest will point to the location in the list node of the destination or terminal nodes in the same adjacency list.

The field link will link together the edges with the same initial node, i.e. the nodes in the same adjacency list.



Node	Adjacent list
1	2, 3, 4
2	3
3	-
4	3, 5
5	3

(b) Adjacent lists of G_1



(Linked Representation of graph G_1)



Data Structure Using C

Topic:
Traversing A Graph

Contributed By:
Mamata Garanayak

Traversing a Graph :

There are two ways for traversing the graph. These are :

- 1) Breadth - First search (BFS)
- 2) Depth - First Search (DFS)

The breadth-first search will use a queue as an auxiliary structure to hold nodes for future processing, and analogously, the depth-first search will use a stack.

1) Breadth-First Search (BFS) :

Algorithm :

Step1 : Initialize all nodes to the ready state.

Step2 : put the starting node in QUEUE.

Step3 : Repeat the steps 4 and step 5 until the QUEUE is empty.

Step4 : Remove the front node of QUEUE.

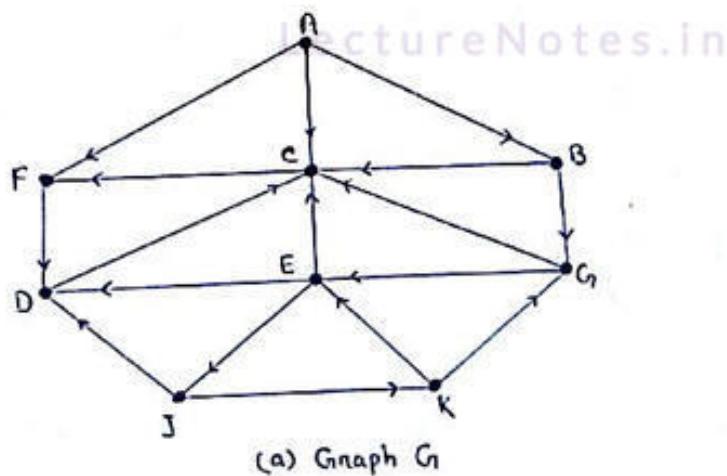
Step5 : Add to the rear of QUEUE all the ~~one~~ neighbors of N.

[End of Step 3 Loop]

Step6 : Exit

NOTE : The BFS algorithm will process only those nodes which are reachable from the starting node.

Example : Consider the following graph



→ The minimum path P can be found by using BFS beginning at A to ending J.

→ During the execution of the search, we will also keep track of the origin of each edge by using an array ORIGIN, together with the QUEUE.

Adjacency List

A :	F, C, B
B :	G, C
C :	F
D :	C
E :	D, C, J
F :	D
G :	C, E
J :	D, K
K :	E, G

(a) Front = 1 QUEUE : A

Rear = 1 ORIGIN : \emptyset

(b) Front = 2 QUEUE : A, F, C, B

Rear = 4 ORIGIN : \emptyset , A, A, A

(c) Front = 3 QUEUE : A, F, C, B, D

Rear = 5 ORIGIN : \emptyset , A, A, A, F

(d) Front = 4 QUEUE : A, F, C, B, D

Rear = 5 ORIGIN : \emptyset , A, A, A, F

(e) Front = 5 QUEUE : A, F, C, B, D, G

Rear = 6 ORIGIN : \emptyset , A, A, A, F, B

(f) Front = 6 QUEUE : A, F, C, B, D, G

Rear = 6 ORIGIN : \emptyset , A, A, A, F, B

(g) Front = 7 QUEUE : A, F, C, B, D, G, E

Rear = 7 ORIGIN : \emptyset , A, A, A, F, B, G

(h) Front = 8 QUEUE : A, F, C, B, D, G, E, J

Rear = 8 ORIGIN : \emptyset , A, A, A, F, B, G, E

We stop as soon as J is added to QUEUE, since J is our end point. Now we backtrack from J using ORIGIN to find the path P. Thus the required path is;

J \leftarrow E \leftarrow G \leftarrow B \leftarrow A ,

2) Depth-First Search : (DFS)

The depth-first search algorithm is similar to the in-order traversal of a binary tree, and very similar to BFS except here a stack is used instead of a queue.

Algorithm :

Step1: Initialize all nodes to the ready state.

Step2: push the starting node onto STACK.

Step3: Repeat Step4 to Step5 until STACK is empty.

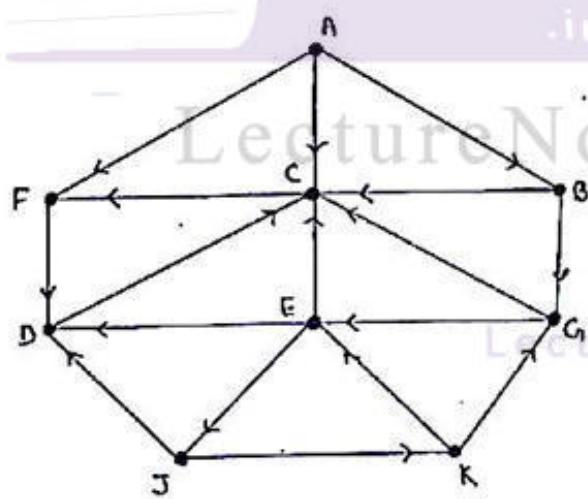
Step4: pop the top node N of STACK and print it.

Step5: push all the neighbors of node N onto stack.

[End of Step3 loop]

Step6: Exit.

Example : Consider the following graph G_1 . Suppose we want to find and print all the nodes reachable from the node J. So in this case we use DFS.



(a) Graph G_1

Adjacency list
A : F, C, B
B : G, C
C : F
D : C
E : D, C, J
F : D
G : C, E
J : D, K
K : E, G

(a) STACK : J

(b) print : J STACK : D, K

- (c) print K STACK : D, E, G
- (d) print G STACK : D, E, C
- (e) print C STACK : D, E, F
- (f) print F STACK : D, E
- (g) print E STACK : D,
- (h) print D

Now the stack is empty, so the depth-First Search (DFS) of graph G_1 starting at J is the nodes which were printed i.e.

J, K, G, C, F, E, D //

The above nodes are the nodes which are reachable from J.

Topological Sorting:

let G_1 be a graph. u and v are the two vertices of graph G_1 .

Consider the relation $<$ on G_1 defined as follows :

$u < v$ - if there is a path from u to v .

such a relation $<$ on graph G_1 is called a partial ordering of G_1 and G_1 with such an ordering is called a partially ordered set or poset. Thus a graph G_1 without cycles may be regarded as a partially ordered set.

On the other hand, suppose G_1 is a partially ordered set with the partial ordering denoted by $<$. Then G_1 may be viewed as a graph whose nodes are the elements of G_1 and whose edges are defined as follows :

(u, v) is an edge in s if $u < v$.

Furthermore, one can show that a partially ordered set G_1 , regarded as a graph has no cycles.

Definition : Let G_1 be a directed graph without cycles (or a partially ordered set). A topological sort T of G_1 is a linear ordering of nodes of G_1 which preserves the original partial ordering of G_1 . That is : if $u < v$ in G_1 (i.e. if there is a path from u to v in G_1) then u comes before v in the linear ordering T .

Algorithm :

Step1 : Find the indegree $\text{INDEG}(N)$ of each node N of G_1 .

Step2 : put in a queue all the nodes with zero indegree.

Step3 : Repeat steps 4 and 5 until the queue is empty.

Step4 : Remove the front node N of the queue by setting
 $\text{front} = \text{front} + 1$.

Step5 : Repeat the following for each neighbor M of the node N

(a) set $\text{INDEG}(M) = \text{INDEG}(M) - 1$

[This deletes the edge from N to M]

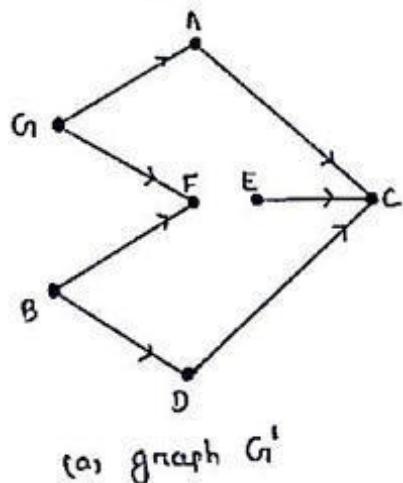
(b) if $\text{INDEG}(M) = 0$ then add M to the rear of the queue.

[End of loop]

[End of step 3 loop]

Step6 : Exit

Example : Let G_1^t be a graph with no. cycles. Thus G_1^t may be regarded as a partially ordered set. Here vertex $G_1 < C$ since there is a path from G_1 to C .



Adjacency List	
A :	C
B :	D, F
C :	
D :	C
E :	C
F :	
G ₁ :	A, F

(b)

1. Find the indegree $\text{INDEG}(N)$ of each node N of graph G_1 .

$$\begin{array}{llll} \text{INDEG}(A) = 1 & \text{INDEG}(B) = 0 & \text{INDEG}(C) = 3 & \text{INDEG}(D) = 1 \\ \text{INDEG}(E) = 0 & \text{INDEG}(F) = 2 & \text{INDEG}(G) = 0 \end{array}$$

2. $\text{Front} = 1$ $\text{Rear} = 3$ $\text{QUEUE} : \underline{B}, E, G$

3.a. Delete front element of QUEUE B.

$$\text{Front} = 1 \quad \text{Rear} = 3 \quad \text{QUEUE} : \underline{B}, E, G$$

3.b. Decrease by 1 the indegree of each neighbor of B.

$$\text{INDEG}(D) = 0 \quad \text{INDEG}(F) = 2 - 1 = 1$$

The neighbor D is added to the rear of QUEUE since its indegree is 0.

$$\text{Front} = 2 \quad \text{Rear} = 4 \quad \text{QUEUE} : \underline{B}, E, G, \underline{D}$$

4.a. $\text{Front} = 3$ $\text{Rear} = 4$ $\text{QUEUE} : \underline{B}, \underline{E}, G, \underline{D}$

$$\text{INDEG}(C) = 3 - 1 = 2$$

5.a. $\text{Front} = 4$ $\text{Rear} = 4$ $\text{QUEUE} : \underline{B}, \underline{E}, \underline{G}, \underline{D}$

$$\text{INDEG}(A) = 1 - 1 = 0 \quad \text{INDEG}(F) = 2 - 1 = 1$$

$$\text{Front} = 4 \quad \text{Rear} = 6 \quad \text{QUEUE} : \underline{B}, \underline{E}, \underline{G}, \underline{D}, A, F$$

6.a. $\text{Front} = 5$ $\text{Rear} = 6$ $\text{QUEUE} : \underline{B}, \underline{E}, \underline{G}, \underline{D}, A, F$

$$\text{INDEG}(C) = 2 - 1 = 1$$

6.b. $\text{INDEG}(C) = 1 - 1 = 0$

7.a. $\text{Front} = 6$ $\text{Rear} = 6$ $\text{QUEUE} : \underline{B}, \underline{E}, \underline{G}, \underline{D}, \underline{A}, F$

$$\text{INDEG}(C) = 1 - 1 = 0$$

$$\text{Front} = 6 \quad \text{Rear} = 7 \quad \text{QUEUE} : \underline{B}, \underline{E}, \underline{G}, \underline{D}, \underline{A}, F, C$$

8.a. $\text{Front} = 7$ $\text{Rear} = 7$ $\text{QUEUE} : \underline{B}, \underline{E}, \underline{G}, \underline{D}, \underline{A}, F, C$

8.b. The node F has no neighbor.

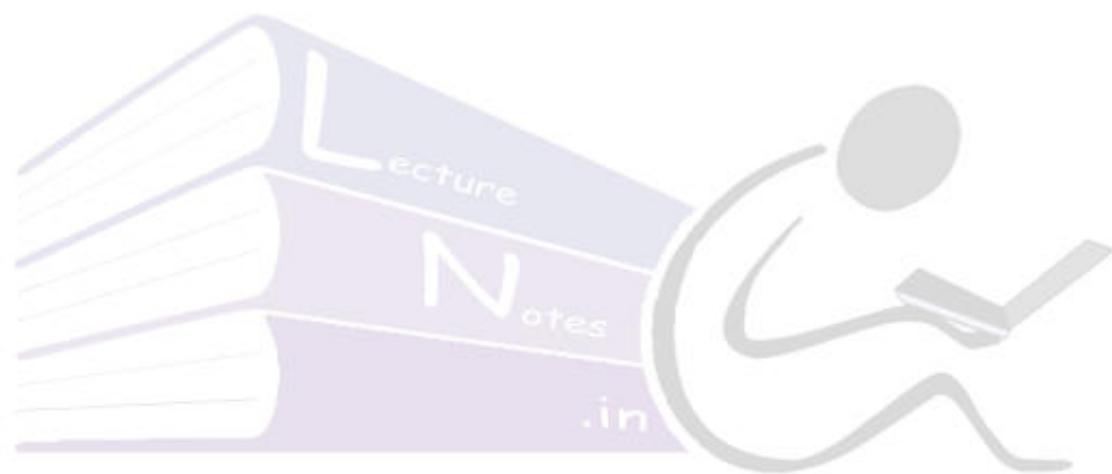
q.a. front = 8 Rear = 7 QUEUE : B.E.G.D.A.F.C

q.b. The node C has no neighbors, so no other changes takes place.

QUEUE has no front element, so the algorithm is completed.

so, the required topological sort T : B, E, G, D, A, F, C

NOTE : The algorithm could have stopped in step 7 b, where Rear is equal to the number of nodes in the graph G.



LectureNotes.in

LectureNotes.in



Data Structure Using C

Topic:
Warshall's Shortest Path

Contributed By:
Mamata Garanayak

Warshall's Shortest path :Algorithm :

PROCEDURE Warshall's-Shortest-path (G, W, n)

// G : Given graph

// W : weight Matrix of the graph

// n : No. of vertex

Step1 : Repeat for $i, j = 1, 2, \dots, n$ [Initialize d]

i) if $W[i, j] = 0$ then set $d[i, j] = \infty$

ii) Else set $d[i, j] = W[i, j]$.

[End of for loop]

Step2 : for $k = 1$ to n

Step3 : for $i = 1$ to n

Step4 : for $j = 1$ to n

Step5 : $d^k[i, j] = \min(d^{k-1}[i, j], d^{k-1}[i, k] + d^{k-1}[k, j])$

[End of Step 2 for loop]

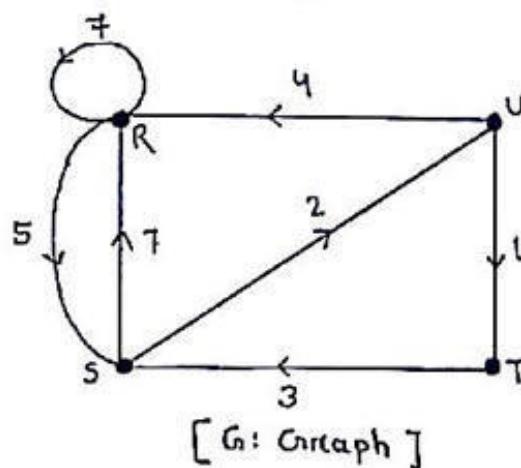
[End of Step 3 for loop]

[End of Step 4 for loop]

Step6 : return $d^{(n)}$.

Step7 : Exit.

Example : Consider the following graph.



For the graph G

$$W = \begin{bmatrix} R & S & T & U \\ R & 7 & 5 & 0 & 0 \\ S & 7 & 0 & 0 & 2 \\ T & 0 & 3 & 0 & 0 \\ U & 4 & 0 & 1 & 0 \end{bmatrix}$$

$$d^{(0)} = \begin{bmatrix} R & S & T & U \\ R & 7 & 5 & \infty & \infty \\ S & 7 & \infty & \infty & 2 \\ T & \infty & 3 & \infty & \infty \\ U & 4 & \infty & 1 & \infty \end{bmatrix} \quad \begin{bmatrix} RR & RS & - & - \\ SR & - & - & SU \\ TS & - & - & - \\ UR & - & UT & - \end{bmatrix}$$

Now $k=1, i=1, j=1$

$$\begin{aligned} d^{(1)}[1,1] &= \min(d^0[1,1], d^0[1,1] + d^0[1,1]) \\ &= \min(7, 7+7) \\ &= \min(7, 14) \\ &= 7 \end{aligned}$$

Now $k=1, i=1, j=2$

$$\begin{aligned} d^{(1)}[1,2] &= \min(d^0[1,2], d^0[1,1] + d^0[1,2]) \\ &= \min(5, 7+5) \\ &= 5 \end{aligned}$$

$k=1, i=1, j=3$

$$\begin{aligned} d^{(1)}[1,3] &= \min(d^0[1,3], d^0[1,1] + d^0[1,3]) \\ &= \min(\infty, 7+\infty) \\ &= \infty \end{aligned}$$

$$\begin{aligned} d^{(1)}[1,4] &= \min(d^0[1,4], d^0[1,1] + d^0[1,4]) \\ &= \min(\infty, 7+\infty) \\ &= \infty \end{aligned}$$

Similarly we can find

$$d^{(1)}[2,1] = 7, d^{(1)}[2,2] = 12, d^{(1)}[2,3] = \infty, d^{(1)}[2,4] = 2$$

$$d^{(1)}[3,1] = \infty, d^{(1)}[3,2] = 3, d^{(1)}[3,3] = \infty, d^{(1)}[3,4] = \infty$$

$$d^{(1)}[4,1] = 4, d^{(1)}[4,2] = 9, d^{(1)}[4,3] = 1, d^{(1)}[4,4] = \infty$$

$$d^{(1)} = \begin{matrix} R & S & T & U \\ \begin{bmatrix} R & 7 & 5 & \infty & \infty \\ S & 7 & 12 & \infty & 2 \\ T & \infty & 3 & \infty & \infty \\ U & 4 & 9 & 1 & \infty \end{bmatrix} & \begin{bmatrix} RR & RS & - & - \\ SR & SRS & - & SU \\ - & TS & - & - \\ UR & URS & UT & - \end{bmatrix} \end{matrix}$$

$$d^{(2)} = \begin{matrix} R & S & T & U \\ \begin{bmatrix} R & 7 & 5 & \infty & 7 \\ S & 7 & 12 & \infty & 2 \\ T & 10 & 3 & \infty & 5 \\ U & 4 & 9 & 1 & 11 \end{bmatrix} & \begin{bmatrix} RR & RS & - & RSU \\ SR & SRS & - & SU \\ TSR & TS & - & TSU \\ UR & URS & UT & URS \\ U & & & U \end{bmatrix} \end{matrix}$$

$$d^{(3)} = \begin{matrix} R & S & T & U \\ \begin{bmatrix} R & 7 & 5 & \infty & 7 \\ S & 7 & 12 & \infty & 2 \\ T & 10 & 3 & \infty & 5 \\ U & 4 & 4 & 1 & 6 \end{bmatrix} & \begin{bmatrix} RR & RS & - & RSU \\ SR & SRS & - & SU \\ TSR & TS & - & TSU \\ UR & UTS & UT & UTSU \end{bmatrix} \end{matrix}$$

$$d^{(4)} = \begin{matrix} R & S & T & U \\ \begin{bmatrix} R & 7 & 5 & 8 & 7 \\ S & 7 & 11 & 3 & 2 \\ T & 9 & 3 & 6 & 5 \\ U & 4 & 4 & 1 & 6 \end{bmatrix} & \begin{bmatrix} RR & RS & RSUT & RSU \\ SR & SURS & SUT & SU \\ TSUR & TS & TSUT & TSU \\ UR & UTS & UT & UTSU \end{bmatrix} // \end{matrix}$$



Data Structure Using C

Topic:
Bubble Sort

Contributed By:
Mamata Garanayak

Bubble Sort:

Lesson Number: 40

```
# include <stdio.h>
# include <conio.h>
# define SIZE 20
void main()
{
    int arr[SIZE];
    int i, j, temp;
    clrscr();
    printf("Enter the elements of the array :\n");
    for( i=0; i<SIZE ; i++ )
        scanf("%d", &arr[i]);
    for( i=0; i< SIZE-1 ; i++ )
    {
        for( j=0; j< SIZE-1-i; j++ )
        {
            if( arr[j] > arr[j+1] )
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    printf("The sorted array is :\n");
    for( i=0; i<SIZE ; i++ )
        printf("%d", arr[i]);
    printf("\n");
    getch();
}
```

Time complexity of Bubble Sort = $O(n^2)$,

Example: Let us take a list of element in unsorted order and we want to sort them by using bubble sort.

Elements of the array are :

40 20 50 60 30 10

pass 1: $i = 0$

	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$
0	40	20	50	60	30
1	20	40	50	60	30
2	50	60	30	10	40
3	60	30	10	20	50
4	30	10	20	40	60
5	10				

In pass 1 $\text{arr}[0] > \text{arr}[1]$, Exchange

$\text{arr}[1] > \text{arr}[2]$, No

$\text{arr}[2] > \text{arr}[3]$, No

$\text{arr}[3] > \text{arr}[4]$, Exchange

$\text{arr}[4] > \text{arr}[5]$, Exchange.

pass 2: $i = 1$

	$j = 0$	$j = 1$	$j = 2$	$j = 3$
0	20	40	50	30
1	40	20	30	50
2	50	30	10	60
3	30	10	20	40
4	10	60	40	50
5	60			

In pass 2 $\text{arr}[0] > \text{arr}[1]$, No
 $\text{arr}[1] > \text{arr}[2]$, No
 $\text{arr}[2] > \text{arr}[3]$, Exchange
 $\text{arr}[3] > \text{arr}[4]$, Exchange

pass 3 : $i = 2$

	$j = 0$	$j = 1$	$j = 2$
0	20	20	20
1	40	40	30
2	30	30	40
3	10	10	10
4	50	50	50
5	60	60	60

Ex Ex

In pass 3 $\text{arr}[0] > \text{arr}[1]$, No
 $\text{arr}[1] > \text{arr}[2]$, Exchange
 $\text{arr}[2] > \text{arr}[3]$, Exchange.

pass 4 : $i = 3$

	$j = 0$	$j = 1$
0	20	20
1	30	30
2	10	10
3	40	40
4	50	50
5	60	60

Ex

In pass 4 $\text{arr}[0] > \text{arr}[1]$, No
 $\text{arr}[1] > \text{arr}[2]$, Exchange

pass 5 : $i = 4$

	$j = 0$
0	20
1	10
2	30
3	40
4	50
5	60

Ex

On pass 5. arr[0] > arr[1], Exchange .

Finally after pass 5 we got the sorted array :

10 20 30 40 50 60

Selection Sort :

The selection sort is also known as push-down sort .

```
# include < stdio.h>
# include <conio.h>
void Selectionsort ( int[], int );
Void main()
{
    int a[20], n, i ;
    clrscr();
    printf (" Enter the no. of elements to be sorted : ");
    scanf ("%d", &n);
    printf (" Enter 1-d elements ", n);
    for (i=0; i<n ; i++)
        scanf ("%d", &a[i]);
    Selectionsort ( a, n );
    printf (" The sorted array is \n");
    for (i=0; i<n ; i++)
        printf (" %d", a[i]);
    getch();
}
*
```

☞ The time complexity of selection sort = $O(n^2)$ //

```
* void selection-sort ( int a[], int n )
{
    int i, pos, j, large;
    for ( i = n-1 ; i > 0 ; i-- )
    {
        large = a[0];
        pos = 0;
        for ( j = 1 ; j <= i ; j++ )
        {
            if ( a[j] > large )
            {
                large = a[j];
                pos = j;
            }
        }
        a[pos] = a[i];
        a[i] = large;
    }
}
```

LectureNotes.in

LectureNotes.in

Example: 30, 20, 35, 14, 90, 25, 32

Store all the elements in an array.

30	20	35	14	90	25	32
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 1: The largest element in between $a[0]$ and $a[6]$ = 90 and its position = 4.

So exchange $a[4]$ with $a[6]$.

30	20	35	14	32	25	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 2: Between $a[0]$ to $a[5]$, the largest element = 35 & its position = 2
Exchange $a[2]$ with $a[5]$.

30	20	25	14	32	35	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 3: Between $a[0]$ to $a[4]$, the largest element = 32 & its position = 4. So no exchange required.

30	20	25	14	32	35	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 4: Between $a[0]$ to $a[3]$, largest = 30 & its position = 0
Exchange $a[0]$ with $a[3]$.

14	20	25	30	32	35	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 5: Between $a[0]$ to $a[2]$, largest = 25 & its position = 2
No exchange required.

14	20	25	30	32	35	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 6:

Between $a[0]$ and $a[1]$, largest = 20. its position = 1
No exchange required.

14	20	25	30	32	35	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

→ Sorted array

Insertion Sort :

```
# include <stdio.h>
# include <conio.h>
void insertionSort (int [], int );
void main()
{
    int x[20], n, i;
    clrscr();
    printf (" Enter the number of no.s to be inserted : ");
    for (i=0; i<n; i++)
        scanf ("%d", &x[i]);
    insertionSort (x, n);
    printf (" Enter the sorted array is : \n");
    for (i=0; i<n; i++)
        printf ("%d", x[i]);
    getch();
}
void insertionSort ( int a[], int n)
{
    int i, j, key;
    for (j=1; j<n; j++)
    {
        key = a[j];
        i = j-1;
        while ( (i>-1) && (a[i] > key))
        {
            a[i+1] = a[i];
            i = i-1;
        }
        a[i+1] = key;
    }
}
```

→ The time complexity of insertion sort :

Best case time complexity = $O(n)$.

Worst case time complexity = $O(n^2)$.

Example : The elements of the array :

30 20 35 14 90 25 32

Store these elements in an array.

LectureNotes.in

30	20	35	14	90	25	32
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 1 : $a[1] > a[0]$. (Yes) (No)

so interchange the position of the elements

20	30	35	14	90	25	32
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 2 : $a[2] > a[1]$. (Yes).

so the position of the element remains same.

$a[2] > a[0]$. (Yes)

so also the position of the element remains same.

20	30	35	14	90	25	32
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 3 : $a[3]$ is less than $a[0], a[1], a[2]$, so insert $a[3]$ before $a[0]$.

14	20	30	35	90	25	32
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 4 : $a[4]$ is greater than $a[3]$, so the position of the elements remain same.

14	20	30	35	90	25	32
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]



Data Structure Using C

Topic:
Quick Sort

Contributed By:
Mamata Garanayak

Lesson Number : 41

Quicksort :

Quicksort is an algorithm of the divide and conquer type, i.e. the problem of sorting a set is reduced to the problem of sorting two smaller sets.

Example :

Suppose A is the list of 12 numbers.

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

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

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

- Beginning with 22, next scan the list in the opposite direction i.e. from left to right, comparing each element with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list.

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

- Beginning with this time with 55, now scan from right to left until meeting the first number less than 44, is 40, interchange them.

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

- Beginning with 40, scan from left to right. The first number greater than 44 is 77. interchange 44 and 77.

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

→ start with 44, scan right to left, find the first number less than 44. we do not meet such number. so 44 is reached to its final position. Now all elements present before 44 are smaller than 44 and after 44 are greater than 44. So the list is divided into two sublist having 44 in its correct position.

22	33	11	40	44	90	77	60	99	55	88	66
First sublist						Second sublist					

→ By solving the above two sublists individually by using the above method we get the sorted array.

C - procedure :

```

void quicksort ( int a[], int p, int n )
{
    int q;
    if ( p < n )
    {
        q = partition ( a, p, n );
        quicksort ( a, p, q );
        quicksort ( a, q+1, n );
    }
}
int partition ( int a[], int p, int n )
{
    int x, i, j, temp;
    x = a[p];
    i = p-1;
    j = n+1;
    while ( 1 )
    {
        do
        {
            j = j-1;
        } while ( a[j] > x );
    }
}

```

```
do
{
    i = i+1;
} while (a[i] < x);

if (i < j)
{
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
else
    return (j);
}
```



LectureNotes.in

LectureNotes.in

Merge Sort :

Mergesort (a, f, l)

// a: Given array

// f: It is the first index of the array.

// l: It is the last index of the array.

Step 1: $p = (f+l)/2$

Step 2: if ($f == l$) then return

Step 3: else

i) Mergesort (a, f, p)

ii) Mergesort (a, p+1, l)

iii) merge-array (a, f, p, l)

[End of if .. else]

Step 4: Exit.

merge-array (a, f, p, l)

Step 1: i) $i = f$

ii) $j = p+1$

iii) $k = 0$

Step 2: while ($i \leq p$ & $j \leq l$)

i) if ($a[i] \leq a[j]$)

a) $c[k] = a[i]$

b) $k++$

c) $i++$

ii) else

a) $c[k] = a[j]$

b) $j++$

[End of while] c) $k++$

Step 3: while ($i \leq p$)

a) $c[k] = a[i]$

b) $K++$
 c) $i++$
 [End of while]

Step 4 : while ($i \leq l$)

a) $c[K] = a[i]$
 b) $K++$
 c) $i++$.

[End of while]

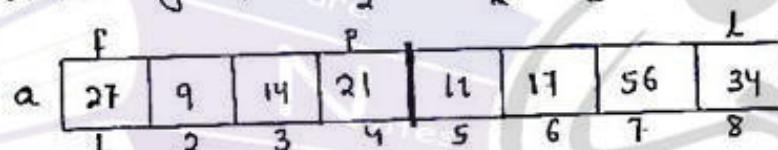
Step 5 : Exit

⇒ The time complexity of mergesort algorithm is : $O(n \log n)$

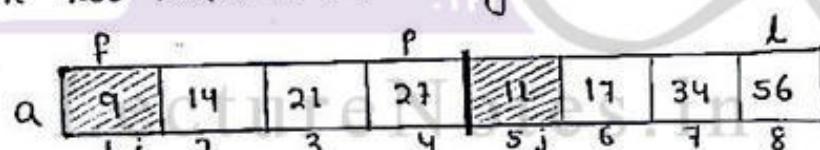
a) Example : The elements of the array are :

27 9 14 21 11 17 56 34

First divide by $p = \frac{f+l}{2} = \frac{1+8}{2} = \frac{9}{2} = 4$



Sort the two halves of the array a.



Merge-array ($a, l, 4, 8$)

Step 1 : $i = 1$

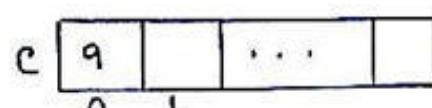
$j = 5$

$K = 0$

Step 2 : while ($i \leq 4 \text{ and } j \leq 8$) (true)

i) if ($a[i] \leq a[j]$) true

a) $c[0] = a[i]$
 b) $K = 1$
 c) $i = 2$



\rightarrow while ($2 \leq 4$ & $5 \leq 8$) (true)
 if ($14 \leq 11$) (false)

else

- i) $c[1] = 11$
- ii) $K = 2$
- iii) $j = 6$

f	a	P	l
	9 14 21 27 11 17 34 56	1 2 i 3 4 5 j 6 7 8	

C
9 11 ...

0 1

\rightarrow while ($2 \leq 4$ & $6 \leq 8$) (true)
 if ($14 \leq 11$) (true)

- i) $c[2] = 14$
- ii) $K = 3$
- iii) $i = 3$

f	a	P	l
	9 14 21 27 11 11 34 56	1 2 i 3 4 5 6 j 7 8	

C
9 11 14 ...

0 1 2

⋮

\rightarrow while ($3 \leq 4$ & $6 \leq 8$) (true)

if ($21 \leq 11$) false
 else

- i) $c[3] = 17$
- ii) $K = 4$
- iii) $j = 7$

f	a	P	l
	9 14 21 27 11 17 17 34 56	1 2 3 i 4 5 6 j 7 8	

C
9 11 14 17 ...

0 1 2 3

\rightarrow while ($3 \leq 4$ & $7 \leq 8$) (true)

if ($21 \leq 34$) (true)
 i) $c[4] = 21$
 ii) $K = 5$
 iii) $i = 4$

f	a	P	l
	9 14 21 21 27 11 17 34 56	1 2 3 i 4 5 6 7 j 8	

C
9 11 14 17 21 ...

0 1 2 3 4

\rightarrow while ($4 \leq 4$ & $7 \leq 8$) (true)

if ($21 \leq 34$) (true)
 i) $c[5] = 27$
 ii) $K = 6$
 iii) $i = 5$

f	a	P	l
	9 14 21 27 11 17 34 56	1 2 3 4 i 5 6 7 j 8	

C
9 11 14 17 21 27 ...

0 1 2 3 4 5

\rightarrow while ($5 \leq 4$ & $7 \leq 8$) (false)

Step 3:
 \rightarrow while ($5 \leq 4$) (false)

Step 4: while ($7 \leq 8$) (true)

i) $c[6] = 34$

ii) $k = 7$

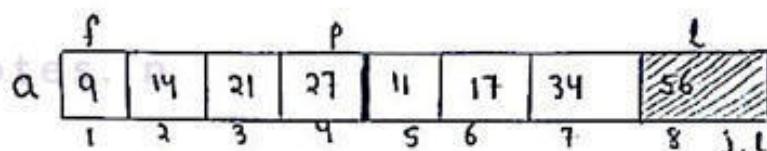
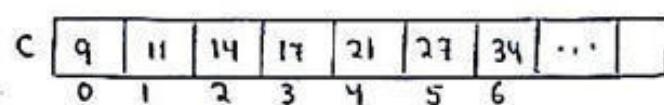
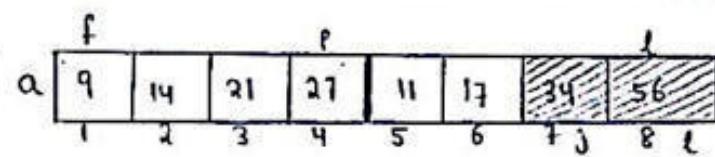
iii) $j = 8$

→ while ($8 \leq 8$) (true)

i) $c[7] = 56$

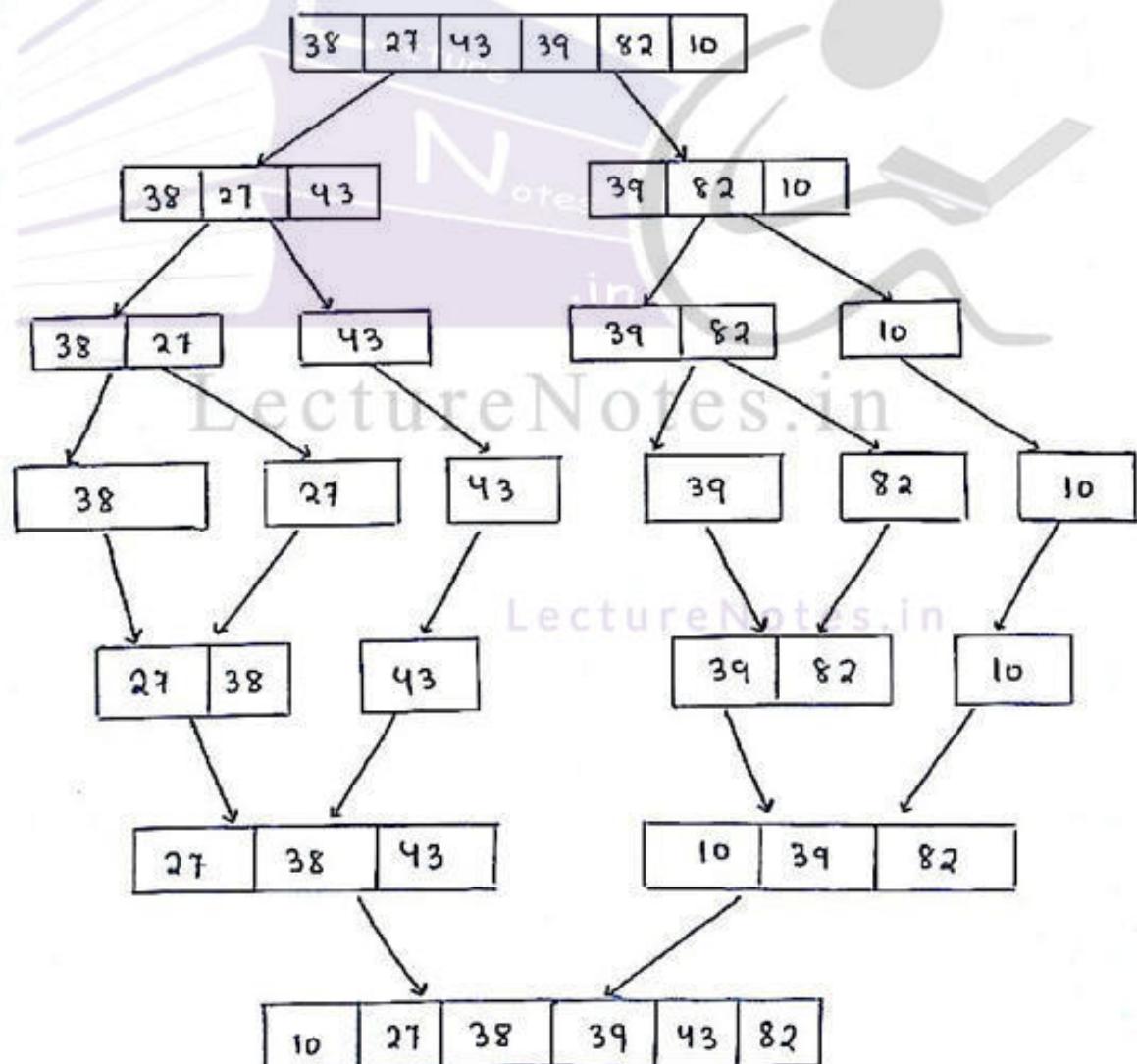
ii) $k = 8$

iii) $j = 9$



Sorted array = c

9	11	14	17	21	27	34	56
0	1	2	3	4	5	6	7



Q. If the total no. of elements in an array is 16 then how many no. of comparisons are required for sorting the array using mergesort?

Soln Given no. of elements in an array = $n = 16$

$$T(n) = 2T(\frac{n}{2}) + n$$

$$\Rightarrow T(16) = 2T(8) + 16$$

$$\Rightarrow T(8) = 2T(4) + 8$$

$$\Rightarrow T(4) = 2T(2) + 4$$

$$\Rightarrow T(2) = 2T(1) + 2$$

$$T(1) = 0.$$

$$\therefore T(2) = 0 + 2 = 2$$

$$T(4) = 2 \times 2 + 4 = 8$$

$$T(8) = 2 \times 8 + 8 = 24$$

$$T(16) = 2 \times 24 + 16 = 48 + 16 = \underline{64} //$$

PASS 5: $a[5]$ is less than $a[2]$, $a[3]$, and $a[4]$, therefore $a[5]$ is inserted before $a[2]$

14	20	25	30	35	90	32
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$

PASS 6: $a[6]$ is less than $a[5]$ and $a[4]$. Therefore, $a[6]$ is inserted before $a[4]$

LectureNotes.in

14	20	25	30	32	35	90
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$

→ sorted array

Quick Sort :

Quicksort (a, p, n)

// a : Given array - $a[p] \dots \dots a[n]$

// p : 1st element of the array

// n : last element of the array

Step1 : if ($p < n$) then

 i) $q = \text{partition}(a, p, n)$

Step2 : Quicksort ($a, p, q-1$);

Step3 : Quicksort ($a, q+1, n$)

Step4 : Exit .

partition (a, p, n)

Step1 : i) $x \leftarrow a[n]$

 ii) $i = p-1$

Step2 : for ($j \leftarrow p$ to $n-1$)

 i) do if ($a[j] \leq x$) then

 a) $i = i + 1$

 b) exchange $a[i] \leftrightarrow a[j]$

[End for]

Step3 : Exchange $a[i+1] \leftrightarrow a[n]$

Step4 : return $i+1$.

Step5 : Exit

```
#include <stdio.h>
#include <Conio.h>

void quicksort ( int *a , int p , int n ) ;
int partition ( int *a , int p , int n ) ;

Void main( )
{
    int *a , Count , n ;
    Clrscr();
    printf (" Enter the no. of elements to be sorted") ;
    Scanf (" %d" , &n ) ;
    printf (" Enter the numbers\n") ;
    for ( Count = 0 ; Count < n ; Count ++ )
        scanf (" %d" , &a[Count] );
    quicksort ( a , 0 , n - 1 ) ;

    printf (" The elements after sorting is : " ) ;
    for ( count = 0 ; Count < n ; Count ++ )
        printf (" .%d" , a[Count] ) ;
    getch();
}

void quicksort ( int *a , int p , int n )
{
    int q ;
    if ( p < n )
    {
        q = partition ( a , p , n ) ;
    }
}
```

```
quicksort( a, p, q-1 );
quicksort( a, q+1, n );
}

int partition( int *a, int p, int n )
{
    int i, x, y, t;
    x = p+1;
    y = n;
    i = a[p];
    while ( y >= x )
    {
        while ( a[x] < i )
            { x++; }
        while ( a[y] > i )
            { y--; }
        if ( y > x )
        {
            t = a[x];
            a[x] = a[y];
            a[y] = t;
        }
        t = a[p];
        a[p] = a[y];
        a[y] = t;
    }
    return y;
}
```



The time complexity of quicksort is:

The bestcase time complexity = $\Theta(n \log n)$

The worstcase time complexity = $\Theta(n^2)$

Example: The elements of the array are :

2 8 7 1 3 5 6 4

Store the elements in the array.

i	p _j	n
2	8	7
1	3	5
3	6	4
a[1]	a[2]	a[3]
a[4]	a[5]	a[6]
a[7]	a[8]	

partition(a, i, 8)

1. $x \leftarrow 4$

2. $i = 1 - 1 = 0$

3. for $j = 1$ to 7,

if ($2 \leq 4$) (true)

$i = 1$

exchange $2 \leftrightarrow 2$

for $j = 2$ to 7

if ($8 \leq 4$) (false)

i	j	n
2	8	7
a[1]	a[2]	a[3]
a[4]	a[5]	a[6]
a[7]	a[8]	

for $j = 3$ to 7

if ($7 \leq 4$) (false)

i	j	n
2	8	7
a[1]	a[2]	a[3]
a[4]	a[5]	a[6]
a[7]	a[8]	

for $j = 4$ to 7

if ($1 \leq 4$) (true)

$i \leftarrow 2$

exchange $8 \leftrightarrow 1$

i	j	exchange
2	8	7
a[1]	a[2]	a[3]
a[4]	a[5]	a[6]
a[7]	a[8]	

for $j = 5$ to 7

if ($3 \leq 4$) (true)

$i = 3$

exchange $7 \leftrightarrow 3$

i	j	Exchange
2	1	7
a[1]	a[2]	a[3]
a[4]	a[5]	a[6]
a[7]	a[8]	

for $j = 6$ to 7
 if ($s \leq 4$) (false)

		<i>i</i>		<i>j</i>		<i>n</i>
a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

for $j = 7$ to 7
 if ($s \leq 4$) (false)

		<i>i</i>		<i>j</i>		<i>n</i>
a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

4. Exchange $8 \leftrightarrow 4$

LectureNotes.in

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]

↑
Pivot Element

The recursively sort $a[1] \dots a[3]$ and $a[5] \dots a[8]$ by using partition (a, p, n) and finally we get the sorted array :

1	2	3	4	5	6	7	8

$a[1]$ $a[2]$ $a[3]$ $a[4]$ $a[5]$ $a[6]$ $a[7]$ $a[8]$

- ⇒ Partition means, choose a pivot element and place it in such a position in an array such that all the elements less than the pivot elements are placed at left side of that and all the elements greater than the pivot elements are placed at the right side of that.



Data Structure Using C

Topic:
Radix Sort

Contributed By:
Mamata Garanayak

Lesson Number : 42

Radix Sort :

- This sort is based on the values of the actual digits in the positional representations of the numbers being sorted.
- For example, the number 367 in decimal notation is written with a 3 in the hundreds position, a 6 in the tens position, and a 7 in the ones position.
- The larger of the two such integers of equal length can be determined as follows : start at the most significant digit and advance through the least significant digit as long as the corresponding digits in the two numbers match.
- The number with the larger digit in the first position in which the digits of the two numbers do not match is the larger of the two numbers. Of course, if all the digits of both numbers match, the numbers are equal.

Example :

151, 60, 875, 342, 12, 477, 689, 128, 15

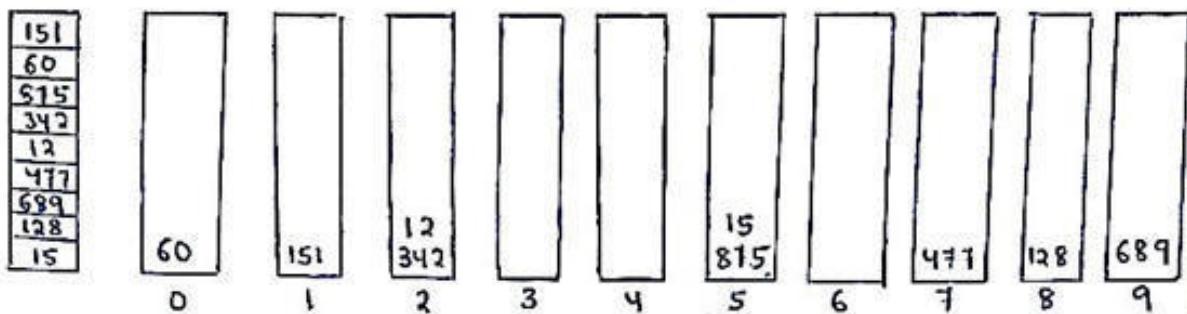
- To sort decimal numbers, we need ten buckets, since the base or radix is ten. These buckets are numbered 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- The elements are stored in an array as follows.

151	60	875	342	12	477	689	128	15
-----	----	-----	-----	----	-----	-----	-----	----

In this case, since the largest number is 875 having 3 digits, so it requires 3 passes to complete the sorting process.

Pass 1 :

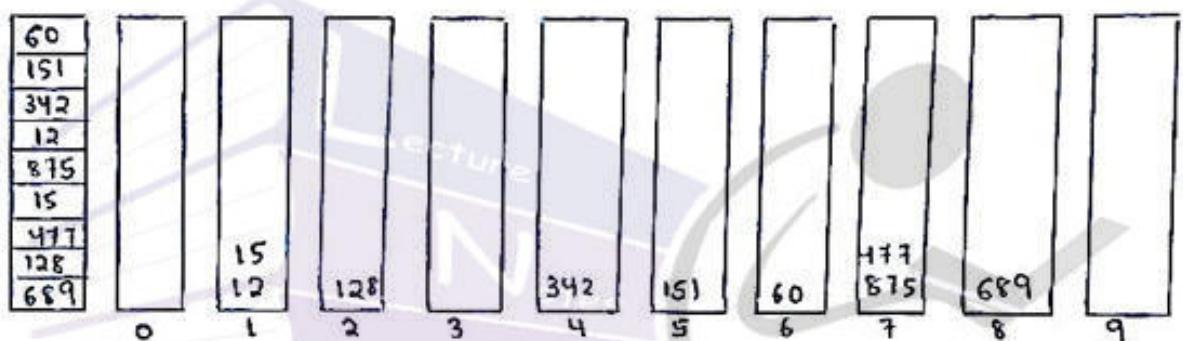
In this step we have to place the numbers in corresponding buckets depends on the least significant digits. If the least significant digit is 5 the number is put in bucket 5.



After pass 1, the numbers are;

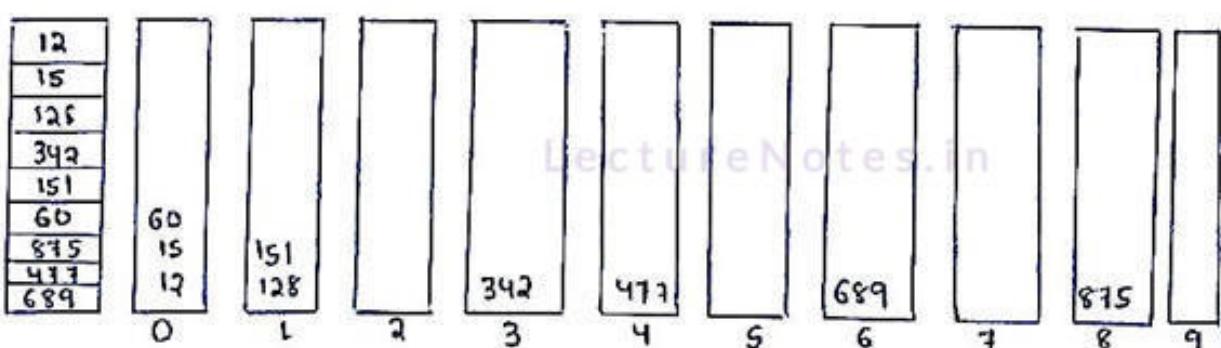
60, 151, 342, 12, 815, 15, 477, 128, 689

Pass 2: In this step we have to place the numbers in corresponding buckets depends on the second digit.



The output of pass 2 is 12, 15, 128, 342, 151, 60, 875, 477, 689.

Pass 3: In this step we have to place the numbers in corresponding buckets depends on the most significant digits.



The output of pass 3 is 12, 15, 60, 128, 151, 342, 477, 689, 875.

After three passes the numbers in the array are sorted.

C-procedure :

```
Void Radix-Sort ( int a[], int n )
{
    int bucket[10][5], buck[10];
    int i, j, k, l, num, div, range, pacs;
    div = 1;
    num = 0;
    range = a[0];
    for (i=0; i<n; i++)
    {
        if (a[i] > range)
            range = a[i];
    }
    while (range > 0)
    {
        num = num + 1;
        range = range / 10;
    }
    for (pacs = 0; pacs < num; pacs++)
    {
        for (k=0; k<10; k++)
            buck[k] = 0;
        for (i=0; i<n; i++)
        {
            l = (a[i]/div) % 10;
            bucket[l][buck[l]++] = a[i];
        }
        c=0;
        for (k=0; k<10; k++)
        {
            for (j=0; j< buck[k]; j++)
                a[c++] = bucket[k][j];
        }
        div = div * 10;
    }
}
```

Searching :

- Information retrieval is one of the most important applications of computers.
- Frequently, it becomes necessary to search a list of records to identify a particular record.
- Usually, each record contains a field whose value is unique, to distinguish among the records, are known as keys.
- A particular record can be identified when the key value of that record is equal to a given input value. This operation is known as searching.

Types of Searching :

The searching problem fall into two cases.
If there are many records, perhaps each one quite large, then it will be necessary to store the records in files on disk or tape, external to the primary memory. This case is called external searching.
In the other case, the records to be searched are stored entirely in the primary memory. This case is called internal searching.
Here we consider only internal searching.

Types of searching techniques:

1. Sequential Search:

- The simplest way to do a search is to begin at one end of the list and scan down it until the desired key is found or the other end is reached.
- This search is applicable to a table organized either as an array or as a linked list.
- This method of searching is also known as linear searching.

- Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program, we want the space to be available for future. One way to bring this about is to immediately reinsert the space into free storage list. But this method is time consuming for the operating system.
- The operating system of a computer may periodically collect all the deleted space onto the free storage list. A technique which does this collection called garbage collection. It is one of the principal methods used in automatic list management.
- The garbage collection process uses a program which is written inside operating system. This routine searches through all of the nodes in the system identifies those that are no longer accessible from an external pointer and restores the in-accessible nodes to the available pool.
- It is done in two phases:
 - a) Marking phase
 - b) Collection phase.
- a) Marking phase:
First the computer (OS) runs through all lists, tagging all marking those cells which are in use. Marking algorithms marks all directly accessible nodes and also all indirectly accessible nodes.
- b) Collection phase:
It involves collecting all untagged or unmarked space onto the free storage list. It sequentially freeing all unmarked nodes.
- The garbage collection may take place when there is no space or minimum space available at free pool.

Search for an element using sequential search:

```
# include <stdio.h>
# include <conio.h>
int sequential-search( int[], int, int );
void main()
{
    int x[20], n, i, p, key;
    clrscr();
    printf(" Enter the number of elements :");
    scanf(" %d", &n);
    printf(" Enter %d elements\n", n);
    for( i=0; i<n; i++ )
        scanf(" %d", &x[i]);
    printf(" Enter the element to be searched");
    scanf(" %d", &key);
    p = sequential-search( x, n, key );
    if( p == -1 )
        printf(" The search is unsuccessful");
    else
        printf(" %d is found at location %d", key, p);
}
int sequential-search( int a[], int n, int k )
{
    int i;
    for( i=0; i<n; i++ )
    {
        if( k == a[i] )
            return i;
    }
    return (-1);
}
```

2. Binary Search :

- sequential search is simple method for searching an element in an array.
- This is efficient for a small list of elements, but highly inefficient for longer lists.
- In the worst case, we will have to make n comparisons, so as to search for the last element in the list.
- Binary search is very efficient search technique which works for sorted lists.
- In this method, we make a comparison between key and the middle element of the array.
- Since, the array is sorted, this comparison results either in a match between key and the middle element of array or identifying the left half or the right half of the array to which the desired element may belong.
- In this case, when the current element is not equal to the middle element of the array, the procedure is repeated on the half in which the desired element is likely to be present, proceeding in this way, either the element is detected or the final division leads to a half consisting of no element.
- There may be an ambiguity over the determination of the middle element of an array when the number of elements is even. In this case, there are two elements in the middle. However, an arbitrary choice of any one of these as the middle element will serve the purpose.
- The time complexity of ~~the~~ Binary search is $O(\log n)$.

C - procedure :

```
int Binarysearch ( int a[], int n, int key )
{
    int low, high, mid ;
    low = 0 ;
    high = n - 1 ;
    while ( low <= high )
    {
        mid = ( low + high ) / 2 ;
        if ( key == a[mid] )
            return mid ;
        if ( key < a[mid] )
            high = mid - 1 ;
        else
            low = mid + 1 ;
    }
    return (-1) ;
}
```

Example : search the key value 33 in the following set of elements using binary search method.

11, 22, 33, 44, 55, 66, 77

11	22	33	44	55	66	77
0	1	2	3	4	5	6
↑ low			↑ mid		↑ high	

search element = 33
key = 33 low = 0 high = 6 n = 7

is $low \leq high$ (yes)

$$mid = \frac{0+6}{2} = 3$$

is $33 = a[3]$ (no)

Since $33 < a[3]$, the steps will be repeated for the lower half i.e.

$$\text{low} = 0 \quad \text{high} = \text{mid}-1 = 2$$

Is $\text{low} \leq \text{high}$ (yes)

$$\text{mid} = \frac{0+2}{2} = 1$$

11	22	33	44	55	66	77
0	1	2	3	4	5	6
↑ low	↑ mid	↑ high				

Is $33 == a[1]$ (no)

since $33 > a[1]$, repeat the step with $\text{low} = \text{mid} + 1 = 2$ and
 $\text{high} = 2$.

Is $\text{low} \leq \text{high}$ (yes) $\text{mid} = \frac{2+2}{2} = 2$

11	22	33	44	55	66	77
0	1	2	3	4	5	6
↑ low		↑ mid = low = high				↑ high

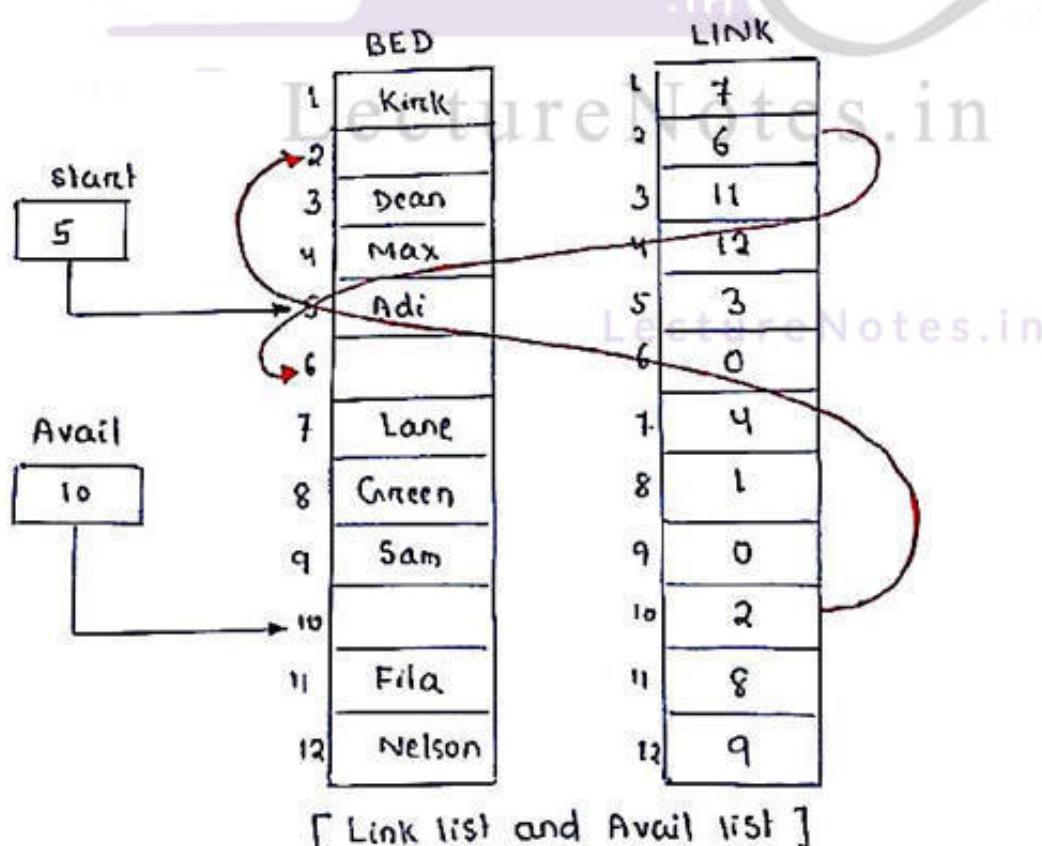
Is $33 == a[2]$ (yes)

This search is successful at index 2.

GARBAGE COLLECTION :

- Garbage collection is the process of adding all the deallocated memory to the memory back (free memory spaces).
- The maintenance of link lists in memory assumes the possibility of inserting new nodes into the list and hence requires some mechanism which provides unused memory space for the new node. Analogously, some mechanism is required whereby the memory space of deleted nodes becomes available for future use.
- Together, with the link list in memory, a special list is maintained which consists of unused memory cells. This list which has its own pointer is called the list of the available space or the free storage list or the free pool.

Example : Let if link lists are implemented by parallel arrays the if operation on link list is done by using the memory spaces linked together by the external pointer start, then the unused memory cells in the array will also be linked together to form a link list using Avail or its list pointer variable.



COMPACTIION :

- Compaction is a method of stuffing the memory contents so as to place all free memory together in one large block.
- process of collecting all the free memory and putting it one side of memory and at the same time collection of all used memory and putting it another part of memory is called compaction
- Compaction is done by the operating system by means of compaction algorithm. The simplest compaction algorithm is to move all processed nodes towards one end of memory ; all holes moves in another direction producing one large available memory .

	✓	✓	
	✓		✓
		✓	
✓			✓

In above figure every block has 4 bytes of size .

✓ = used space .

- Suppose we have declared int a[5]. For this 10 bytes of memory will going to be allocated when compiler will search for continuous 10 bytes of memory , in memory pool , it will not find but free spaces are available in scattered manner . At this time Compaction process is need to avail continuous memory location .

✓	✓	*	*
✓	✓	*	*
✓	✓	*	*
✓	*	*	*

* → free memory

- compaction is a dynamic memory management scheme .



Data Structure Using C

Topic:
Heap Sort

Contributed By:
Mamata Garanayak

Lesson Number : 43

Heapsort :

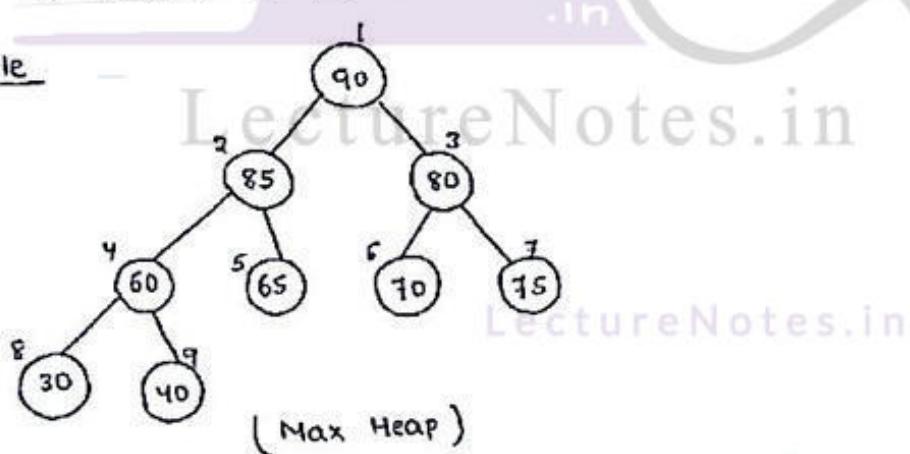
Introduction: A binary tree is called a heap, if keys are assigned to its node (one key per node) and must satisfy the following two properties:

1. Tree's structure: The binary tree is essentially complete that means the tree is completely filled on all levels with a possible exception where the lowest level is filled from left to right, and some right most leaves leaves may be missing.
2. Heap Order: For every node 'n' in a binary tree that follows the tree's structure; other than the root node, the key stored in 'n' is greater/ smaller than or equal to the key stored in the parent of node 'n'

Max Heap:

A heap is known as a Max heap, if for every node 'n' (other than root), the relation between node 'n' and its parent is the "smaller relation" ($n \leq \text{parent}(n)$). In this maximum value is stored in the root

Example



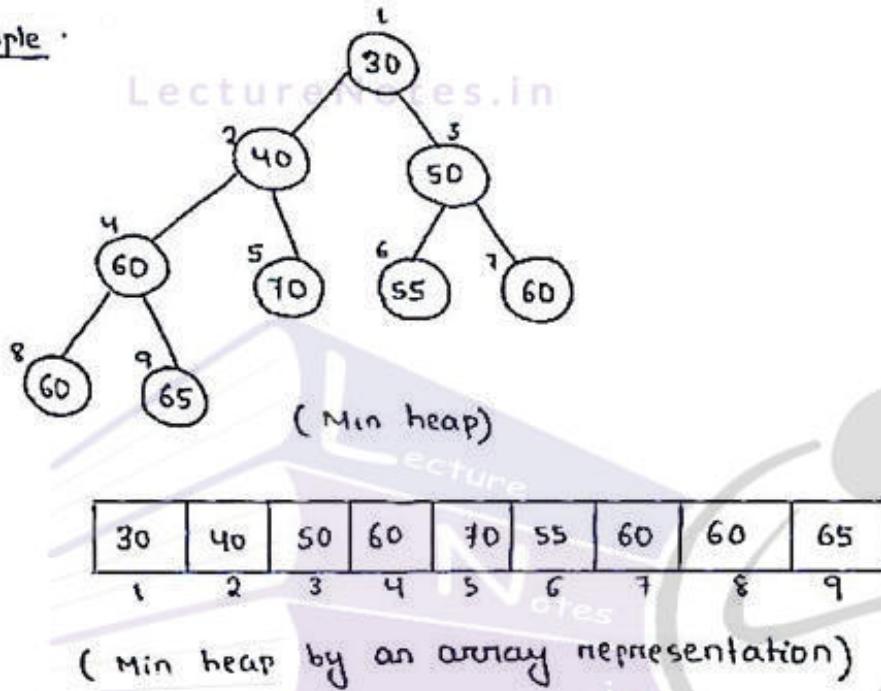
90	85	80	60	65	70	75	30	40
1	2	3	4	5	6	7	8	9

(Max heap by an array representation)

Min Heap:

A heap is known as Min heap, if for every node 'n' (other than the root), the relation between node 'n' and its parent is the "greater relation" ($n \geq \text{parent}(n)$). In this, the minimum value is stored in the root.

Example :



Building a Heap :

Algorithm:

Build-Max-heap (A)

1. heap-size [A] \leftarrow length [A]
2. for $i + \text{length}[A]/2$ down to 1
3. do Max-Heapify (A, i)

Max-Heapify (A, i): Max-Heapify (A, i) is an important subroutine for manipulating max-heaps.

Algorithm:

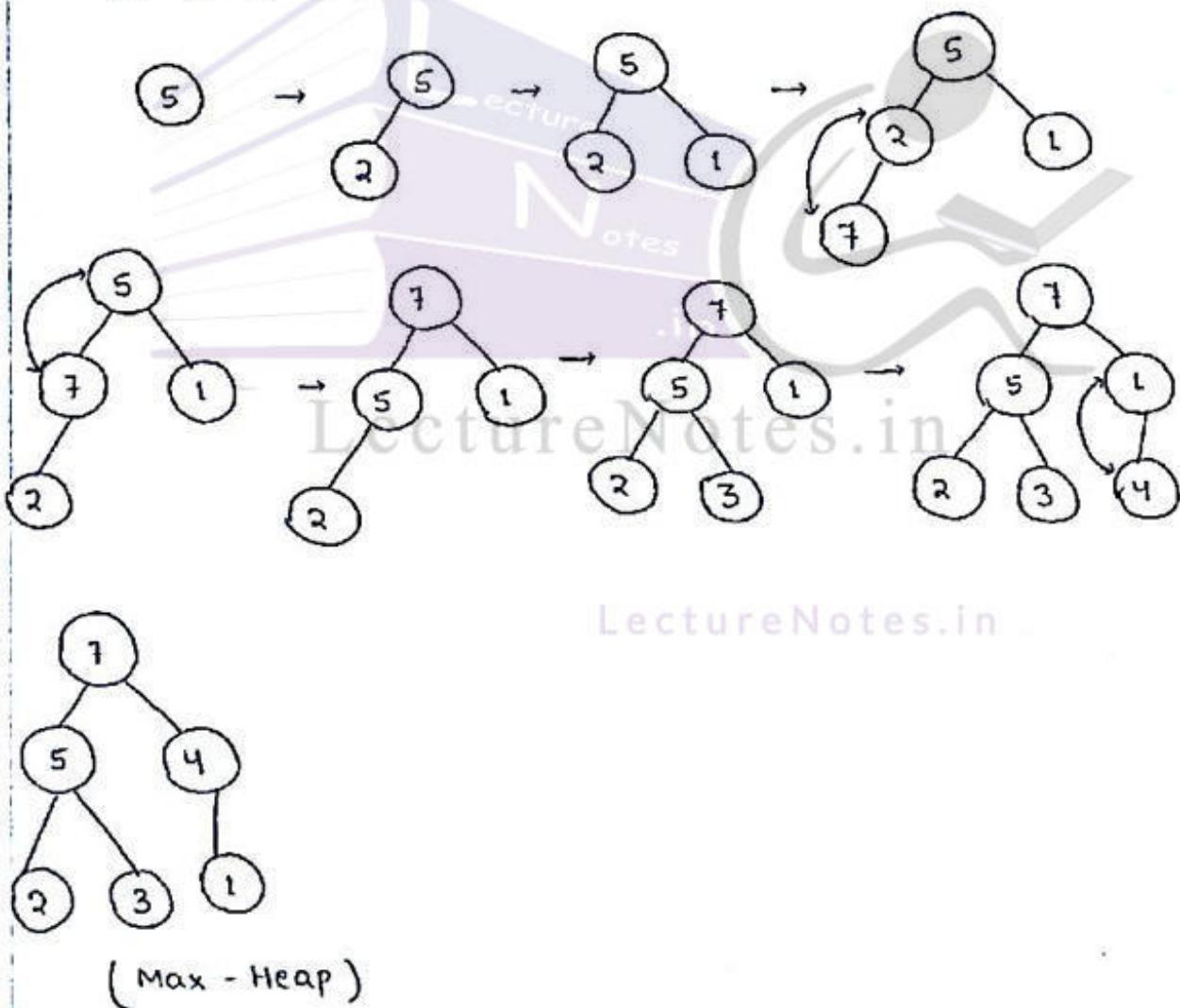
Max-Heapify (A, i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$

3. if $l \leq \text{heap_size}[n]$ and $A[l] > A[i]$
4. then $\text{largest} \leftarrow l$
5. else $\text{largest} \leftarrow i$
6. if $n \leq \text{heap_size}[n]$ and $A[n] > A[\text{largest}]$
7. then $\text{largest} \leftarrow n$
8. if $\text{largest} \neq i$
9. then ~~LectureNotes.in~~ exchange $A[i] \leftrightarrow A[\text{largest}]$
10. Max-Heapify ($A, \text{largest}$).

Q. Build a max heap using the following elements.

5, 2, 1, 7, 3, 4



Heapsort Algorithm :

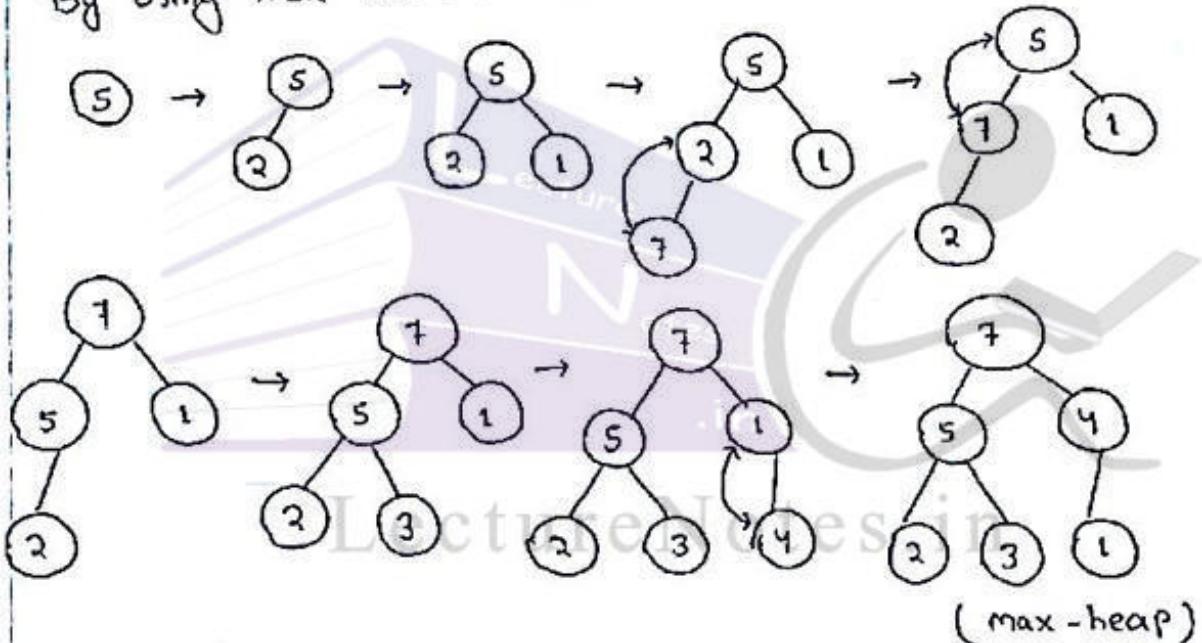
Heapsort(A)

1. Build - Max - heap(A)
2. for $i \leftarrow \text{length}[A]$ down to 2
3. do exchange $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. Max-Heapify(A, 1)

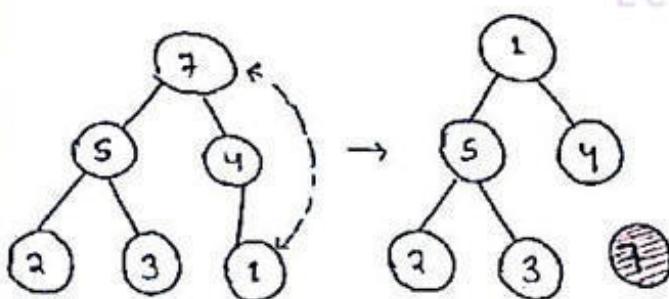
Q. Sort the following elements by using heapsort.

5, 2, 1, 7, 3, 4

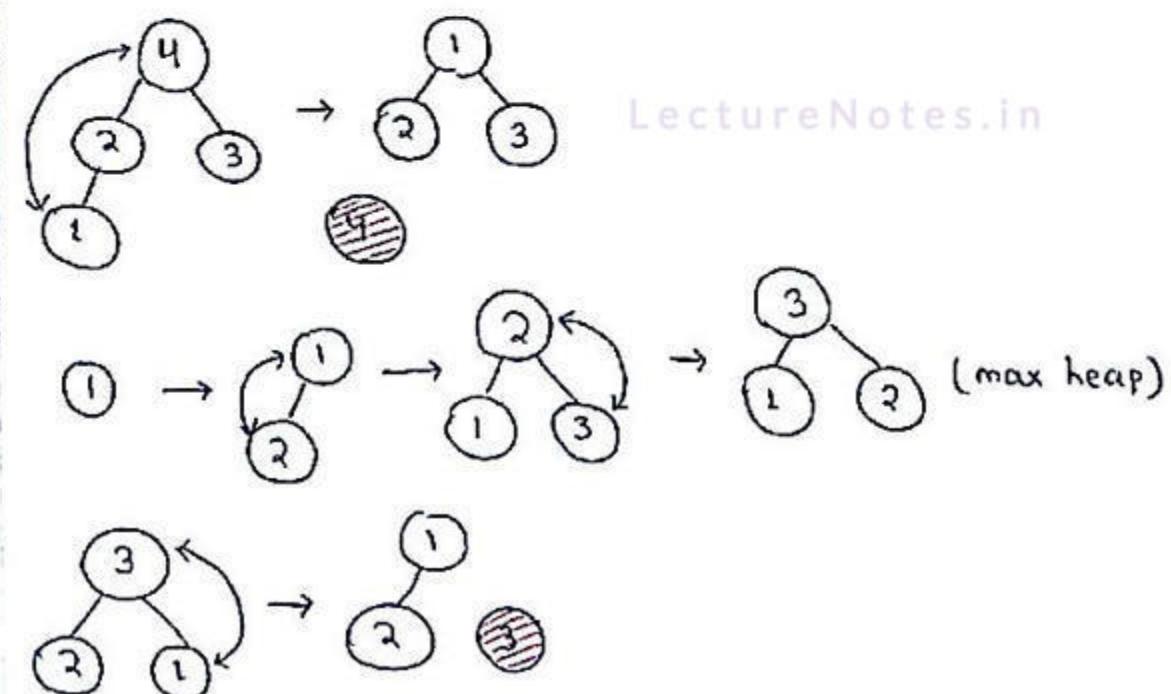
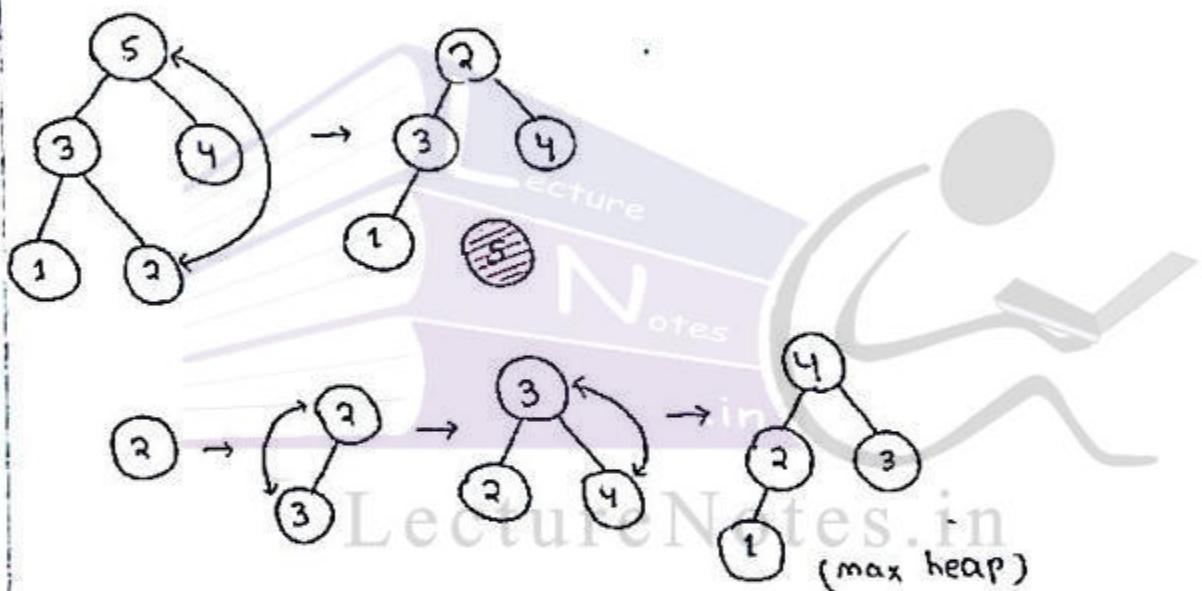
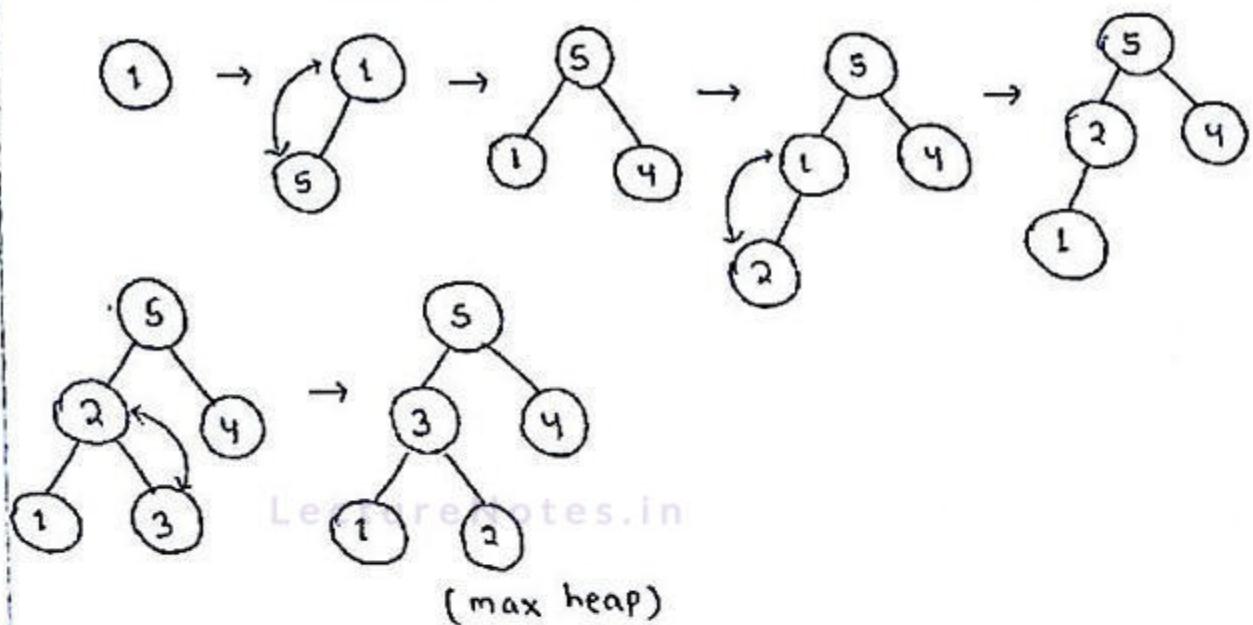
By using these elements we have to first make a max heap.

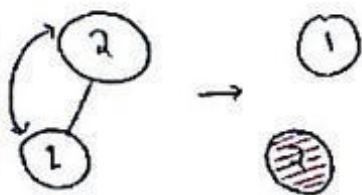
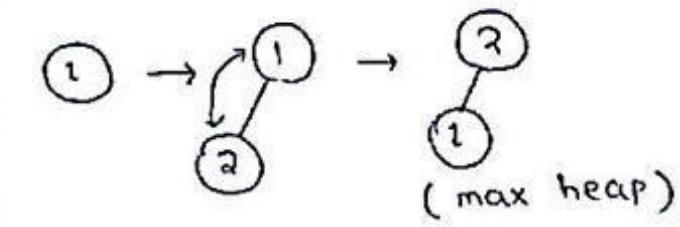


Heapsort :



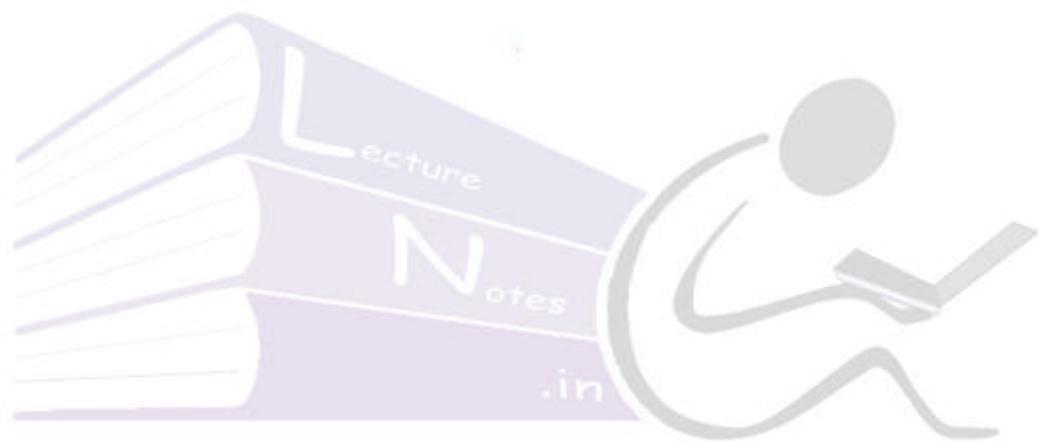
Again by using the elements 1, 5, 4, 2, 3 we have to make a max heap.





LectureNotes.in

The elements after the heapsort are 7, 5, 4, 3, 2, 1 //



LectureNotes.in

LectureNotes.in



Data Structure Using C

Topic:
Hashing

Contributed By:
Mamata Garanayak

HASHING :

Hashing is a method of searching in which searching is independent of number of elements stored in the data structure.

Hash Table :

The data structure used to store data in hashing is called hash table. A hash table is simply an array that is addressed via a hash function. The hash table is divided into a number of buckets and each bucket is capable of storing a number of records. Thus we can say that a bucket has number of slots and each slot is capable of holding one record.

location	Slot 1
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

(A hash table)

Hash Function :

This is a mapping used to find the key value by means of which elements are stored in the hash table. The basic idea in hashing is the transformation of a key into the corresponding location in the hash table. This is done by a hash function.

A hash function can be defined as a function that takes key as input and transforms it into a hash table index.

There are basically two types of hash function.

1. Distribution - independent function
2. Distribution - dependent function

→ The distribution - independent function do not utilize the distribution of the keys of a hash table in order to compute the location of the desired record.

→ The distribution - dependent functions are obtained by examining the subset of key values corresponding to the desired records.

The most popular distribution independent hash functions are :-

1. Division Method
2. mid-Square Method
3. Folding Method
4. Digit analysis Method
5. Length dependent Method.

The main idea behind any hash function is to find a one-to-one correspondence between a key value and index in the hash table where the key value can be placed. The criteria in deciding a hash function $H : K \rightarrow I$ are as follows.

- (i) The function H should be very easy and quick to compute.
- (ii) The function H should achieve an even distribution of keys that actually occur across the range of indices.

1. Division Method :

- A simple choice for hash function is obtained by using modulo operator.
- Convert the key to an integer then divide it by the size of the index range and take the remainder as the result. Choose a number m larger than the number n of keys in K . (The number m is usually chosen to be a prime no. or a no. without small divisors, since this frequently minimizes the no. of collision).

The Hash function H is then defined by :-

$$H(K) = K \cdot i \quad \text{if indices start from 0}$$

$$H(K) = (K \cdot i + 1) \quad \text{if indices start from 1}$$

Example: let us choose hash table size, tablesize is equal to 97 and key value is 2163, then

$$H(2163) = 2163 \% 97$$

= 48 if indices start from 0 and it is

= 49 if indices start from 1.

2. Mid - Square Method :

The hash function H is defined by $H(K) = x$, where x is obtained by selecting an appropriate no. of bits or digits from the middle of the square of the key value K . This selection usually depends on the size of the hash table. It should be emphasized that the same criteria should be used for selecting the bits or digits for all of the keys.

Example: suppose key values are of integer type, and we require 3 digits address. Our selection criteria is to select 3 digits at even positions starting from the right-most digit in the square. So, the address calculations, for 3 distinct keys and with the hash function is :

$$K : 1234 \quad 2345 \quad 3456$$

$$K^2 : 1522756 \quad 5499025 \quad 11943936$$

$$H(K) : 925 \quad 492 \quad 933$$

Hence, the second, fourth and sixth digits, counting from the right are chosen for the hash addresses.

The mid-square method has been criticized because of time consuming computation but it usually gives good results so far uniform distribution of the keys over the hash table is concerned.

3. Folding Method :

This method can be defined as follows :

partition the key K into a number of parts k_1, k_2, \dots, k_n , where each part, except possibly the last, has the same no. of bits or digits as the required address width. Then the parts are added together, ignoring the last carry, if any. Alternatively,

$$H(K) = k_1 + k_2 + \dots + k_n, \text{ where the last carry, if any, is ignored.}$$

LectureNotes.in

If the keys are in binary form, the exclusive-OR operation may be substituted for addition.

There are many variations known to this method. One is called fold shifting method, where the even no. parts k_2, k_4, \dots are each reversed before the addition.

Another variation is called the fold boundary method. Here two boundary parts namely k_1 and k_n each are reversed and then they are added with all other parts.

Example : Let us take the size of each part be 2.

$$K : \begin{array}{r} 152276 \\ 5499025 \\ \hline 11943936 \end{array}$$

$$\text{Chopping: } \begin{array}{rrr} 01 & 52 & 21 & 56 \\ 05 & 49 & 90 & 25 \\ \hline 11 & 94 & 39 & 36 \end{array}$$

$$\text{pure folding: } \begin{array}{rrr} 01+52+21+56 & 05+49+90+25 & 11+94+39+36 \\ = 136 & = 169 & = 180 \end{array}$$

$$\text{Fold shifting: } \begin{array}{rrr} 01+25+21+65 & 05+94+90+52 & 11+49+39+63 \\ = 118 & = 241 & = 162 \end{array}$$

$$\text{Fold boundary: } \begin{array}{rrr} 10+52+27+65 & 50+49+90+52 & 11+94+39+63 \\ = 154 & = 241 & = 207 \end{array}$$

Folding is a hashing function which is also useful in converting multi-word keys into a single word so that other hashing function can be used on that. In fact the term hashing comes from chopping a key into pieces.

4. Digit Analysis Method :

The basic idea of this hashing function is to form hash addresses by extracting and/or shifting the extracted digits or bits of the original key.

Example: Given a key value = 6732541

It can be transformed to the hash address 421 by extracting the digits in even positions and then reversing it.

For a given set of keys, the position in the keys and same rearrangement pattern must be used consistently.

This method is particularly useful in the case of static files where the key values of all the records are known in advance.

Collision Resolution Techniques :

Let us consider a hash table of size 10 whose indices are 0, 1, 2, ..., 8, 9. Suppose a set of key values are 10, 19, 35, 43, 62, 59, 31, 49, 77, 33.

Let assume the hash function H as :

- Add the two digits in the key.
- Take the digit at unit place of the result at index ; ignore the digit at decimal place, if any.

Using this hash function, the mapping from key values to indices and hash table are as follows.

K	I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

$$H: K \rightarrow I$$

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash table

For the given set of key values, the hash function does not distribute them uniformly over the hash table; some entries are there which are empty, and in some entries more than one key values are to be stored. Allotment of more than one key values in one location in the hash table is called collision. We have found 3 collisions for 59, 31 and 77.

Collision in hashing cannot be ignored, whatever the size of the hash tables. There are several techniques to resolve collisions. Two important methods are:

- i) Closed Hashing (also called linear probing)
- ii) Open Hashing (also called chaining)

i) Closed Hashing:

- The simplest method to resolve a collision is closed hashing.
- Suppose there is a hash table of size n and the given key value is mapped to an address location i , with a hash function.
- The closed hashing can be stated as follows:

Start with the hash address where the collision has occurred, let it be i . Then follow the sequence of locations in the hash table and do the sequential search.

$$i, i+1, i+2, \dots, n, 1, 2, \dots, i-1$$

- The search will continue until any one of the following cases occurs
 - 1. The key value is found.
 - 2. Unoccupied or empty location is encountered.
 - 3. It reaches to the location where the search was started.
- The first case corresponds to the successful search and the last two cases correspond to unsuccessful search.
- Here the hash table is considered as circular, so that when the last location is reached, the search proceeds to the first location of the table. This is why, the technique is called as closed hashing. Since the technique searches in straight line,

It is also alternatively termed as linear probing, probe means Key Comparison.

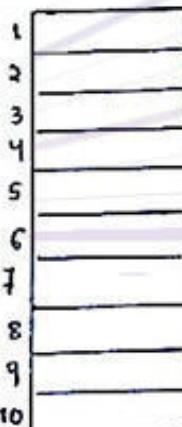
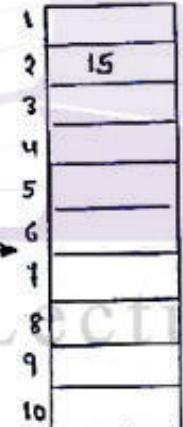
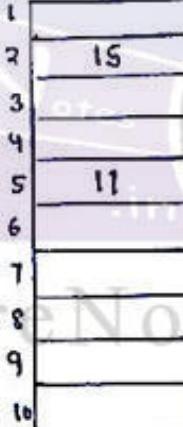
Example:

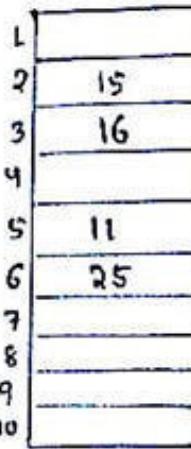
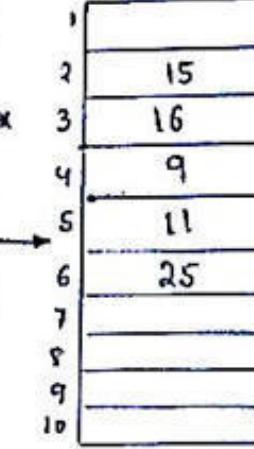
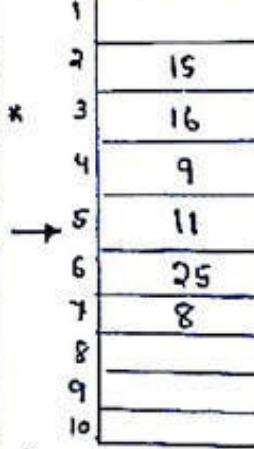
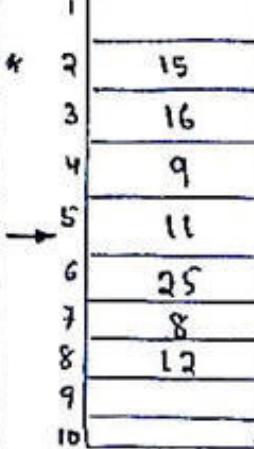
Assume that there is a hash table of size 10 and hash function uses the division method with remainder modulo 7, namely, $H(K) = (K \% 7) + 1$.

Let us consider the build up of the hash table (initially, which is empty) with the following set of key values.

15 11 25 16 9 8 12

The loading of hash table will take place successively by performing a search for key and insert it into the table in an empty room if the key is not in the table and leave if it is overflow, that is, no free room to accommodate any further key value.

			
Initially the hash table is empty	insertion of 15	insertion of 11	insertion of 25

			
insertion of 16	insertion of 9	insertion of 8	insertion of 12



Data Structure Using C

Topic:

Drawbacks Of Closed Hashing And Its Remedies

Contributed By:

Mamata Garanayak

Drawbacks of Closed Hashing and its remedies :

The major drawbacks of closed hashing is that, as half of the hash table is filled, there is a tendency towards clustering, that is key values are clustered in large groups and as a result sequential search becomes slower and slower. This kind of clustering typically known as primary clustering.

LectureNotes.in
Some solutions to avoid this situations are follows :

- Random probing
- Double hashing or rehashing
- Quadratic probing.

a) Random probing :

- This method uses a pseudo random number generator to generate a random sequence of locations, rather than an ordered sequence as was the case in linear probing method.
- The random sequence generated by the pseudo random generator contains all positions between 1 and h, the highest location of the hash table.
- An example of a pseudo random number generator that produces such a random sequence of location is :

$$i = (i + m) \% h + 1$$

where i = number in the sequence
 $m \neq h$ = are integers that are relatively prime to each other.

For example suppose $m=5$ and $h=11$ and initially $i=2$; then the above mentioned pseudo random number generator generates the sequence as ;

$$8, 3, 9, 4, 10, 5, 11, 6, 1, 7, 2$$

We stop producing the numbers when the first location is duplicated. Here all the numbers betn 1 and 11 are generated but randomly .

b) Double Hashing:

- Random hashing however is not free from clustering. Another type of clustering, called secondary clustering is involved here.
- In particular, clustering occurs when two keys are hashed into the same location.
- In such an instance, if the same sequence of locations is generated for two different keys by the random probing method then a clustering takes place.
- An alternative approach to avoid the secondary clustering problem is to use a second hash function in addition to the first one.
- This second hash function results the value of m for the pseudo random no. generator as employed in random probing method.
- This second function should be selected in such a way that hash addresses generated by two hash functions are distinct and the second function generates a value m for the key K so that m and h are relatively prime.

Example:

Suppose $H_1(K)$ is initially used hash function and $H_2(K)$ is its second one. These two functions are defined as;

$$\begin{aligned}H_1(K) &= (K \% h) + 1 \\H_2(K) &= (K \% (h \cdot 4)) + 1\end{aligned}$$

Let $h = 11, K = 50$

then $H_1(50) = 7$ and $H_2(50) = 2$.

Therefore $H_1(50) \neq H_2(50)$, that is H_1 and H_2 are independent.

and $m = 2$ & $h = 11$ are relatively prime.

Hence using $i = [(i+2) \% 11] + 1$ and initially $i = 7$ we have the random sequence as

10, 2, 5, 9, 11, 3, 6, 9, 1, 4, 7

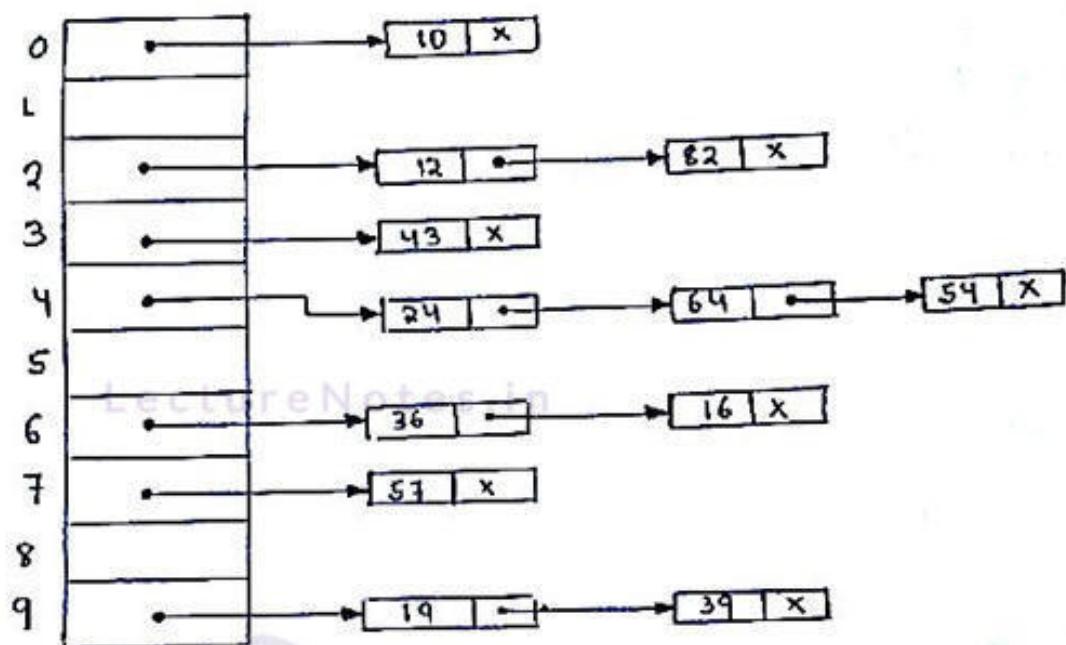
c) Quadratic probing :-

- Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing.
- For linear probing, if there is a collision at location i then next $i+1, i+2, i+3, \dots$, locations are to be probed, but in quadratic, the next location to be probed are $i+1^2, i+2^2, i+3^2, \dots$ etc.
- Mathematically, if h be the size of the hash table and $H(x)$ is the hash function then the quadratic probing searches the locations ;
$$H(x) + i^2 \mod h \text{ for } i=1, 2, 3, \dots$$
- In quadratic probing the increment function is i^2 . It is also assumes the hash table as close on circular like linear probing.
- This method substantially reduces primary clustering, but it does not probe all locations in the table.

ii) Open Hashing :-

- The closed hashing method deals with arrays as hash tables and thus we are able to refer quickly to random positions in the tables.
- But there are two main difficulties with this technique : first it is very difficult to handle the situation of table overflow in a satisfactory manner. second, key values are empl keys haphazardly intermixed and, on the average, majority of the keys are far from their hash locations and thus increasing the number of probes which degrades the overall performance.
- To resolve these problems another hashing method called open hashing also called chaining is known.
- The chaining method uses a hash table as an array of pointers ; each pointer points to a linked list. That is,

here, hash table is an array of list headers.



(An open Hashing)

- In the above figure, a hash table of size 10 is considered.
→ The index of the hash table varies from 0 to 9 and the key values are taken as integers.
→ Hash address for a key is decided by its last digit i.e. right most digit.
→ For a given key value, hash address is calculated. It then searches the linked list pointed by the pointers at that locations. If the element is found it returns the pointer to node containing that key value else insert the element at the end of that list.

Assignment: List out the drawbacks of Quadratic probing ?

Drawbacks of Quadratic probing:

- In quadratic probing there is no guarantee of finding an empty cell once more than half of the table gets full or even before that if the table size is not prime.
- In quadratic probing it is also very crucial that the table size should be a prime. If the table size is not prime, then the no. of alternate locations can be severely reduced.
As an example, if the table size is 16, then the only alternate locations would be at distances 1, 4, 9, etc.

Assignment: List out the advantages & disadvantages of chaining.

There are several advantages of chaining method.

1. Overflow situation never arises. Hash table maintains lists which can contain any no. of key values.
2. Collision resolution can be achieved very efficiently if the lists maintain an ordering of keys, so that keys can be searched quickly.
3. Insertion and deletion become quick and easy task in open hashing. Deletion proceeds in exactly the same way as deletion of a node in single link list.
4. Open hashing is best suitable in applications where no. of key values varies drastically as it uses dynamic storage management policy.

The Chaining method has only disadvantage of maintaining linked lists and extra storage space for link fields.