



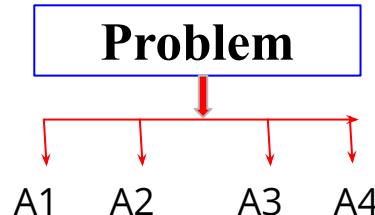
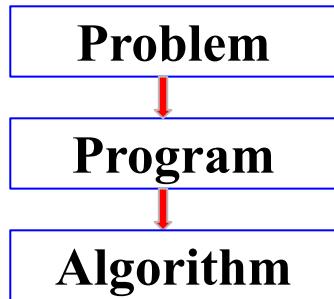
Data Structure and Algorithms

Dr. Sunil Saumya, IIIT Dharwad



Program

- In order to solve any problem in computer science we generally write the **program**.
 - We write an informal description of a solution prior to a program.
 - That informal description of a solution is also known as **Algorithm**.



- Before writing a program of a problem we find the best algorithm.

Algorithm Vs Program

- What is an **algorithm**?
 - It is a step by step process for solving a computational problem.
- What is a **program**?
 - It is also a step by step process for solving a computational problem.
- Then what is the **difference** between **Algorithm** and **Program**?

Algorithm and Program difference

Algorithm	Program
Written at design time.	Written at implementation time.
Person with domain knowledge write the Algorithm.	Programmer writes the program.
High level language with some notations.	Written only using programming language
Hardware and OS independent	Hardware and OS dependent
Analysis	Testing

Priori Analysis Vs Posteriori Testing

Algorithm	Program
Priori Analysis	Posteriori Testing
Independent of language	Language dependent
Hardware independent	Hardware dependent
Time and Space function	Watch time and bytes

Characteristics of an Algorithm

- **Input:** zero or more
- **Output:** at least one output
- **Definiteness:** only known steps are written
- **Finiteness:** algorithm must stop at some point
- **Effectiveness:** not writing any unnecessary steps

How to write an Algorithm?

- Algorithm can be written in:
 - A simple high level language.
 - High level language with some notation involved
 - Mix of high level language and some programming keywords, also known as pseudo code.

Algorithm swap (a,b)

Begin

temp ← a

a ← b

b ← temp

End

How to analyze an Algorithm?

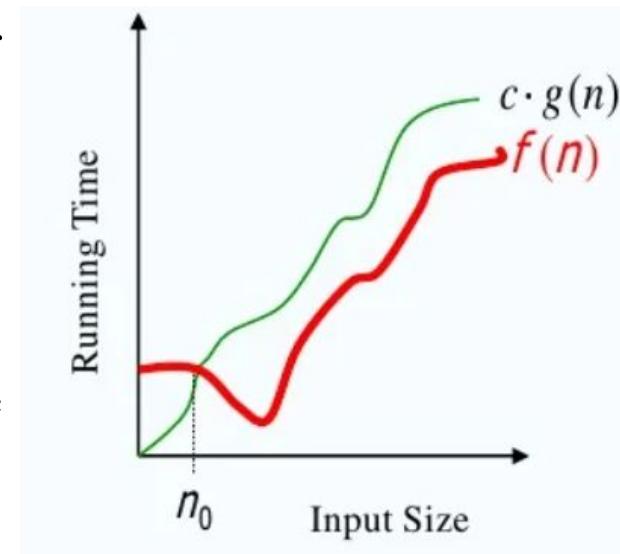
- Two major criteria:
 - Time
 - Space
- Other important criteria:
 - Network
 - Power
 - CPU registers

Asymptotic notation

- The asymptotic analysis clearly explains about how the running time of an algorithm increases with increase in input size within the limit.
- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- Three asymptotic notations are:
 - Big-oh
 - Big-omega
 - Thetha

“Big-oh” O-notation

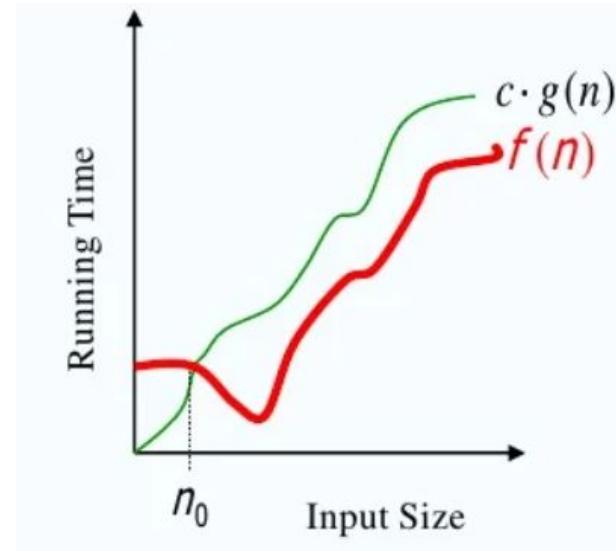
- Asymptotic upper bound. It indicates the upper or highest growth rate that the algorithm can have.
- $f(n) = O(g(n))$, if there exists constants c and n_0 such that $f(n) \leq c g(n)$ for $n \geq n_0$
- $f(n)$ and $g(n)$ are functions over non-negative integers and they are also non-decreasing.
- Used for worst-case analysis.



“Big-oh” O-notation

Say $f(n) = 3n+2$ and $g(n) = n$

Can we say the $f(n) = O(n)$? If yes then how?



“Big-oh” O-notation

Say $f(n) = 3n+2$ and $g(n) = n$

Can we say the $f(n) = O(n)$? If yes then how?

If this has to be followed then

$$f(n) \leq cg(n), \text{ where } c > 1 \text{ and } n > n_0 \text{ for } n_0 \geq 1$$

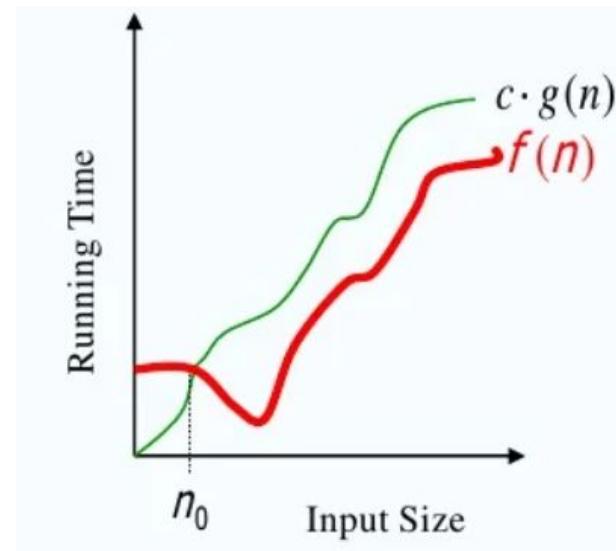
Therefore,

$$3n+2 \leq cn \quad (\text{substitute values of } c)$$

We find that when $c=4$ then above equation is true.

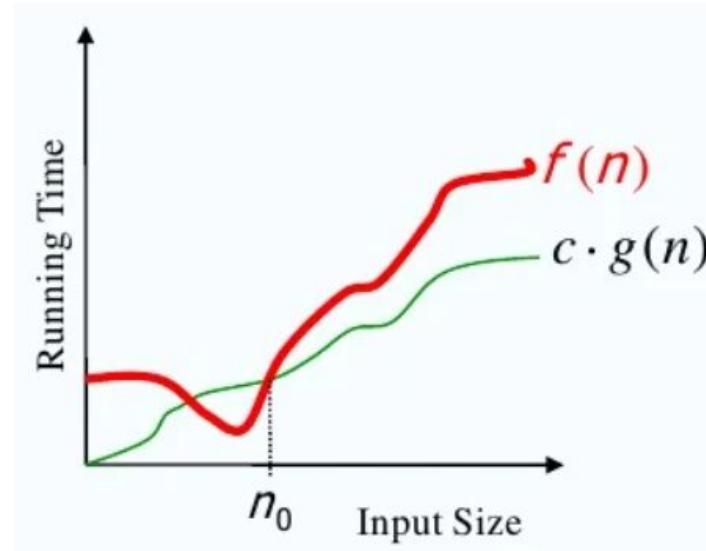
$$3n+2 \leq 4n, \text{ where } c=4$$

We also get $n \geq 2$ Therefore, $f(n) = O(n)$



“Big-omega” Ω -notation

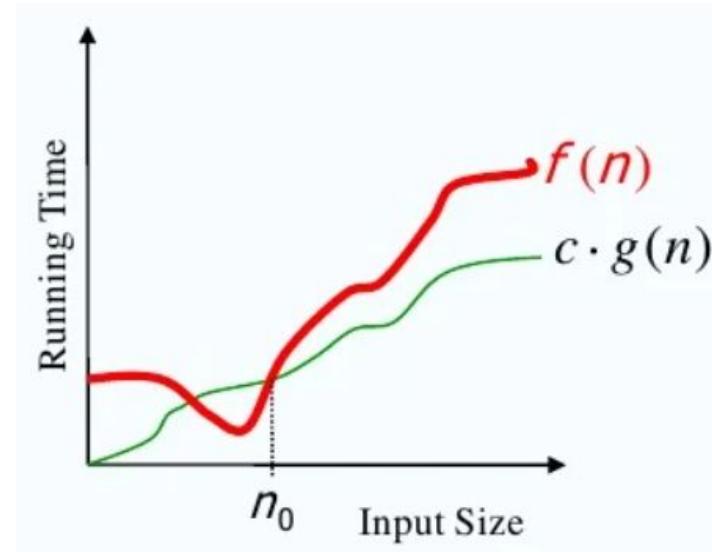
- The “big-Omega” notation provides a lower bound.
- The function $f(n)=\Omega(g(n))$ if there exists constants $f(n)\geq cg(n)$ for $n\geq n_0$.
- Used to describe the best case running times or lower bonds of algorithmic problems.
- Example: lower bond of searching in an unsorted array is $\Omega(n)$.



“Big-omega” Ω -notation

Say $f(n) = 3n+2$ and $g(n) = n$

Can we say the $f(n) = \Omega(n)$? If yes then how?



“Big-omega” Ω -notation

Say $f(n) = 3n+2$ and $g(n) = n$

Can we say the $f(n) = \Omega(n)$? If yes then how?

If this has to be followed then

$$f(n) \geq cg(n), \text{ where } c > 1 \text{ and } n > n_0 \text{ for } n_0 \geq 1$$

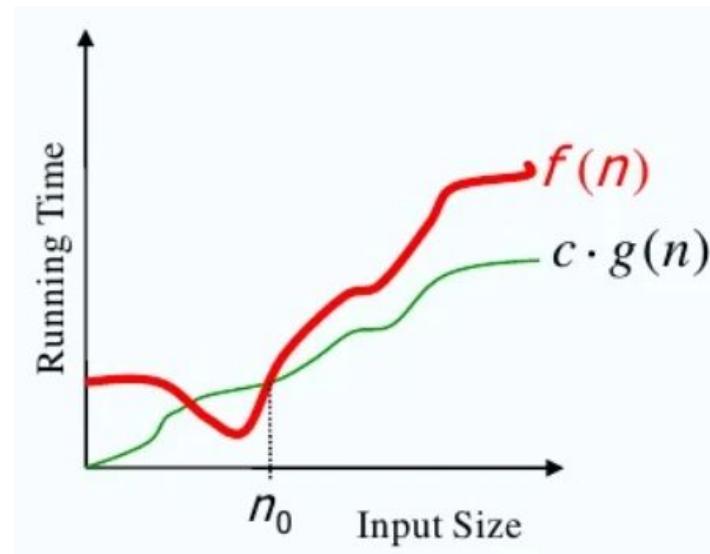
Therefore,

$$3n+2 \geq cn \quad (\text{substitute values of } c)$$

We find that when $c=1$ then above equation is true.

$$3n+2 \geq n, \text{ where } c=1 \text{ for all values of } n$$

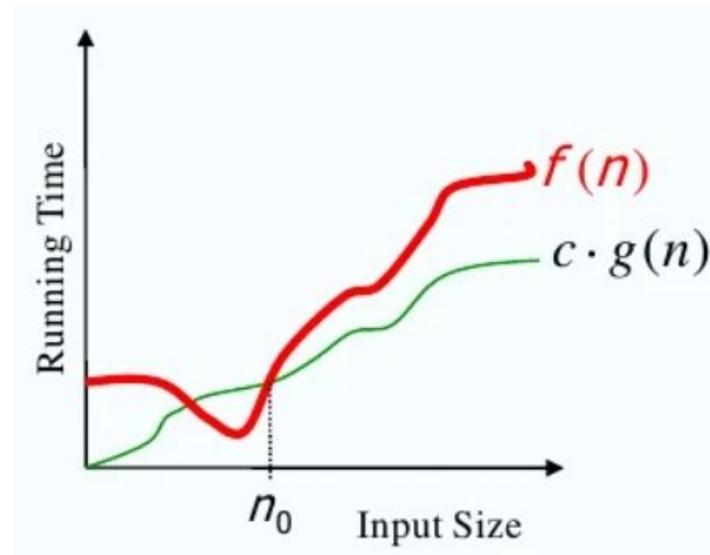
We also get $n_0 = 1$ Therefore, $f(n) = \Omega(n)$



“Big-omega” Ω -notation

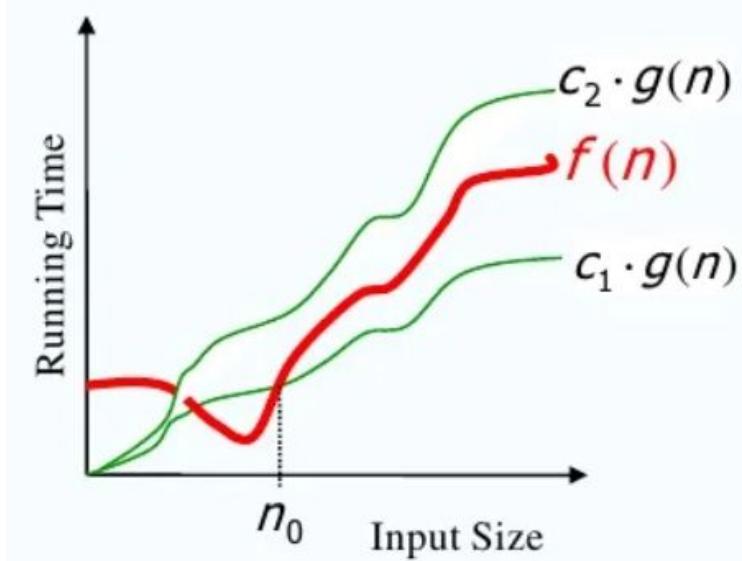
Say $f(n) = 3n+2$ and $g(n) = n$

Can we say the $f(n) = \Omega(n^2)$?



“theta” Θ -notation

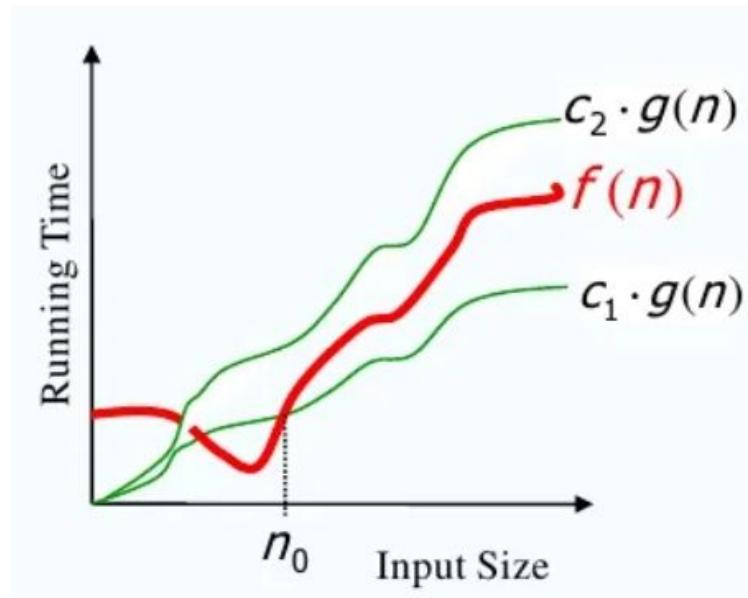
- It is asymptotic tight/average bound.
- The function $f(n)=\Theta(g(n))$ if there exists constants C_1 and C_2 such that $f(n)$ is sandwiched between $C_1 \cdot g(n)$ and $C_2 \cdot g(n)$.
$$C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n), \text{ for } n \geq n_0$$
- Another way of thinking is:
 $f(n)=\Theta(g(n))$ if and only if
 $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$



“Big-theta” Θ -notation

Say $f(n) = 3n+2$ and $g(n) = n$

Can we say the $f(n) = \Theta(n)$?



“Big-theta” Θ -notation

Say $f(n) = 3n+2$ and $g(n) = n$

Can we say the $f(n) = \Theta(n)$?

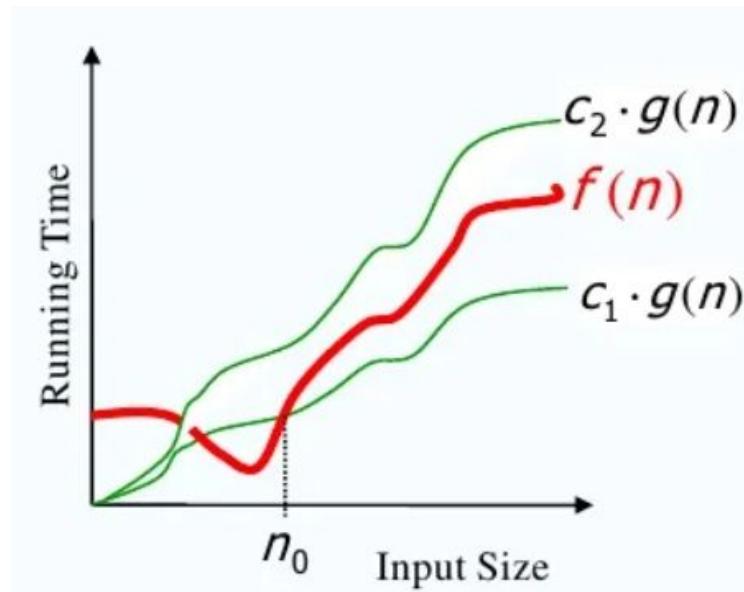
If this has to be followed then

$$f(n) \leq c_2 g(n) \text{ and } f(n) \geq c_1 g(n)$$

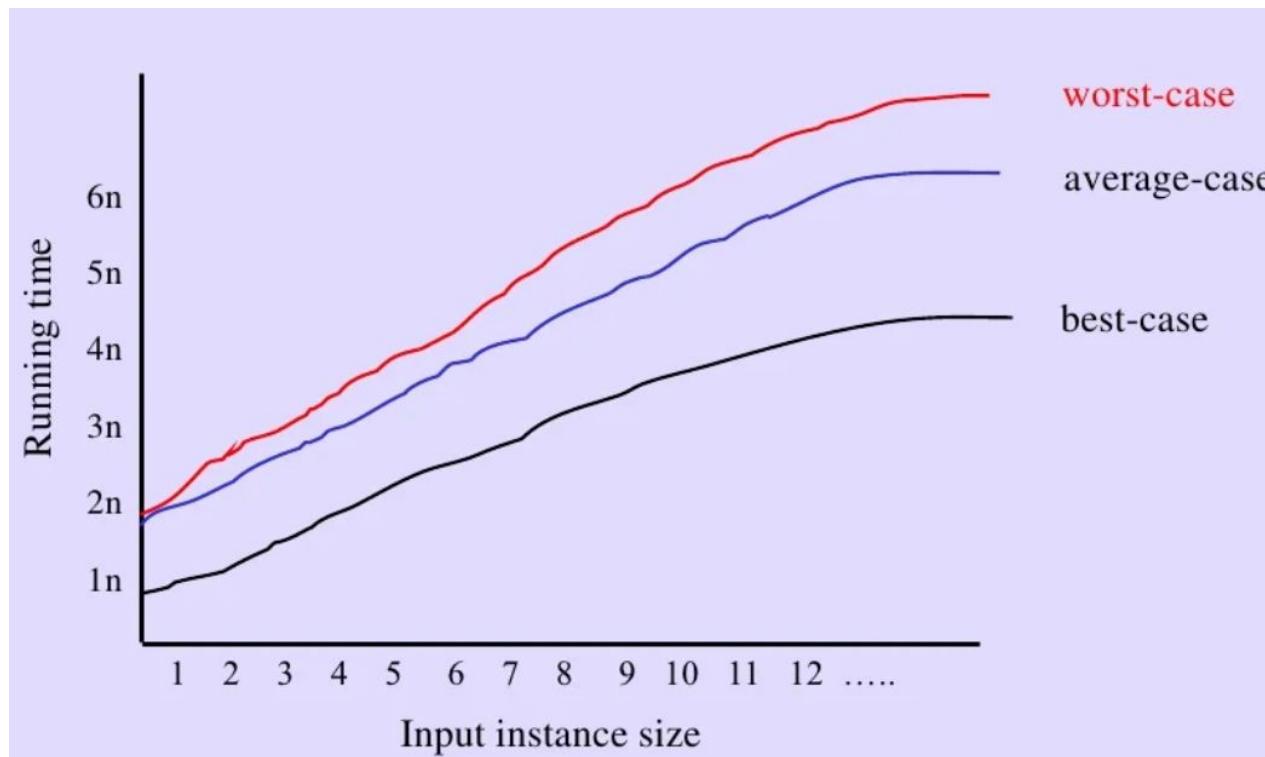
$$3n+2 \leq 4n \text{ and } 3n+2 \geq n \text{ for all } n=1$$

Where $c_2 = 4$ and $c_1 = 1$

Therefore, we can say $f(n) = \Theta(n)$



Best, Worst and Average case complexity



How to analyze an algorithm?

- Every simple statements takes a single unit of time.

temp ← a 1 unit

a ← b 1 unit

b ← temp 1 unit

Total time $f(n)$ = 3 unit

Here $f(n)$ is a time function

Similarly,
In the algorithm three
parameters are used

a 1 unit

b 1 unit

temp 1 unit

Begin

temp ← a

a ← b

b ← temp

End

Total space $s(n)$ = 3 unit
Here $s(n)$ is space complexity

Algorithm analysis: Frequency count method

- Every simple statements takes 1 unit of time. **Algorithm sum (A,n)**

- But if any statements repeats ‘n’ number of times then we find its frequency and then calculate the time complexity.

- **Time complexity:**

$$f(n) = 1 + n+1 + n+1 = 2n+3$$

Therefore $f(n) = O(n)$

- **Space complexity:**

$$s(n) = A \text{ (n words)} + n \text{ (1 word)} + s \text{ (1 word)} + i \text{ (1 word)}$$

$$s(n) = n+1+1+1 = n+3 \text{ hence, } s(n) = O(n)$$

Begin

$s = 0;$ $\rightarrow 1$

$\text{for}(i=0;i < n;i++)$ $\rightarrow n+1$

$s=s+A[i]$ $\rightarrow n$

$\text{return } s;$ $\rightarrow 1$

End

Given Array A = {4, 3, 5, 6, 7}

Algorithm analysis: Frequency count method

Algorithm sum (A,B, C,n)

{

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

{

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

{

$C[i,j] = A[i, j]+B[i, j]$

}

}

}

Algorithm analysis: Frequency count method

Time complexity:

$$= n+1 + n^2 + n + n^2$$

$$= 2n^2 + 2n + 1$$

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

Space complexity:

$$A \rightarrow n^2$$

$$B \rightarrow n^2$$

$$C \rightarrow n^2$$

$$i \rightarrow 1$$

$$j \rightarrow 1$$

$$n \rightarrow 1$$

$$\text{Space complexity } s(n) = 3n^2 + 3$$
$$s(n) = O(n^2)$$

Algorithm sum (A,B, C,n)

{

 for($i=0;i<n;i++$) $\rightarrow n+1$

{

 for($j=0;j<n;j++$) $\rightarrow n \times (n+1)$

{

$C[i,j] = A[i, j]+B[i, j]$ $\rightarrow n \times n$

}

}

}

Algorithm analysis: Frequency count method

```
Algorithm multiply(A,B, C,n)
{
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            C[i,j] = 0;
            for(k=0, k<n;k++)
                C[i,j] = C[i,j]+A[i, k]*B[k, j]
        }
    }
}
```

Algorithm analysis: Frequency count method

Time complexity:

$$= n+1 + n^2 + n + n^2 + n^3 + n^2 + n^3 \\ = 2n^3 + 3n^2 + 2n + 1$$

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

Space complexity:

$$\begin{array}{ll} A \rightarrow n^2 & \\ B \rightarrow n^2 & \text{Space complexity} \\ C \rightarrow n^2 & s(n) = 3n^2 + 4 \end{array}$$

$$\begin{array}{ll} i \rightarrow 1 & \\ j \rightarrow 1 & s(n) = O(n^2) \\ k \rightarrow 1 & \\ n \rightarrow 1 & \end{array}$$

Algorithm multiply(A,B,C,n)

{

 for($i=0;i<n;i++$) $\rightarrow n+1$

{

 for($j=0;j<n;j++$) $\rightarrow n * (n+1)$

{

$C[i,j] = 0;$ $\rightarrow n * n$

 for($k=0, k<n;k++$) $\rightarrow n * n * (n+1)$

$C[i,j] = C[i,j] + A[i, k]*B[k, j]$ $\rightarrow n * n * n$

}

}

Algorithm analysis: Exercise 1

```
for(i=0;i<n;i++)  
{  
    statements;  
}
```

Algorithm analysis: Exercise

```
for(i=0;i<n;i++) → n+1  
{  
    statements; → n  
}
```

Time complexity:

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

Space complexity:

$$s(n) = 2$$

Algorithm analysis: Exercise 2

```
for(i=0;i<n;i++) → n+1  
{  
    statements; → n  
}
```

Time complexity:
 $f(n) = 2n+1 = O(n)$

Space complexity:
 $s(n) = 2$

```
for(i=n;i>0;i--) → n+1  
{  
    statements; → n  
}
```

Time complexity:
 $f(n) = 2n+1 = O(n)$

Space complexity:
 $s(n) = 2$

```
for(i=1;i<n;i=i+2) →  
{  
    statements; →  
}
```

Time complexity:
 $f(n) = ?$

Space complexity:
 $s(n) = ?$

Analysis of Algorithm

- **Primitive Operation:** Low-level operation independent of programming language. Can be identified in pseudo-code. For eg:
 - Data movement (assign)
 - Control (branch, subroutine call, return)
 - arithmetic and logical operations (e.g. addition, comparison)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

Example: Sorting



Correctness (requirements for the output)

For any given input the algorithm halts with the output:

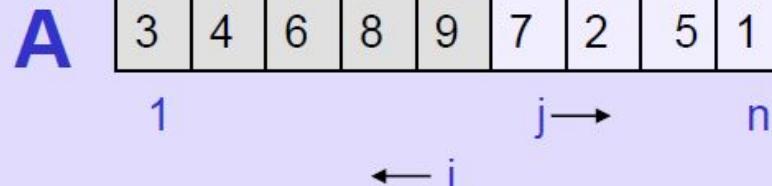
- $b_1 < b_2 < b_3 < \dots < b_n$
- $b_1, b_2, b_3, \dots, b_n$ is a permutation of $a_1, a_2, a_3, \dots, a_n$

Running time

Depends on

- number of elements (n)
- how (partially) sorted they are
- algorithm

Insertion Sorting



Strategy

- Start “empty handed”
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

INPUT: $A[1..n]$ – an array of integers
OUTPUT: a permutation of A such that
 $A[1] \leq A[2] \leq \dots \leq A[n]$

```
for j←2 to n do
    key ← A[j]
    Insert A[j] into the sorted sequence
    A[1..j-1]
    i←j-1
    while i>0 and A[i]>key
        do A[i+1]←A[i]
            i--
    A[i+1]←key
```

Analysis of Insertion Sorting

	cost	times
for $j \leftarrow 2$ to n do	c_1	n
key $\leftarrow A[j]$	c_2	$n-1$
Insert $A[j]$ into the sorted sequence $A[1..j-1]$	0	$n-1$
$i \leftarrow j-1$	c_3	$n-1$
while $i > 0$ and $A[i] > key$	c_4	$\sum_{j=2}^n t_j$
do $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^n (t_j - 1)$
$i--$	c_6	$\sum_{j=2}^n (t_j - 1)$
$A[i+1] \leftarrow \tilde{A}$ key	c_7	$n-1$

$$\begin{aligned}\text{Total time} = & n(c_1 + c_2 + c_3 + c_7) + \sum_{j=2}^n t_j (c_4 + c_5 + c_6) \\ & - (c_2 + c_3 + c_5 + c_6 + c_7)\end{aligned}$$

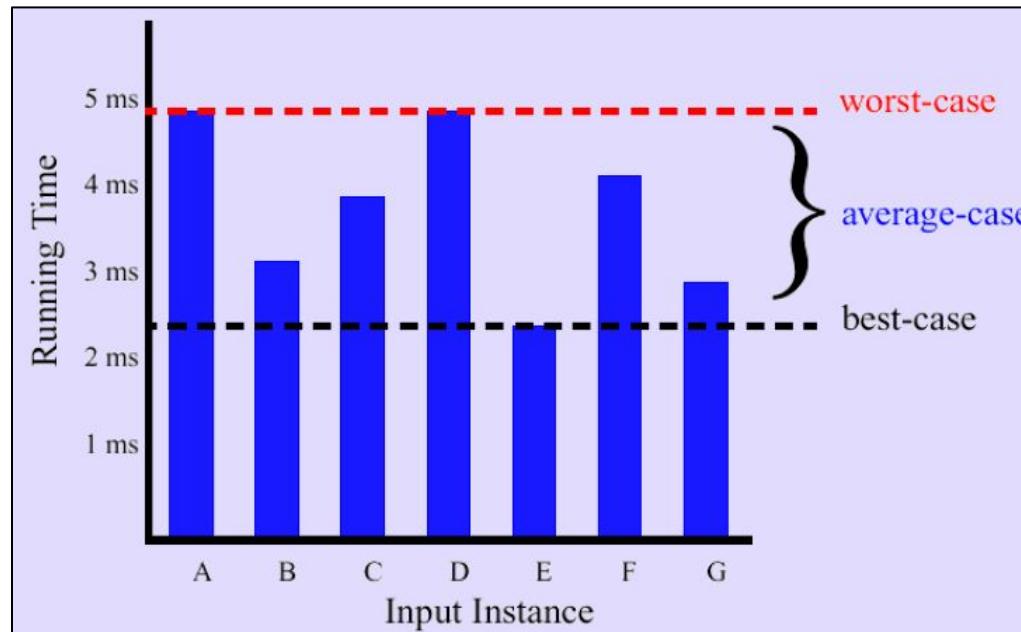
Best/Worst/Average Case

$$\begin{aligned}\text{Total time} &= n(c_1 + c_2 + c_3 + c_7) + \sum_{j=2}^n t_j (c_4 + c_5 + c_6) \\ &\quad - (c_2 + c_3 + c_5 + c_6 + c_7)\end{aligned}$$

- **Best case:** elements already sorted; $A=[1,2,3,4,5]$
 $t_j=1$, running time = $f(n)$, i.e., linear time.
- **Worst case:** elements are sorted in inverse order; $A=[5,4,3,2,1]$
 $t_j=j$, running time = $f(n^2)$, i.e., quadratic time
- **Average case:** elements are roughly half sorted;
 $t_j=j/2$, running time = $f(n^2)$, i.e., quadratic time

Best/Worst/Average Case (2)

- For a specific size of input n , investigate running times for different input instances:



Example of Asymptotic Analysis

Algorithm prefixAverages1(X):

Input: An n -element array X of numbers.

Output: An n -element array A of numbers such that $A[i]$ is the average of elements $X[0], \dots, X[i]$.

```
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
        A[i] ← a/(i+1)
    return array A
```

i iterations
with
 $i=0,1,2\dots n-1$

n iterations

Example of Asymptotic Analysis

Algorithm prefixAverages1(X):

Input: An n -element array X of numbers.

Output: An n -element array A of numbers such that $A[i]$ is the average of elements $X[0], \dots, X[i]$.

```
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
        A[i] ← a/(i+1)
    return array A
```

i iterations
with
 $i=0,1,2\dots n-1$

n iterations

Analysis:
running time is $O(n^2)$

Example of Asymptotic Analysis

Algorithm prefixAverages1(X):

Input: An n -element array X of numbers.

Output: An n -element array A of numbers such that $A[i]$ is the average of elements $X[0], \dots, X[i]$.

```
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j] ← 1
        A[i] ← a/(i+1)
    return array A
```

i iterations
with
 $i=0,1,2\dots n-1$

n iterations

Analysis:
running time is $O(n^2)$

Can we do better?

Example of Asymptotic Analysis

Algorithm prefixAverages2(X):

Input: An n -element array X of numbers.

Output: An n -element array A of numbers such that $A[i]$ is the average of elements $X[0], \dots, X[i]$.

```
s ← 0
for i ← 0 to n do
    s ← s + X[i]
    A[i] ← s/(i+1)
return array A
```

Analysis:
running time is **O(n)**

Example of Asymptotic Notation (terminology)

- **Special classes of algorithms:**
 - **Logarithmic**: $O(\log n)$
 - **Linear**: $O(n)$
 - **Quadratic**: $O(n^2)$
 - **Polynomial**: $O(n^k)$, $k \geq 1$
 - **Exponential**: $O(a^n)$, $a > 1$

Recursive Algorithm analysis

Recursive Algorithm()

```
{
```

```
long int factorial(int n)
```

```
{
```

```
    if (n>=1)
```

```
        return n*factorial(n-1);
```

```
    else return 1;
```

```
}
```

```
}
```

- Here also, the function factorial() is executing number of times as long as condition is true.
- But, the analysis of recursive algorithm is different from the iterative algorithm.
- So, how to analyse the recursive algorithm?

Note: A function calling itself is called recursive function.

Recursive Algorithm analysis: example 1

$A(\text{int } n)$ → Calling Algo A with input size n, say it takes $T(n)$ times for execution

{

 if ($n > 1$) → 1 unit time

 return ($A(n-1)$); → $T(n-1)$ times to execute

}

Note: Let's apply **Back Substitution** to solve total time complexity of above algo.

$$T(n) = 1 + T(n-1) \dots (1)$$

$$T(n-1) = 1 + T((n-1)-1) = 1 + T(n-2) \dots (2); \text{ Substitute } n \text{ as } n-1 \text{ in eq (1)}$$

$$T(n-2) = 1 + T((n-2)-1) = 1 + T(n-3) \dots (3); \text{ Substitute } n \text{ as } n-2 \text{ in eq (1)}$$

Recursive Algorithm analysis: example 1

A(int n) → T(n)

```
{  
    if (n>1) → 1 unit time  
    return (A(n-1)); → T(n-1)  
}
```

$$T(n) = 1 + T(n-1) \dots \dots (1)$$

$$T(n-1) = 1 + T(n-2) \dots \dots (2)$$

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

.....

$$T(1) = 1; \text{ for checking}$$

condition n=0, Terminating condition

Now, let's do the back substitution as follows:

Substitute $T(n-1)$ from eq(2) in eq(1)

$$T(n) = 1 + T(n-1)$$

$$T(n) = 2 + T(n-2)$$

$$T(n) = 3 + T(n-3)$$

Likewise;
 $= 4 + T(n-4)$

$= \dots \dots \dots$

$= k + T(n-k); \text{ after } k \text{ iterations}$

Terminating condition is $n = 0$, when we will get $T(1) = 1$.

In the equation $k + T(n-k)$, we get $n-k = 0$ when $k=n$, let's put this value of k.

$$T(n) = n + T(n-n) = n + T(0) = n + 1 = O(n)$$

Recursive Algorithm analysis: example 2

Consider a recursive algorithm as follows:

$$T(n) = n + T(n-1), \text{ when } n > 0$$

$$= 1, \text{ when } n = 0$$

Find the time complexity.

Recursive Algorithm analysis: example 3

Consider a recursive algorithm as follows:

$$T(n) = T(n-1) + \log n, \text{ when } n > 1$$

$$= 1, \text{ when } n = 1$$

Find the time complexity.

Recursive Algorithm analysis summary

$$T(n) = T(n-1) + 1 \rightarrow O(n)$$

$$T(n) = T(n-1) + n \rightarrow O(n^2)$$

$$T(n) = T(n-1) + \log n \rightarrow O(n \log n)$$

$$T(n) = T(n-1) + n^2 \rightarrow O(n^3)$$

$$T(n) = T(n-2) + 1 \rightarrow O(n/2) \rightarrow O(n)$$

$$T(n) = T(n-200) + 1 \rightarrow O(n/200) \rightarrow O(n)$$

$$T(n) = 2T(n-1) + 1 \rightarrow O(?)$$

Recursive Algorithm analysis summary

$$T(n) = T(n-1) + 1 \rightarrow O(n)$$

$$T(n) = T(n-1) + n \rightarrow O(n^2)$$

$$T(n) = T(n-1) + \log n \rightarrow O(n \log n)$$

$$T(n) = T(n-1) + n^2 \rightarrow O(n^3)$$

$$T(n) = T(n-2) + 1 \rightarrow O(n/2) \rightarrow O(n)$$

$$T(n) = T(n-200) + 1 \rightarrow O(n/200) \rightarrow O(n)$$

$$T(n) = 2T(n-1) + 1 \rightarrow O(?) \rightarrow \mathbf{O(2^n)}$$

Dividing Recursive Algorithm

Recurrence Relation:

```
Algo Test(int n)
{
    If (n>1)
    {
        printf("%d", n);
        Test(n/2);
    }
}
```

Dividing Recursive Algorithm

Recurrence Relation:

$$T(n) = T(n/2) + 1$$

```
Algo Test(int n)
{
    if (n>1)
    {
        printf("%d", n);
        Test(n/2);
    }
}
```

Dividing Recursive Algorithm

Recurrence Relation:

$$T(n) = T(n/2) + 1$$

$$T(n) = T(n/2^2) + 2$$

$$= T(n/2^3) + 3$$

$$= T(n/2^4) + 4$$

=::::::::::::::::::

$$= T(n/2^k) + k$$

Assume $n/2^k = 1$, then $k = \log_2 n$

$$T(n) = T(1) + \log_2 n = O(\log_2 n)$$

```
Algo Test(int n)
{
    if (n>1)
    {
        printf("%d", n);
        Test(n/2);
    }
}
```

Dividing Recursive Algorithm

Recurrence Relation:

$$T(n) = T(n/2) + n \text{ for } n > 1 \text{ and}$$

$$1 \quad n=1$$

$$T(n) = ?$$

$$\begin{aligned} T(n) &= T(n/2^2) + n/2 + n \\ &= T(n/2^3) + n/2^2 + n/2 + n \\ &= T(n/2^4) + n/2^3 + n/2^2 + n/2 + n \\ &\quad \vdots \\ &= T(n/2^k) + n/2^k + n/2^{k-1} + \dots + n/2^3 + n/2^2 + n/2 + n \end{aligned}$$

$$\text{Assume } n/2^k = 1, \text{ then } k = \log_2 n$$

$$\begin{aligned} T(n) &= T(1) + n/2^k + n/2^{k-1} + \dots + n/2^3 + n/2^2 + n/2 + n \\ &= n(1/2^k + 1/2^{k-1} + \dots + 1/2^3 + 1/2^2 + 1/2 + 1/2^0) \\ &= n \sum 1/2^i \end{aligned}$$

$$T(n) = n * 1 = O(n)$$

Dividing Recursive Algorithm

Recurrence Relation:

$$T(n) = 2T(n/2) + n \text{ for } n > 1 \text{ and}$$

$$1$$

$$n=1$$

$$T(n) = ?$$

$$\begin{aligned} T(n) &= 2[2T(n/2^2) + n/2] + n = 2^2T(n/2^2) + n + n \\ &= 2^2[2T(n/2^3) + n/2^2] + n + n = 2^3T(n/2^3) + 2n + n \\ &= 2^3[2T(n/2^4) + n/2^4] + n + n + n = 2^4T(n/2^4) + 3n + n \\ &\quad \vdots \\ &= 2^{k-1}[2T(n/2^k) + n/2^{k-1}] + kn \\ &= 2^kT(n/2^k) + n + \dots + kn \end{aligned}$$

Assume $n/2^k = 1$, then $n = 2^k$ and $k = \log_2 n$

$$\begin{aligned} T(n) &= 2^kT(1) + kn \\ &= n + kn \\ &= n + n \log n \end{aligned}$$

$$T(n) = O(n \log n)$$

Master Theorem for Dividing Recursive Algorithm

General form of recurrence relation is:

$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1, b > 1, \text{ and}$$

$$f(n) = \Theta(n^k \log^p n), \text{ where } k \geq 0 \text{ and } p \text{ is any real number.}$$

If $a > b^k$, then $T(n) = \Theta(n^{\log_b^a})$

If $a = b^k$

a. If $p > -1$, then $T(n) = \Theta(n^{\log_b^a} \log^{p+1} n)$

b. If $p = -1$, then $T(n) = \Theta(n^{\log_b^a} \log \log n)$

c. If $p < -1$, then $T(n) = \Theta(n^{\log_b^a})$

If $a < b^k$

a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$

b. If $p < 0$, then $T(n) = O(n^k)$

Unit 2: Abstract Data Types

Abstract Data Types

- ADT is a mathematically specified entity that defines a set of its *instances*, with:
 - a specific interface – a collection of signatures of operations that can be invoked on an instance,
 - a set of axioms (preconditions and postconditions) that define the semantics of the operations (i.e., what the operations do to instances of the ADT, but not how)
- Types of operations:
 - **Constructors:** create an instance of that ADT.
 - **Access functions:** access the element of that ADT
 - **Manipulation procedures:** manipulate the ADT.

Abstract Data Types

- Why do we need to talk about ADTs in a DS course?
 - They serve as specifications of requirements for the building blocks of solutions to algorithmic problems
 - Provides a language to talk on a higher level of abstraction
 - ADTs encapsulate *data structures* and *algorithms* that **implement** them
 - Separate the issues of correctness and efficiency

Example: Dynamic sets

- We will deal with ADTs, instances of which are sets of some type of elements.
 - Operations are provided that change the set
- We call such class of ADTs *dynamic sets*
- **Methods:**
 - New():ADT
 - Insert(S:ADT, v:element):ADT
 - Delete(S:ADT, v:element):ADT
 - IsIn(S:ADT, v:element):boolean
- **Insert** and **Delete** – manipulation operations
- **IsIn** – Access method method

Example: Dynamic sets

- **Axioms** that define the **methods**:
 - $\text{IsIn}(\text{New}(), v) = \text{false}$
 - $\text{IsIn}(\text{Insert}(S, v), v) = \text{true}$
 - $\text{IsIn}(\text{Insert}(S, u), v) = \text{IsIn}(S, v)$, if $v \neq u$
 - $\text{IsIn}(\text{Delete}(S, v), v) = \text{false}$
 - $\text{IsIn}(\text{Delete}(S, u), v) = \text{IsIn}(S, v)$, if $v \neq u$

Other Examples

- **Simple ADTs:**
 - Stack
 - Queue
 - Dequeue

Unit 2: Array and Linked List

Data Structure (DS) Classification

- Data Structure (DS) are classified as either linear or nonlinear.
- A data structure is said to be linear if its elements form a sequence, in other words, a linear list.
- There are two basic ways of representing such DS in memory:
 - Linear relationship represented by means of sequential memory locations are called as **Array**.
 - Linear relationship represented by means of pointers are called as **Linked List**.

Data Structure (DS) Operations

- The following operations can be performed on any linear DS (an Array or Linked list):
 - **Traversal**: Processing each element in the list.
 - **Search**: Finding the location of the element with a given value or the record with a given key.
 - **Insertion**: Adding a new element in the list.
 - **Deletion**: Removing an element from the list.
 - **Sorting**: Arranging the elements in an order.
 - **Merging**: Combining two lists into a single list.

Note: A particular DS is chosen for a situation depends on the frequency of these operations.

Array Data Structure (DS)

- A linear array is a list of finite number of n of homogeneous data elements such that:
 - Elements of an array are stored in successive memory locations.
 - Elements of an array referenced by an index numbers.
- The number n of elements is called the length or size of the array.
- The index set usually starts with integers $1, 2, \dots, n$.
- Length of the array can also be obtained as:

$$\text{Length} = \text{Upper Bound (UB)} - \text{Lower Bound (LB)} + 1$$

Array Data Structure (DS) example

- Let DATA be a 6-element linear array of integers such that

DATA[1]= 247, DATA[2]= 56, DATA[3]= 429,

DATA[4]=135, DATA[5]= 87, DATA[6]= 156

DATA

It is usually denoted as:

247	56	429	135	87	156
-----	----	-----	-----	----	-----

1 2 3 4 5 6
LB UB

Length= Upper Bond (UB) - Lower Bond (LB)+1 = 6-1+1 = 6

Array Data Structure (DS) exercise

- An automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 to 1984. What is total number of elements in array AUTO?

Say $\text{AUTO}[k]$ = number of automobiles sold in the year k

$\text{LB} = 1932$, and $\text{UB} = 1984$

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

$$= 1984 - 1932 + 1 = 53$$

Array Data Structure (DS) declaration in C

- Array Declaration:

`int arr[5] = {23, 7, 12, 5, 4};` Here LB= 1, and UB= 5

Length = 5-1+1 = 5

- Each programming has its own rule of declaration of an Array. But all declaration give three information:
 - The name of the array
 - The data type of the array
 - The index set of the array
- Array declaration can be done at compile time (statically) or run time (dynamically).

Representation of Array in memory

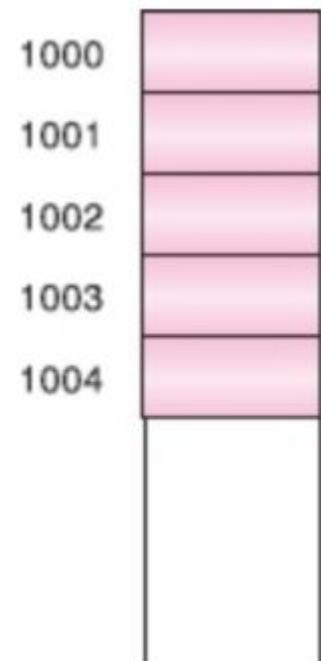
- Let LA be a linear array in the memory of the computer.
- As we know memory of the computer is simply a sequence of address locations, a location of an element in the array can be written as:

$\text{LOC}(\text{LA}[k]) = \text{address of the element } \text{LA}[k] \text{ of the array LA}$

- Base (LA) gives the base address of the array which is used to find the location of other elements in the array:

$\text{LOC}(\text{LA}[k]) = \text{Base}(\text{LA}) + w (k - \text{Lower bound})$

Here w is size of each element in the array.



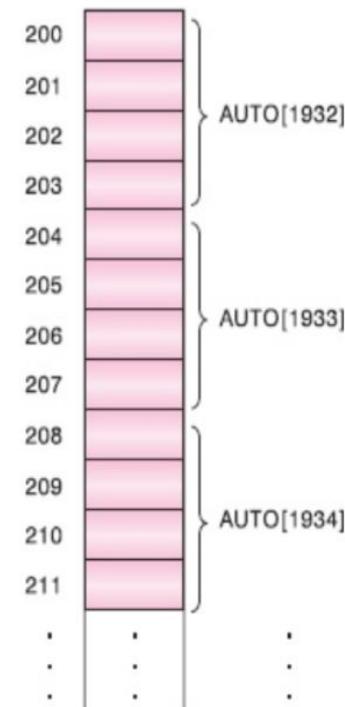
Representation of Array in memory

$$\text{LOC}(\text{LA}[k]) = \text{Base}(\text{LA}) + w (k - \text{Lower bound})$$

- Observe that time to calculate the **LOC (LA[k])** is same for any value of **k**.
- Furthermore, given any subscript **k**, we can locate and access content of **LA[k]** without scanning any other element of **LA**.

Exercise: Suppose an automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 to 1984. The $\text{BASE}(\text{AUTO}) = 200$ and $w = 4$, then find the following:

$$\text{LOC} (\text{AUTO}[1965]) = ?$$



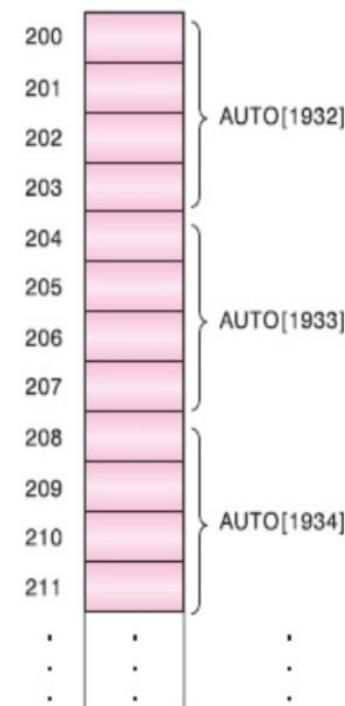
Representation of Array in memory

$$\text{LOC}(\text{LA}[k]) = \text{Base}(\text{LA}) + w (k - \text{Lower bound})$$

- Observe that time to calculate the **LOC (LA(k))** is same for any value of **k**.
- Furthermore, given any subscript **k**, we can locate and access content of **LA[k]** without scanning any other element of **LA**.

Exercise: Suppose an automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 to 1984. The $\text{BASE}(\text{AUTO}) = 200$ and $w = 4$, then find the following:

$$\begin{aligned}\text{LOC}(\text{AUTO}[1965]) &= \text{BASE}(\text{AUTO}) + w (k - \text{LB}) \\ &= 200 + 4 (1965 - 1932) = 332\end{aligned}$$



Linear Array property

- Linear Array can be indexed.
 - A collection A of data elements is said to be indexed if any element of A , which we shall call A_k , can be located and processed in a time that is independent of k .
- This is very important property of linear arrays.
- The other linear DS linked list do not have this property.

Array Operation: Traversing

- Traversing an array means visiting each element of an array:
 - To print the contents of each element of array
 - To count the number of elements
- **Algorithm to traverse an array:**

(Traversing a Linear Array) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter.] Set K := LB.
2. Repeat Steps 3 and 4 while K ≤ UB.
3. [Visit element.] Apply PROCESS to LA[K].
4. [Increase counter.] Set K := K + 1.
[End of Step 2 loop.]
5. Exit.

Array Operation: Traversing

- Traversing an array means visiting each element of an array:
 - To print the contents of each element of array
 - To count the number of elements
- **Algorithm to traverse an array:**

(Traversing a Linear Array) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter.] Set K := LB. \longrightarrow 1
 2. Repeat Steps 3 and 4 while $K \leq UB$. \longrightarrow $n+1$
 3. [Visit element.] Apply PROCESS to $LA[K]$. \longrightarrow n
 4. [Increase counter.] Set $K := K + 1$. \longrightarrow n
- [End of Step 2 loop.]
5. Exit.

Complexity:

- Worst case: $O(n)$
- Best Case: $O(n)$
- Average case: $O(n)$

Array Operation: Insertion and Deletion

- Let A be an array. Insertion means adding a new element in the array A.
- Inserting a new element at the “end” of a linear array can be easily done provided memory space is available.
- On the other hand, inserting element at any other place (say in the middle or beginning) would require, on an average, half of the elements must be moved downwards to new locations.
- Similarly, deleting an element at the “end” can be done easily, but deleting an element somewhere in the middle would require that each subsequent element be moved one location upward in order to fill up the array.

Array Operation: Insertion example

- Let **NAME** is an array of string which can store 8-element in alphabetical order.
- But initially we have only five names as shown in the figure (a).
- Suppose now we need to add name “Ford” in the array **NAME**.
- It should be added at index 3, after Davis and Before Johnson.
- Hence, the elements Johnson, Smith, and Wagner are first shifted one position downward.
- Then, index 3 would be empty and hence, Ford would be added.

	NAME
1	Brown
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

(a)

Array Operation: Insertion example

- After addition it would be as shown in Figure (b).

NAME	
1	Brown
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

(a)

NAME	
1	Brown
2	Davis
3	Ford
4	Johnson
5	Smith
6	Wagner
7	
8	

(b)

Array Operation: Insertion example

- After adding Taylor the NAME would be

	NAME
1	Brown
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

(a)

	NAME
1	Brown
2	Davis
3	Ford
4	Johnson
5	Smith
6	Wagner
7	
8	

(b)

	NAME
1	Brown
2	Davis
3	Ford
4	Johnson
5	Smith
6	Taylor
7	Wagner
8	

(c)

Array Operation: Deletion example

- In the Figure (c) , if we remove Davis every element below will shifted one position upward.

NAME	
1	Brown
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

(a)

NAME	
1	Brown
2	Davis
3	Ford
4	Johnson
5	Smith
6	Wagner
7	
8	

(b)

NAME	
1	Brown
2	Davis
3	Ford
4	Johnson
5	Smith
6	Taylor
7	Wagner
8	

(c)

NAME	
1	Brown
2	Ford
3	Johnson
4	Smith
5	Taylor
6	Wagner
7	
8	

(d)

Array Operation: Insertion Algorithm

(Inserting into a Linear Array) **INSERT (LA, N, K, ITEM)**

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter.] Set $J := N$.
2. Repeat Steps 3 and 4 while $J \geq K$.
3. [Move Jth element downward.] Set $LA[J + 1] := LA[J]$.
4. [Decrease counter.] Set $J := J - 1$.
[End of Step 2 loop.]
5. [Insert element.] Set $LA[K] := ITEM$.
6. [Reset N.] Set $N := N + 1$.
7. Exit.

Array Operation: Insertion Algorithm Complexity

(Inserting into a Linear Array) INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter.] Set $J := N$.
2. Repeat Steps 3 and 4 while $J \geq K$.
3. [Move Jth element downward.] Set $LA[J + 1] := LA[J]$.
4. [Decrease counter.] Set $J := J - 1$.
[End of Step 2 loop.]
5. [Insert element.] Set $LA[K] := ITEM$.
6. [Reset N.] Set $N := N + 1$.
7. Exit.

Complexity:

- Worst case: $O(n)$
- Best Case: $O(1)$
- Average case: $O(n)$

Array Operation: Deletion Algorithm

(Deleting from a Linear Array) **DELETE(LA, N, K, ITEM)**

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].

2. Repeat for $J = K$ to $N - 1$:

[Move $J + 1$ st element upward.] Set LA[J] := LA[J + 1].

[End of loop.]

3. [Reset the number N of elements in LA.] Set N := N - 1.

4. Exit.

Array Operation: Deletion Algorithm Complexity

(Deleting from a Linear Array) **DELETE(LA, N, K, ITEM)**

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].

2. Repeat for J = K to N – 1:

[Move J + 1st element upward.] Set LA[J] := LA[J + 1].

[End of loop.]

3. [Reset the number N of elements in LA.] Set N := N – 1.

4. Exit.

Complexity:

- Worst case: O(n)
- Best Case: O(1)
- Average case: O(n)

Array Operation: Remarks

- We emphasize that if many deletions and insertions are to be made in a collection of data elements, then a linear array may not be the most efficient way of storing the data.

Bubble Sort Use case:

- Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order.
- Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a **bubble sort**.

Array Use Case: Bubble Sort

Bubble Sort Use case:

- Suppose we have an array of elements as: **-2, 45, 0, 11, -9**

First Iteration: Compare and Swap

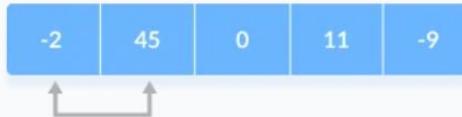
- Starting from the first index, compare the first and the second elements.
- If the first element is greater than the second element, they are swapped.
- Now, compare the second and the third elements. Swap them if they are not in order.
- The above process goes on until the last element. After each iteration, the largest element among the unsorted elements is placed at the end.

Let's see the process:

Array Use Case: Bubble Sort

step = 0

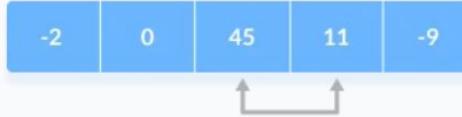
i = 0



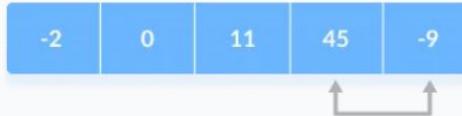
i = 1



i = 2

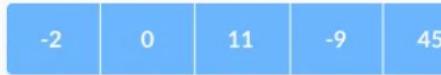
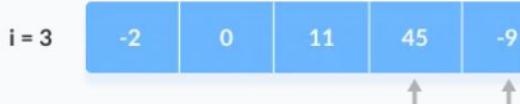
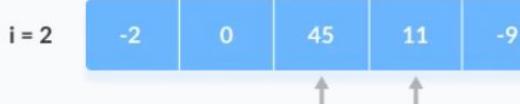
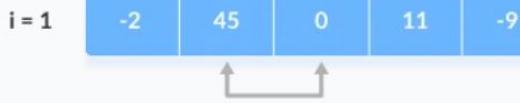
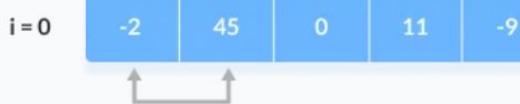


i = 3

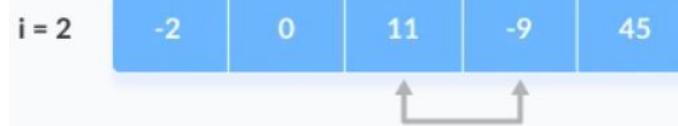
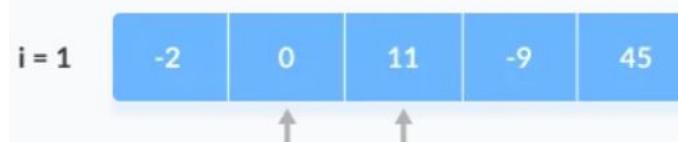


Array Use Case: Bubble Sort

step = 0

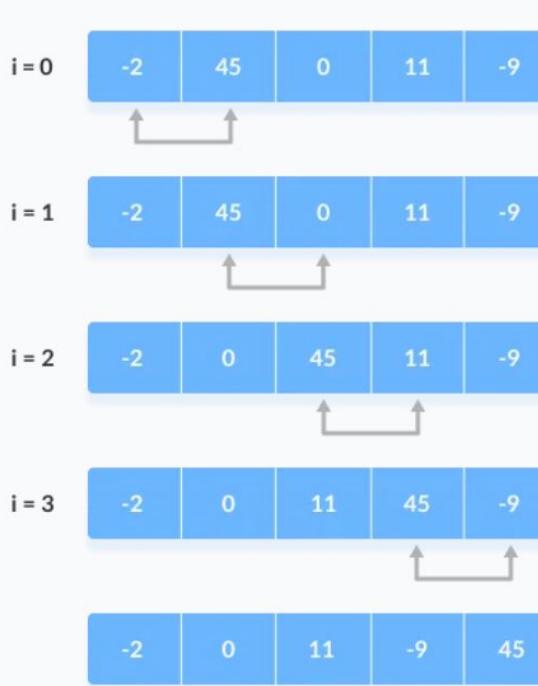


step = 1

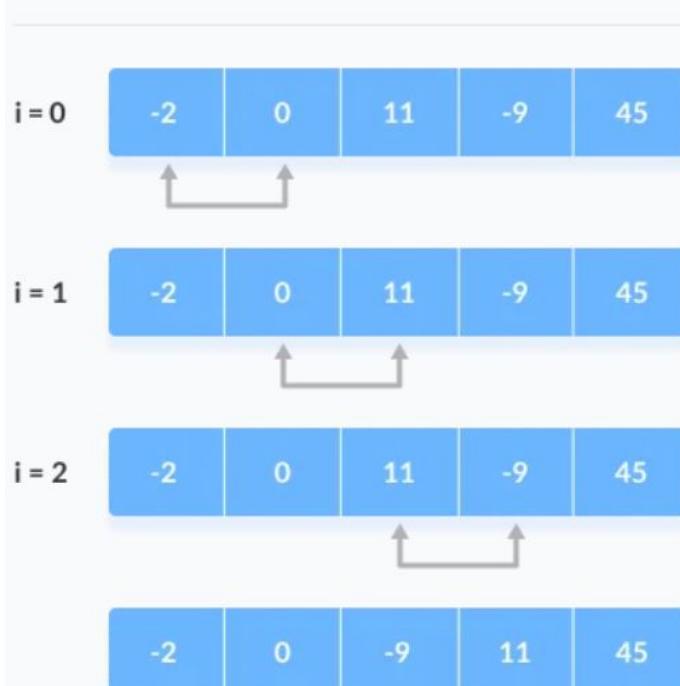


Array Use Case: Bubble Sort

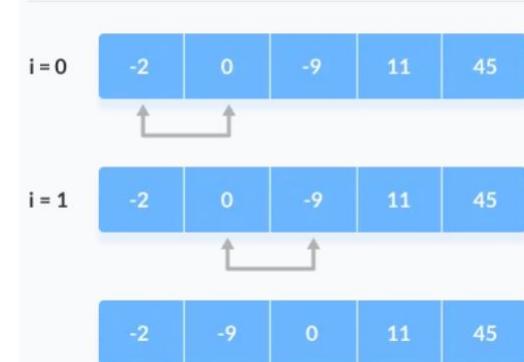
step = 0



step = 1

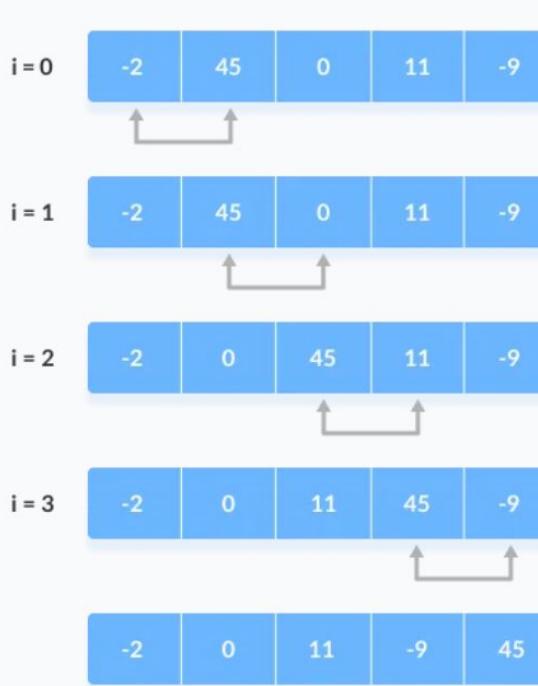


step = 2



Array Use Case: Bubble Sort

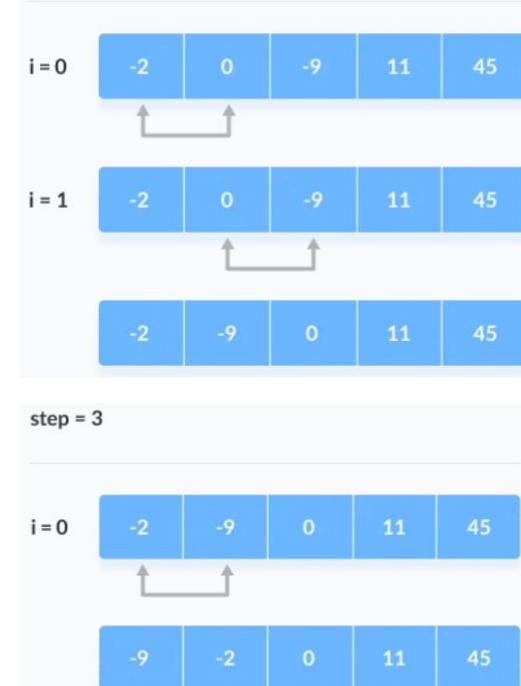
step = 0



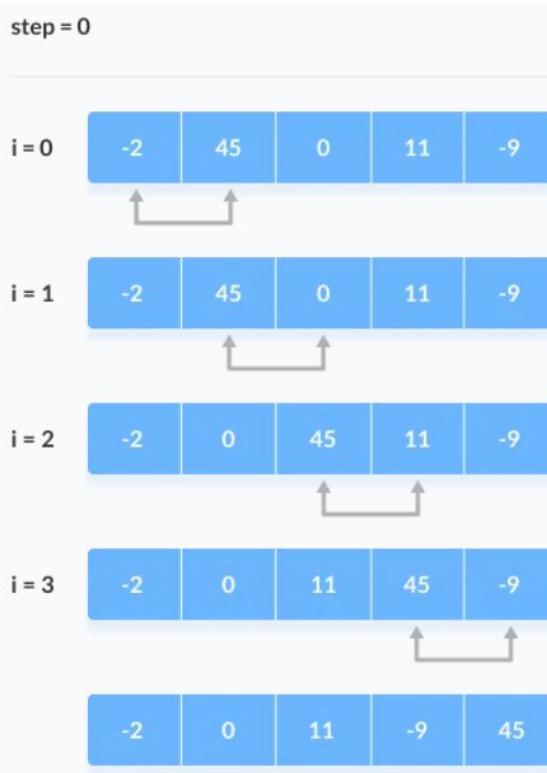
step = 1



step = 2



Array Use Case: Bubble Sort Algorithm



Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for K = 1 to N – 1.
2. Set PTR := 1. [Initializes pass pointer PTR.]
3. Repeat while PTR ≤ N – K: [Executes pass.]
 - (a) If DATA[PTR] < DATA[PTR + 1], then:
Interchange DATA[PTR] and DATA[PTR + 1].
[End of If structure.]
 - (b) Set PTR := PTR + 1.
[End of inner loop.]
4. Exit.

Array Use Case: Bubble Sort Algorithm Complexity

- Time for a sorting algorithm is measured in terms of number of comparisons.
- During first pass it is $n-1$. In 2nd pass it is $n-2$ and so on.
- Total time:=

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

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

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for $K = 1$ to $N - 1$.
2. Set PTR := 1. [Initializes pass pointer PTR.]
3. Repeat while $PTR \leq N - K$: [Executes pass.]
 - (a) If $DATA[PTR] < DATA[PTR + 1]$, then:
Interchange $DATA[PTR]$ and $DATA[PTR + 1]$.
[End of If structure.]
 - (b) Set PTR := PTR + 1.
[End of inner loop.]
4. Exit.

Array Use Case: Bubble Sort Algorithm Complexity

- Bubble sort employs two loops: an inner loop and an outer loop.
- The inner loop performs $O(n)$ comparisons deterministically.

Worst Case Complexity:

- In the worst-case scenario, the outer loop runs $O(n)$ times.
- As a result, the worst-case time complexity of bubble sort is $O(n \times n) = O(n^2)$.

Best Case Complexity:

- In the best-case scenario, the array is already sorted, but just in case, bubble sort performs $O(n)$ comparisons. As a result, the time complexity of bubble sort in the best-case scenario is $O(n)$.

Array Use Case: Bubble Sort Algorithm Complexity

- Bubble sort employs two loops: an inner loop and an outer loop.
- The inner loop performs $O(n)$ comparisons deterministically.

Average Case Complexity:

- Bubble sort may require $(n/2)$ passes and $O(n)$ comparisons for each pass in the average case.
- As a result, the average case time complexity of bubble sort is $O(n/2 \times n) = O(n/2 \times n) = O(n/2 \times n) = O(n^2)$.

- Suppose DATA is a linear array of n elements.
- To search an ITEM in the DATA through linear search could be:
 - Compare ITEM with each element in the DATA one by one.
 - First we check DATA[1]= ITEM, the DATA[2] =ITEM and so on.
- Suppose DATA is 13, 9, 21, 15, 39, 19, 27, and say searched ITEM =39

39						
13	9	21	15	39	19	27
0	1	2	3	4	5	6

39						
13	9	21	15	39	19	27
0	1	2	3	4	5	6

(Linear Search) LINEAR(DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA[N + 1] := ITEM.
2. [Initialize counter.] Set LOC := 1.
3. [Search for ITEM.]

 Repeat while DATA[LOC] \neq ITEM:

 Set LOC := LOC + 1.

 [End of loop.]

4. [Successful?] If LOC = N + 1, then: Set LOC := 0.
5. Exit.

- Observe that Step 1 guarantees that the loop in Step 3 must terminate.
- Without Step 1, the repeat statement in Step 3 must be replaced by the following statement, which involves two comparisons, not one:

Repeat while LOC \leq N and DATA[LOC] \neq ITEM:

- On the other hand, in order to use step 1, one must guarantee that there is unused memory location at N+1.

- As noted the complexity of linear search algorithm is measured by the number of $f(n)$ of comparisons required to find the ITEM in DATA, where DATA has n elements.

Worst Case:

- Worst case occurs when one must search through the entire array but ITEM does not appear in DATA. In this case algorithm requires
$$f(n) = n+1 \text{ comparisons.}$$

Therefore, in worst case the complexity is $O(n)$.

- As noted the complexity of linear search algorithm is measured by the number of $f(n)$ of comparisons required to find the ITEM in DATA, where DATA has n elements.

Average Case:

- When the element to be searched is in the middle of the array, the average case of the Linear Search Algorithm is $O(n)$.

Best Case:

- The element being searched could be found in the first position. In this case, the search ends with a single successful comparison. Thus, in the best-case scenario, the linear search algorithm performs $O(1)$ operations.

- Binary Search is a searching algorithm for finding an element's position in a sorted array.
- In this approach, the element is always searched in the middle of a portion of an array.

Note: Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

- Binary Search Algorithm can be implemented in two ways which are discussed below.
 - Iterative Method
 - Recursive Method

- The recursive method follows the divide and conquer approach. The general steps for both methods are discussed below.

Suppose the initial Array is : [3, 4, 5, 6, 7, 8, 9]

- Let $x = 4$ be the element to be searched.

Step 1: Set two pointers low and high at the lowest and the highest positions respectively.



Step 2: Find the middle element mid of the array i.e.,

$$\text{mid} = (\text{low} + \text{high})/2 = 6$$

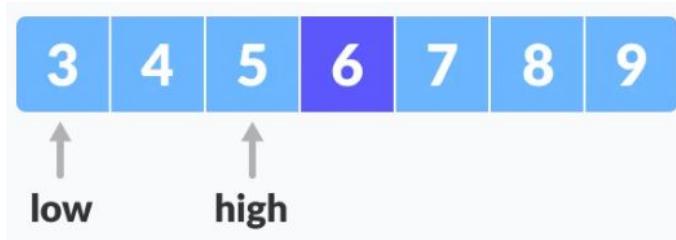


Step 3: If **x == mid**, then return mid. Else, compare the element to be searched with mid.

Step 4: If **x > mid**, compare x with the middle element of the elements on the right side of mid. This is done by **setting low to low = mid + 1**.

Step 5: Else, compare x with the middle element of the elements on the left side of mid. This is done by **setting high to high = mid - 1**.

In our case, **x= 4** and **mid =6**, so **4< 6**, hence, we will go to **step 5**.



Step 6: Repeat steps 2 to 5 until low meets high.



Here, $x = \text{mid} = 4$, Found.



Array Use Case: Binary Search Algorithm

Iteration method:

```
do until the pointers low and high meet each other.  
    mid = (low + high)/2  
    if (x == arr[mid])  
        return mid  
    else if (x > arr[mid]) // x is on the right side  
        low = mid + 1  
    else                      // x is on the left side  
        high = mid - 1
```

Array Use Case: Binary Search Algorithm

Recursion method:

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > arr[mid]          // x is on the right side
            return binarySearch(arr, x, mid + 1, high)
        else                          // x is on the left side
            return binarySearch(arr, x, low, mid - 1)
```

Array Use Case: Binary Search Algorithm

(Binary Search) **BINARY**(DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]

 Set BEG := LB, END := UB and MID = INT((BEG + END)/2).

2. Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.

3. If ITEM < DATA[MID], then:

 Set END := MID – 1.

 Else:

 Set BEG := MID + 1.

 [End of If structure.]

4. Set MID := INT((BEG + END)/2).

 [End of Step 2 loop.]

5. If DATA[MID] = ITEM, then:

 Set LOC := MID.

 Else:

 Set LOC := NULL.

 [End of If structure.]

6. Exit.

Array Use Case: Binary Search Algorithm Complexity

Best case: O(1)-Mid Element is the searched element

Worse case: O(logN)- Consider the searched element is either 1st or last element. In that case every time after finding MID we will either go to the left or right of mid until we don't reach the searched element. So we reduce the number of comparisons by half everytime we find new MID.

Suppose total elements in the array is 8:

1st time: $8/2 = 4$ this can be written as $8/2^1=4$

2nd time: $4/2 = 2$ this can be written as $8/2^2=2$

3rd time $2/2 = 1$ this can be written as $8/2^3=1$

Average case: O(logN)

In general $8/2^3 = 1$ can be written as $N/2^K = 1$

Take log both side:

$$\log N = K \log 2$$

Therefore, k - number of comparison= **O (logN)**

Array Use Case: Binary Search Algorithm Complexity

Worse case: O(logN)-

Initial length of array = n

Iteration 1 - Length of array = n/2

Iteration 2 - Length of array = (n/2)/2=n/2²

Iteration k - Length of array = n/2^k

After k iterations, the size of the array becomes 1 (narrowed down to the first element or last element only).

Length of array = $n/2^k=1 \Rightarrow n=2^k$

Applying log function on both sides: $\Rightarrow \log_2(n)=\log_2(2^k)$

$\Rightarrow \log_2(n)=k*\log_2(2) = k$

$\Rightarrow k=\log_2(n)$

Limitation of Array DS

- The term list refers to a linear collection of data items as the one shown in Figure (a).
- Frequently, we add or delete elements from the list.
 - Say after adding three new items and deleting two items the new list is as shown in Figure (b).
- That means a dynamic list involves multiple access of the data.
- We saw organization of such list through [Array](#) DS earlier.
 - We can clearly say that Array representation of such lists are **expensive**.
- The other way of representing such list is using [Linked List](#).

Milk
eggs
butter
tomatoes
apples
oranges
bread

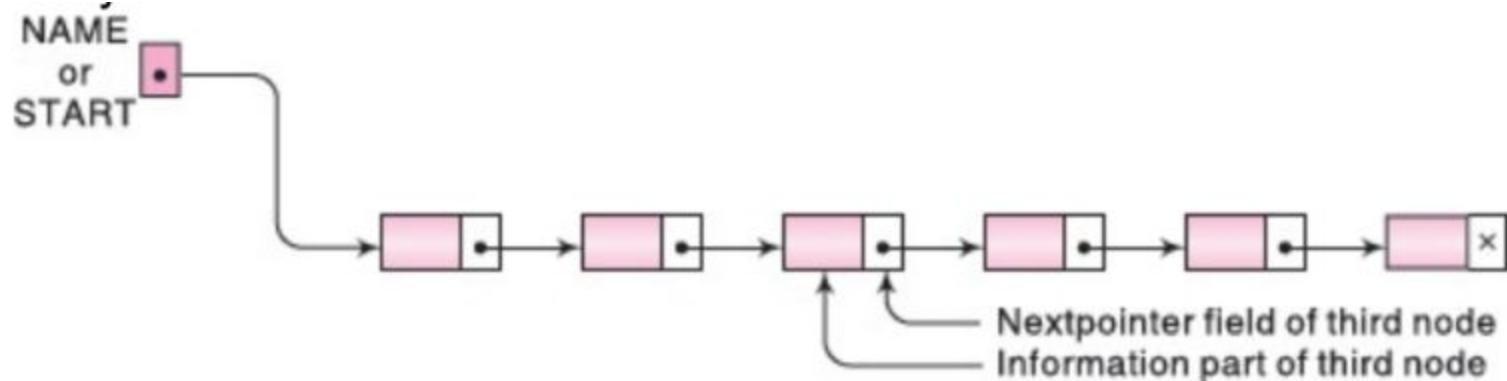
(a)

...
Milk
eggs
tomatoes
apples
bread
chicken
corn
lettuce

(b)

Linked Lists

- A linked list, or One-way list, is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**.
- That is each node is divided into two parts:
 - The first part contains the **information** of the element
 - The second part contains the address of the next node in the list called as **link field** or **nextpointer field**



Linked Lists

- A linked list is a data structure which can change during the execution of a program.
 - Successive elements are connected by pointers.
 - Last element points to **NULL**.
 - It can grow or shrink in size during execution of a program.
 - It can be made just as long as required.
 - It does not waste memory space.
- Linked list provide flexibility in allowing the items to be rearranged efficiently.
 - Insert an element
 - Delete an element

Illustration: Insertion

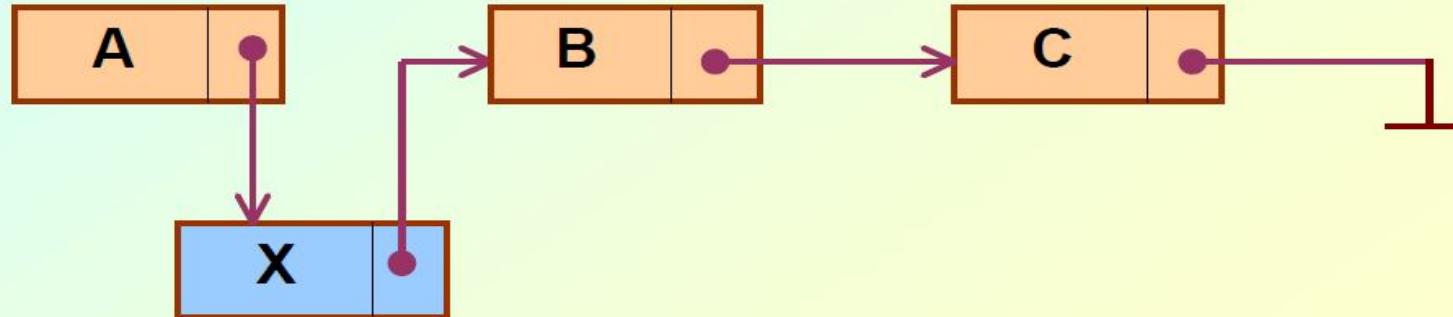
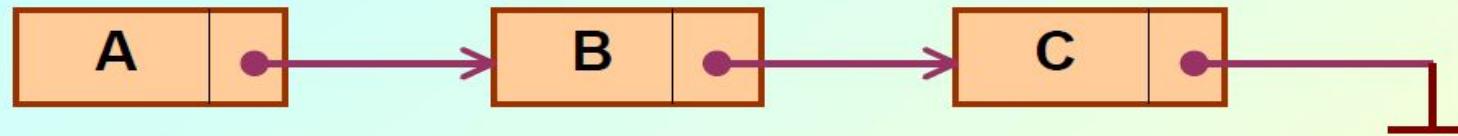
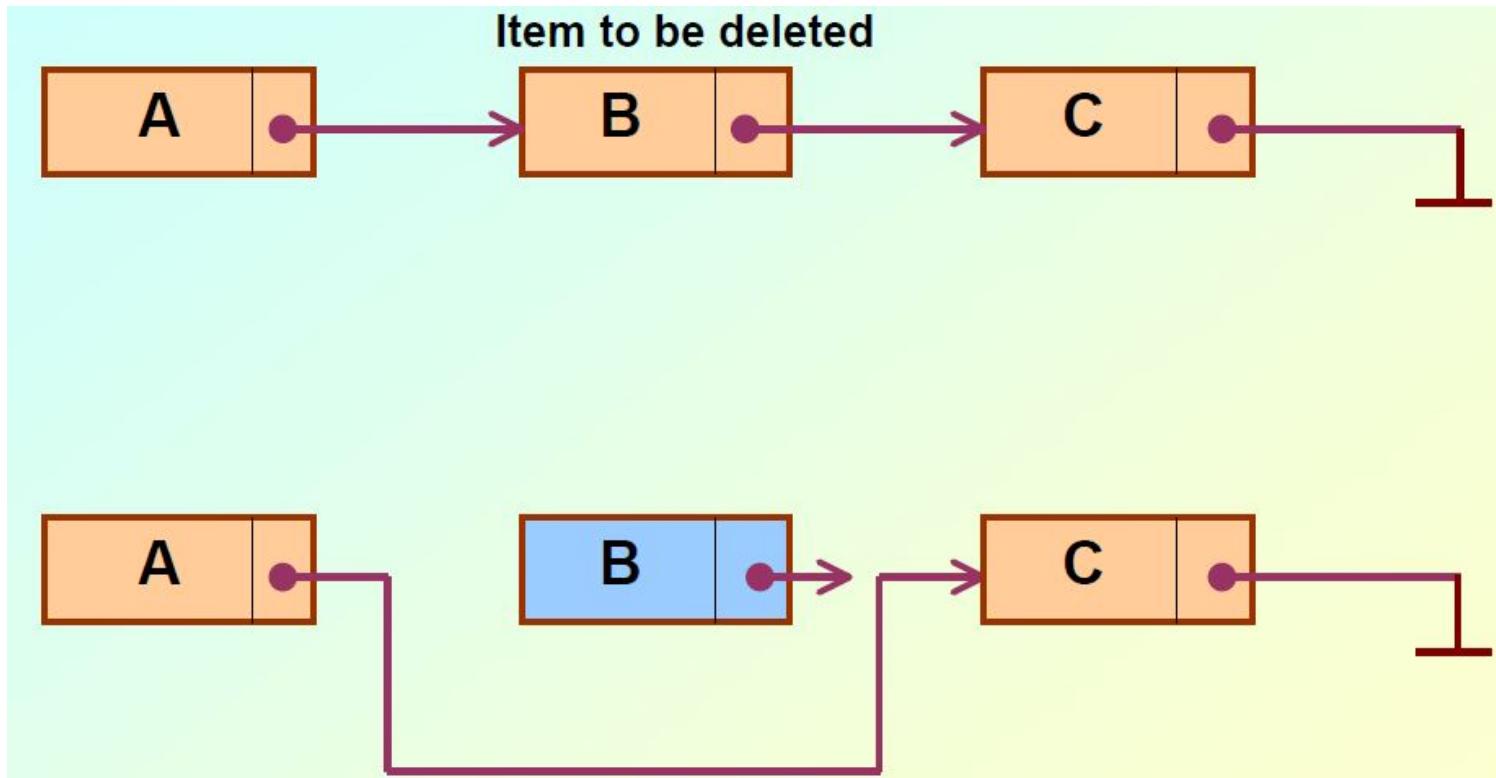


Illustration: Deletion

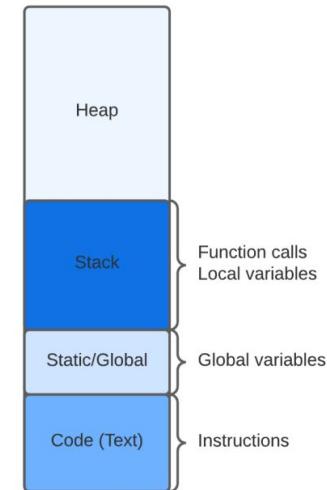


Arrays Vs Linked list

- **Arrays are suitable for:**
 - Inserting/deleting an element at the end.
 - Randomly accessing any element.
 - Searching the list for a particular value.
- **Linked lists are suitable for:**
 - Inserting an element.
 - Deleting an element.
 - Applications where sequential access is required.
 - In situations where the number of elements cannot be predicted beforehand.

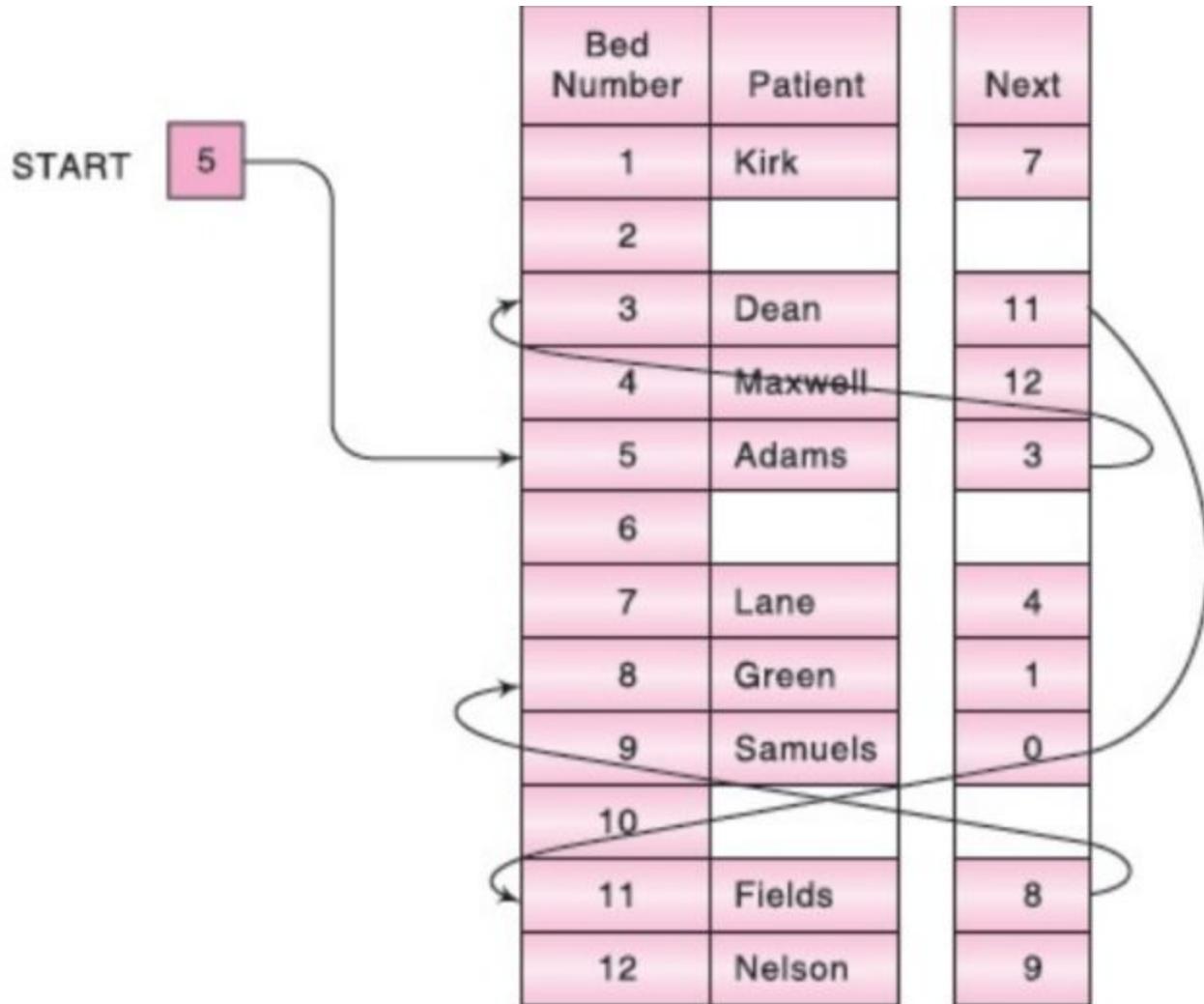
Arrays Vs Linked list

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.



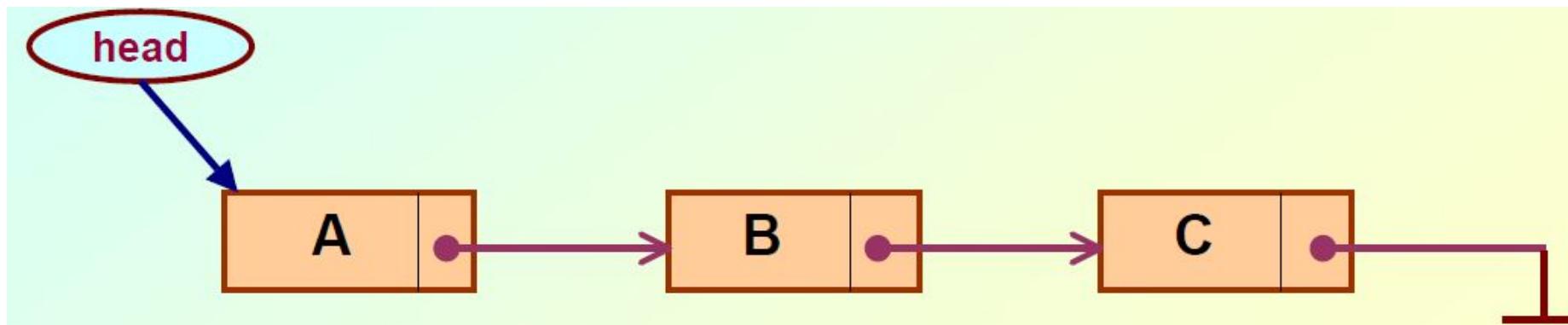
Eg: Linked List

Name	Bed No.
Adams	5
Dean	3
Fields	11
Green	8
Kirk	1
Lane	7
Maxwell	4
Nelson	9
Samuels	0



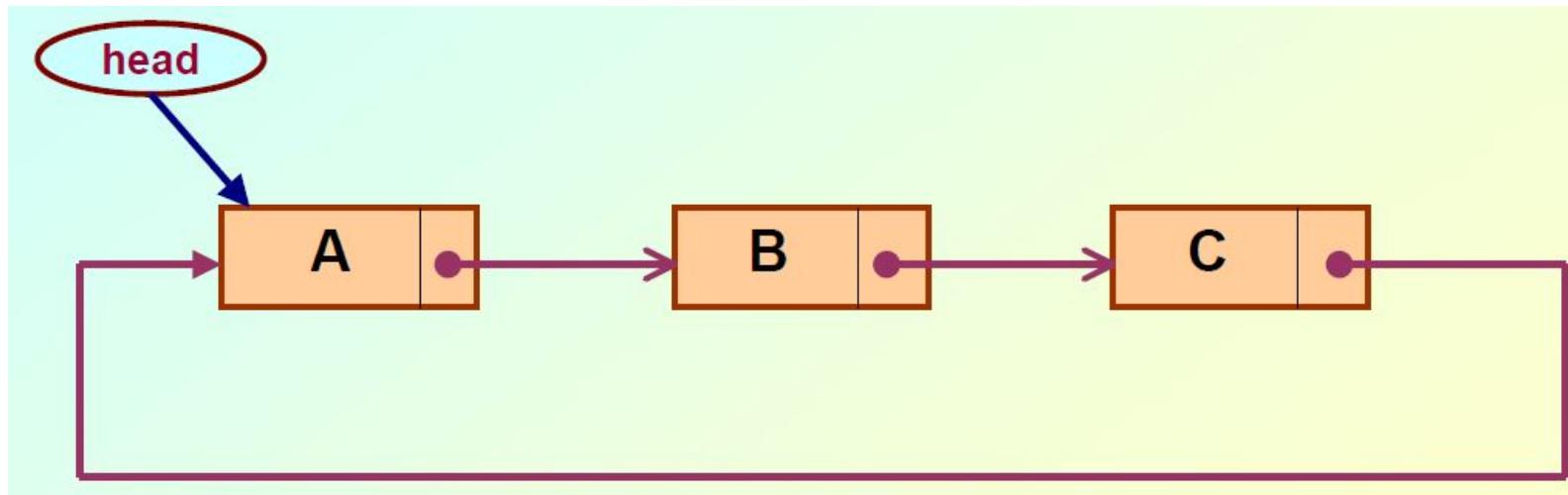
Types of linked list

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
 - Linear singly-linked list (or simply linear list)
 - One we have discussed so far.



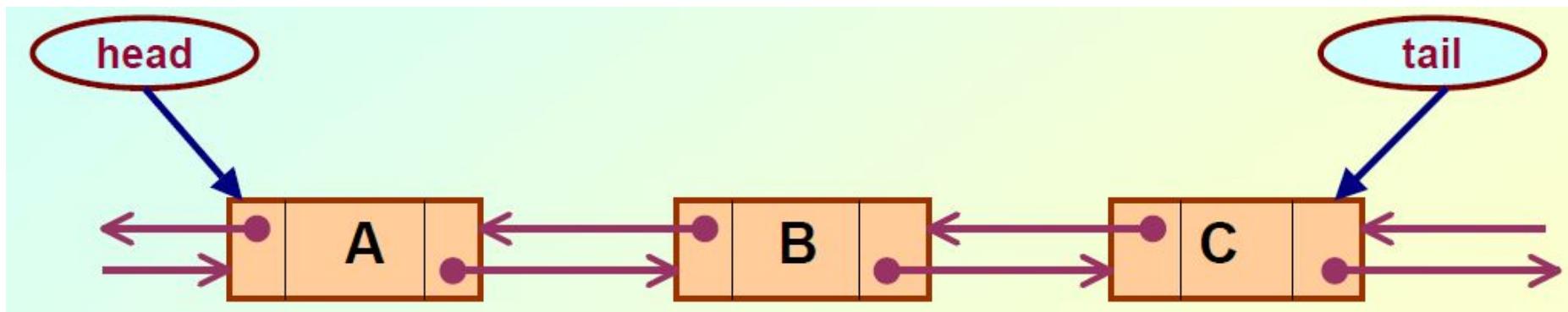
Types of linked list

- Circular linked list:
 - The pointer from the last element in the list points back to the first element.



Types of linked list

- **Doubly linked list:**
 - Pointers exist between adjacent nodes in both directions.
 - The list can be traversed either forward or backward.
 - Usually two pointers are maintained to keep track of the list, head and tail.

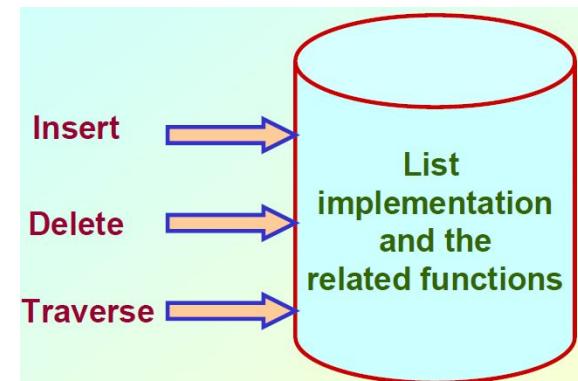


Basic operations on a linked list

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

linked list as an abstract data type

- What is an abstract data type?
 - It is a data type defined by the user.
 - Typically more complex than simple data types like int, float, etc.
- Why abstract?
 - Because details of the implementation are **hidden**.
 - When you do some operation on the list, say insert an element, you just call a function.
 - Details of how the list is implemented or how the insert function is written is no longer required.





Representation of linked list in memory

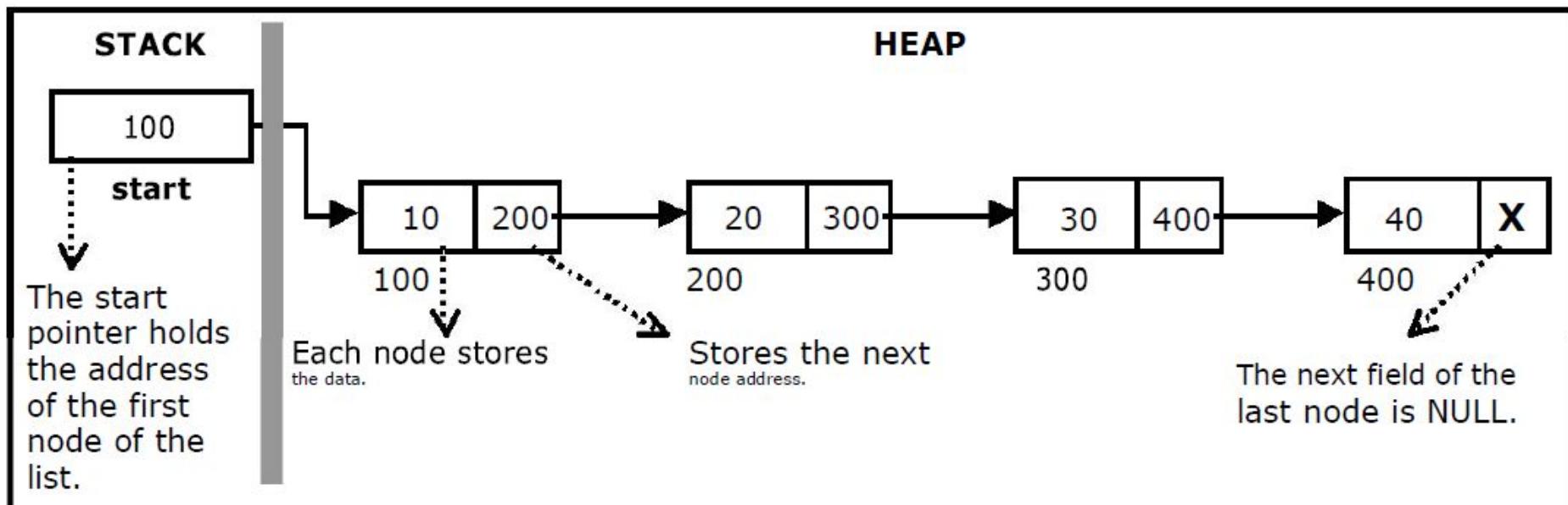
- Till now we have been seeing the linear list representation in the memory as an Array as shown in the figure, where elements were stored in a consecutive memory locations.
- Now, we would see the other way of representing the same array or list that is in a non-consecutive memory locations called as [linked list](#).
- To represent the linked list in memory we will use user defined data types called “Structure”.
 - Memory will be allocated to every structure [variable/node](#).
 - Those nodes are linked together by means of pointers.

Single linked list

- A linked list allocates space for each element separately in its own block of memory called a "**node**".
- Each node contains two fields; a "**data**" field to store whatever element, and a "**next**" field which is a pointer used to link to the next node.
- Each node is allocated in the heap using **malloc()**, so the node memory continues to exist until it is explicitly de-allocated using **free()**.
- The front of the list is a pointer to the “**start**” node.

Single linked list

- A single linked list is shown in figure below:



Single linked list: Implementation

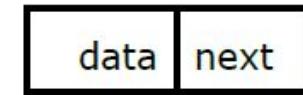
- Let's create the 'Start' node through which we will access other nodes.
- Also create the a node of the list with a data item and a next pointer.
 - This is called self-referential structure.

```
struct slinklist
{
    int data;
    struct slinklist* next;
};

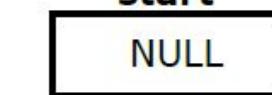
typedef struct slinklist node;

node *start = NULL;
```

node:



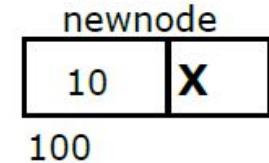
Empty list:



Single linked list: Creating a node

- The function `getnode()`, is used for **creating** a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally **returns the address** of the node.

```
node* getnode()
{
    node* newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: "); scanf("%d",
    &newnode -> data);
    newnode -> next = NULL;
    return newnode;
}
```



Single linked list: Creating a list with ‘n’ nodes

- The following steps are to be followed to create ‘n’ number of nodes:
 - Get the new node using `getnode()`

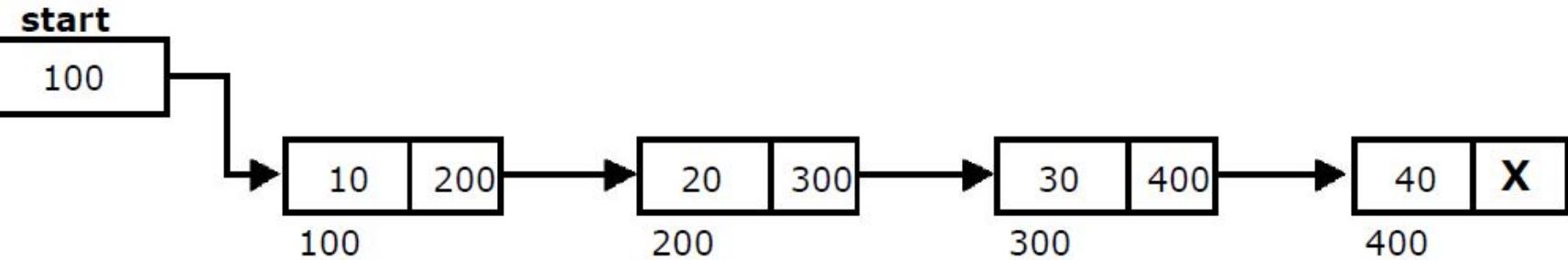
`newnode = getnode();`

- If the list is empty, assign new node as start.

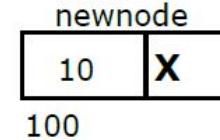
`start = newnode;`

- If the list is not empty, follow the steps given below:
 - The `next` field of the `new node` is made to point the `first node` (i.e. `start` node) in the list by assigning the address of the first node.
 - The `start pointer` is made to point the `new node` by assigning the address of the new node.
- Repeat the above steps ‘n’ times.

Single linked list: Creating a list with 'n' nodes



```
node* getnode()
{
    node* newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: "); scanf("%d",
    &newnode -> data);
    newnode -> next = NULL;
    return newnode;
}
```

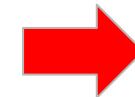


Single linked list: Creating a list with 'n' nodes

```
vo id createlist(int n)
{
    int i;
    node * new node;
    node *temp;
    for(i = 0; i < n ; i+ +)
    {
        new node = getnode();
        if(start == NULL)
        {
            start = new node;
        }
        else
        {
            temp = start;
            while(temp - > next != NULL)
                temp = temp - > next;
            temp - > next = new node;
        }
    }
}
```

Single linked list: Creating a list with 'n' nodes

```
void createlist(int n)
{
    int i;
    node * new node;
    node *temp;
    for(i = 0; i < n ; i+ +)
    {
        new node = getnode();
        if(start == NULL)
        {
            start = new node;
        }
        else
        {
            temp = start;
            while(temp - > next != NULL)
                temp = temp - > next;
            temp - > next = new node;
        }
    }
}
```



```
void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    for (i=0; i<n; i++)
    {
        newnode= getnode();
        if (start=NULL)
        {
            start=temp=newnode;
        }
        else
        {
            temp->next=newnode;
            temp=newnode;
        }
    }
}
```

Single linked list: Inserting new node in a list

- One of the most primitive operations that can be done in a singly linked list is the insertion of a node.
- Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data.
- The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL.
- The new node can then be inserted at three different places namely:
 - Inserting a node at the beginning.
 - Inserting a node at the end.
 - Inserting a node at intermediate position.

Single linked list: Inserting new node at beginning

- The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using `getnode()`

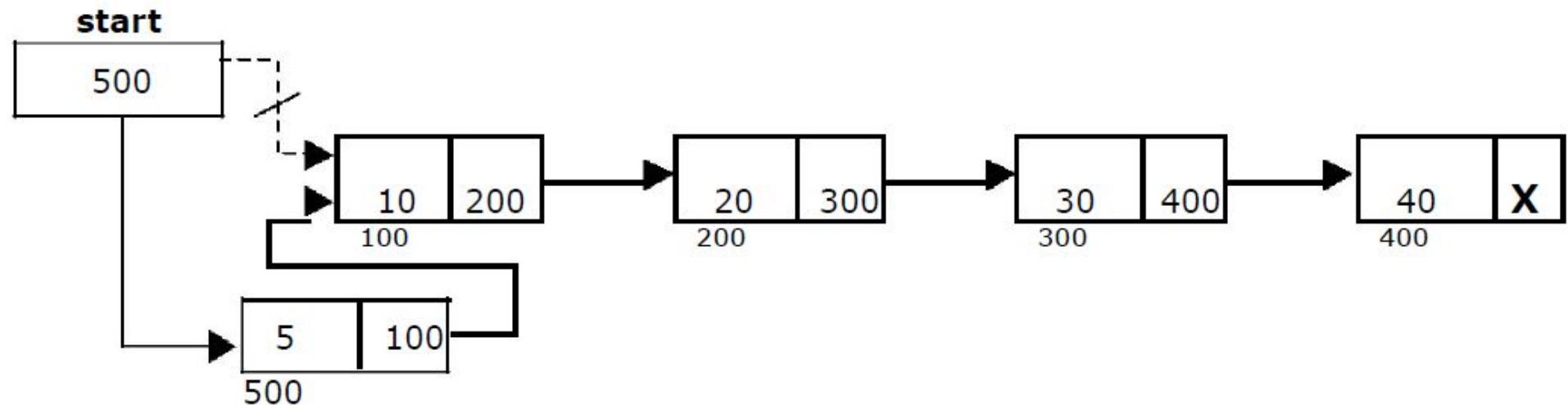
`newnode = getnode();`

- If the list is empty then `start = newnode.`
 - If the list is not empty, follow the steps given below:

`newnode -> next = start;`

`start = newnode;`

Single linked list: Inserting new node at beginning



Single linked list: Inserting new node at beginning

The function `insert_at_beg()`, is used for inserting a node at the beginning:

```
void insert_at_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        newnode -> next = start;
        start = newnode;
    }
}
```

Single linked list: Inserting new node at the end

- The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()

newnode = getnode();

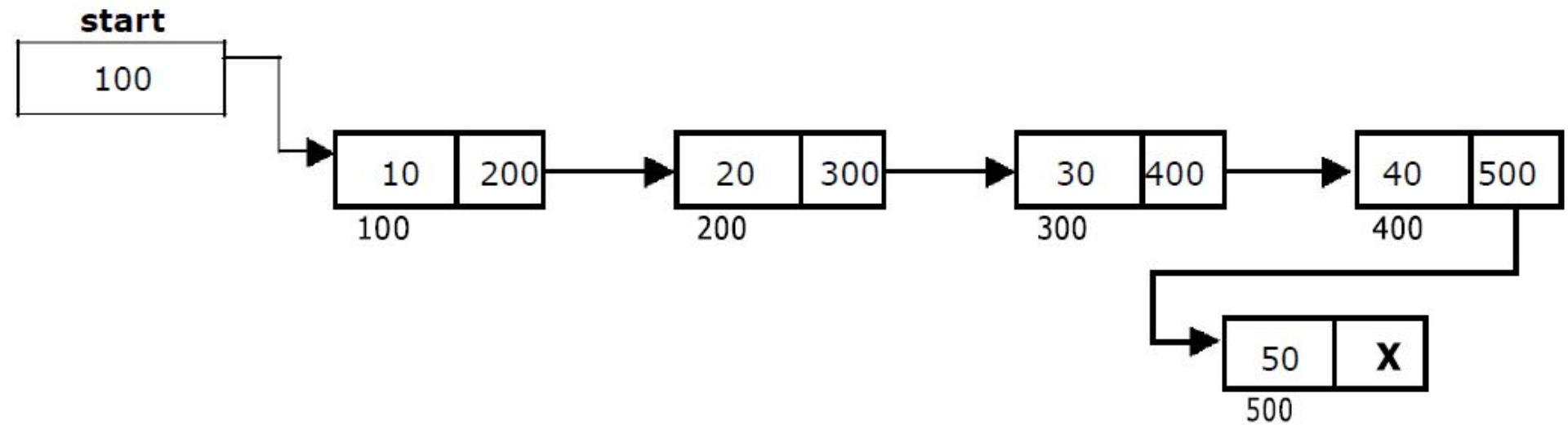
- If the list is empty then

start = temp=newnode.

- If the list is not empty follow the steps of else:

```
if (start=NULL)
{
    start=temp=newnode;
}
else
{
    while(temp->next!=NULL)
    {
        temp=temp->next;
    }
    temp->next=newnode;
}
```

Single linked list: Inserting new node at the end



Single linked list: Inserting new node at the end

The function `insert_at_last()`, is used for inserting a node at the last:

```
void insert_at_end()
{
    node *newnode;
    node *temp;
    newnode= getnode();
    if (start=NULL)
    {
        start=temp=newnode;
    }
    else
    {
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=newnode;
    }
}
```

Single linked list: Inserting new node in between

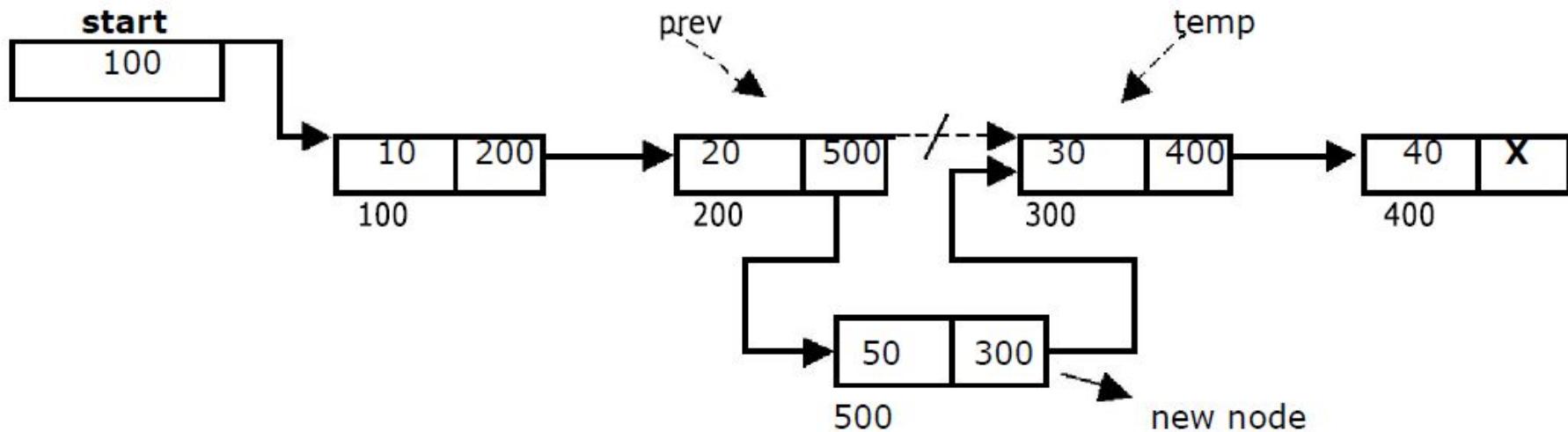
- The following steps are followed, to insert a new node in an intermediate position in the list:
 - Get the new node using `getnode()`.
`newnode = getnode();`
 - Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by `countnode()` function.
 - Store the starting address (which is in start pointer) in `temp` and `prev` pointers. Then traverse the `temp` pointer upto the specified position followed by `prev` pointer.

Single linked list: Inserting new node in between

- After reaching the specified position, follow the steps given below:

`prev -> next = newnode;`

`newnode -> next = temp;` Let the intermediate position be 3.



Single linked list: Inserting new node in between

```
void insert_at_mid()
{
    node *newnode, *temp, *prev;
    int pos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos > 1 && pos < nodectr)
    {
        temp = prev = start;
        while(ctr < pos)
        {
            prev = temp;
            temp = temp -> next;
            ctr++;
        }
        prev -> next = newnode;
        newnode -> next = temp;
    }
    else
    {
        printf("position %d is not a middle position", pos);
    }
}
```

```
int countnode(node *st)
{
    if(st == NULL)
        return 0;
    else
        return(1 + countnode(st -> next));
}
```

Single linked list: Inserting new node in between-2

```
void insert_at_mid()
{
    int pos, i=1;
    struct node *newnode, *temp;
    int count= countnode(start);
    printf("Enter the position");
    scanf("%d", &pos);
    if (pos>count)
    {
        printf("Invalid position");
    }
    else
    {
        temp=start;
        while(i<pos)
        {
            temp=temp->next;
            i++;
        }
        newnode=getnode();
        printf("Enter data for newnode");
        scanf("%d", &newnode->data);
        newnode->next=temp->next;
        temp=newnode->next;
    }
}

int countnode(node *st)
{
    if(st == NULL)
        return 0;
    else
        return(1 + countnode(st -> next));
}
```

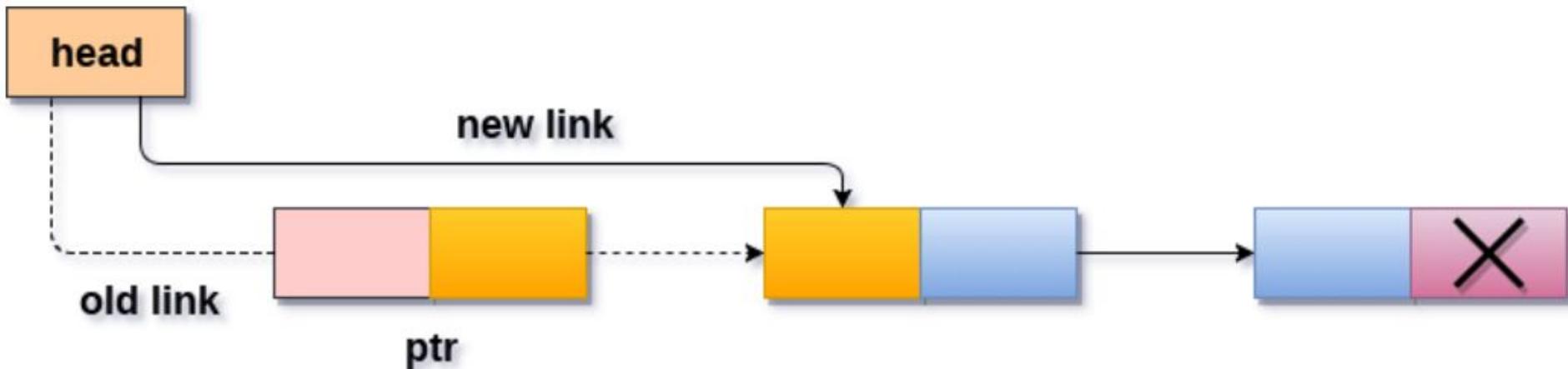
Single linked list: node deletion

- Another primitive operation that can be done in a singly linked list is the deletion of a node.
- Memory is to be released for the node to be deleted.
- A node can be deleted from the list from three different places namely.
 - Deleting a node at the beginning.
 - Deleting a node at the end.
 - Deleting a node at intermediate position.

Single linked list: node deletion at beginning

- The following steps are followed, to delete a node at the beginning of the list:
 - If the list is empty then display message “Empty list”.
 - If the list is not empty, follow the steps given below:

```
ptr= head;    head= ptr-> next;    free(ptr);
```



Single linked list: node deletion at beginning

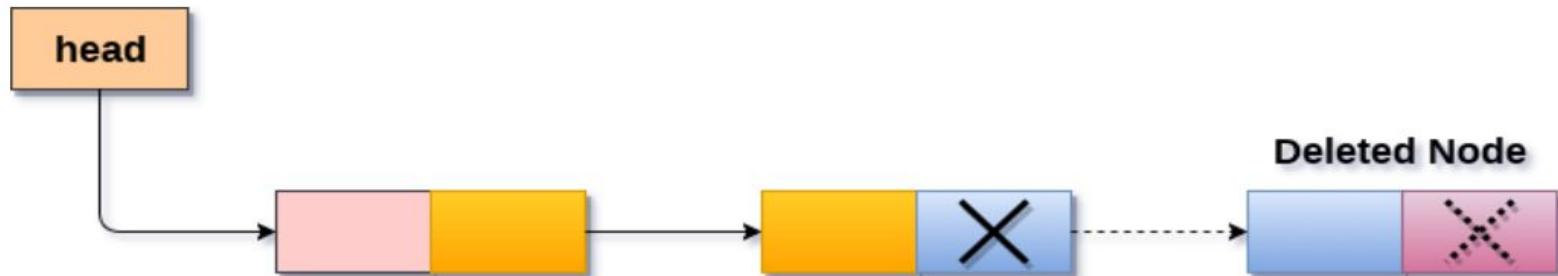
- The following function will delete node at the beginning:

```
void delete_at_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes are exist..");
        return ;
    }
    else
    {
        temp = start;
        start = temp -> next;
        free(temp);
        printf("\n Node deleted ");
    }
}
```

Single linked list: node deletion at the end

- The following steps are followed to delete a node at the end of the list:
 - If the list is not empty, follow the steps given below:

```
temp = prev = start;  
while(temp -> next != NULL)  
{  
    prev = temp;  
    temp = temp -> next;  
} prev -> next = NULL; free(temp);
```



Single linked list: node deletion at the end

- The following steps are followed to delete a node at the end of the list:

```
void delete_at_last()
{
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n Empty List..");
        return ;
    }
    else
    {
        temp = start;
        prev = start;
        while(temp -> next != NULL)
        {
            prev = temp;
            temp = temp -> next;
        }
        prev -> next = NULL;
        free(temp);
        printf("\n Node deleted ");
    }
}
```

Single linked list: node deletion in between

- The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).
- If the list is not empty, follow the steps given below.

```
if(pos > 1 && pos < nodectr)
{
    temp = prev = start;
    ctr = 1;
    while(ctr < pos)
    {
        prev = temp;
        temp = temp -> next;
        ctr++;
    }
    prev -> next = temp -> next;
    free(temp);
    printf("\n node deleted..");
}
```



Single linked list: node deletion in between

- The following steps are followed to delete a node after a specified position:

```
void delete_at_mid()
{
    int ctr = 1, pos, nodectr;
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n Empty List..");
        return ;
    }
    else
    {
        printf("\n Enter position of node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > nodectr)
        {
            printf("\nThis node doesnot exist");
        }
        if(pos > 1 && pos < nodectr)
        {
            temp = prev = start;
            while(ctr < pos)
            {
                prev = temp;
                temp = temp -> next;
                ctr++;
            }
            prev -> next = temp -> next;
            free(temp);
            printf("\n Node deleted..");
        }
        else
        {
            printf("\n Invalid position..");
            getch();
        }
    }
}
```

Single linked list: node traversing

```
void traverse()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right):
\n"); if(start == NULL )
        printf("\n Empty List");
    else
    {
        while (temp != NULL)
        {
            printf("%d ->", temp -> data);
            temp = temp -> next;
        }
    }
    printf("X");
}
```

Left to Right

```
void rev_traverse(node *st)
{
    if(st == NULL)
    {
        return;
    }
    else
    {
        rev_traverse(st -> next);
        printf("%d ->", st -> data)
    }
}
```

Right to Left

Single Linked List: Exercise

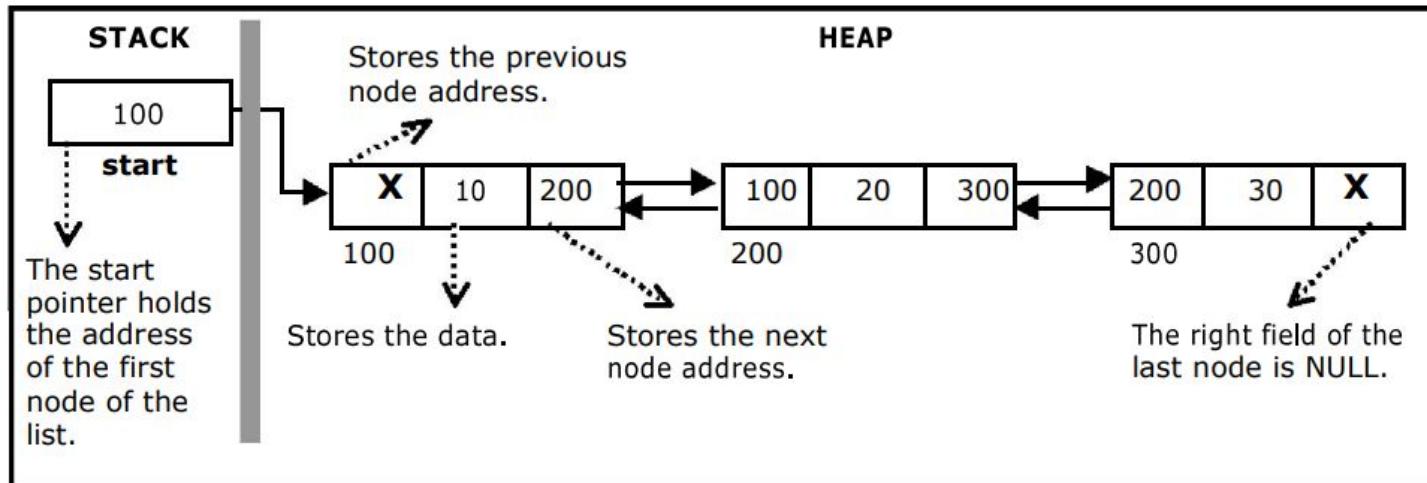
- WAP to perform following operations:
 - Linked list creation.
 - Insert at the beginning
 - Insert at end
 - Insert at mid
 - Delete at beginning
 - Delete at end
 - Delete at mid
 - Traverse left to right
 - Traverse right to left
 - Count node

Double linked list

- A double linked list is a two-way list in which all nodes will have two links.
- This helps in accessing both **successor** node and **predecessor** node from the given node position.
- It provides bi-directional traversing.
- Each node contains three fields:
 - Left link.
 - Data.
 - Right link.
- The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.
- Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

Double linked list: Structure and Operations

- The beginning of the double linked list is stored in a "start" pointer which points to the first node. The **first node left link** and **last node right link** is set to **NULL**.
- The basic operations in a double linked list are:
 - Creation, Insertion, Deletion, Traversing.

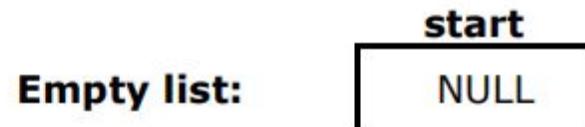
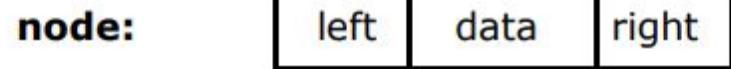


Double linked list: Structure implementation

- The following code gives the structure definition:

```
struct dlinklist
{
    struct dlinklist *left;
    int data;
    struct dlinklist *right;
};

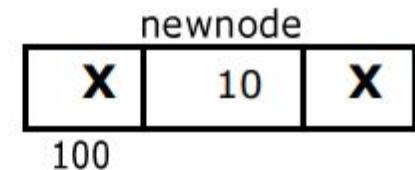
typedef struct dlinklist node;
node *start = NULL;
```



Double linked list: creating a node

- Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node.

```
node* getnode()
{
    node* newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> left = NULL;
    newnode -> right = NULL;
    return newnode;
}
```



Double linked list: creating ‘n’ nodes

- The following steps are to be followed to create ‘n’ number of nodes.
 - Get the new node using `getnode()`.

`newnode =getnode();`

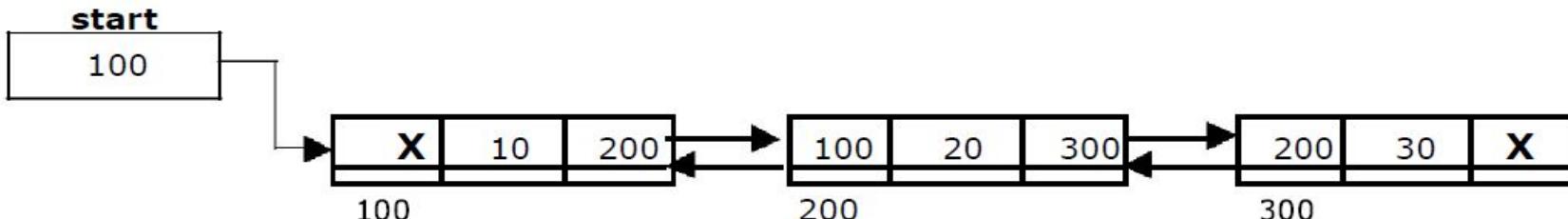
- If the list is empty then

`start = newnode;`
- If the list is not empty, follow the steps given below:
 - The `left field` of the `newnode` is made to point the `previous node`.
 - The `previous nodes right field` must be assigned with address of the `newnode`.
- Repeat the above steps ‘n’ times.

Double linked list: creating 'n' nodes

- The function `createlist()` is used to create 'n' number of nodes:
- Below we can see the double linked list with 3 nodes.

```
void createlist(int n)
{
    int i;
    node * new node;
    node *temp p;
    for(i = 0; i < n; i++)
    {
        new node = getnode();
        if(start == NULL)
        {
            start = new node;
        }
        else
        {
            temp = start;
            while(temp -> right)
                temp = temp -> right;
            temp -> right = new no de;
            new node -> left = temp;
        }
    }
}
```



Double linked list: inserting node in the beginning

- The following steps are to be followed to insert a new node in the beginning.
 - Get the new node using `getnode()`

`newnode=getnode();`

- If the list is empty then

`start = newnode;`

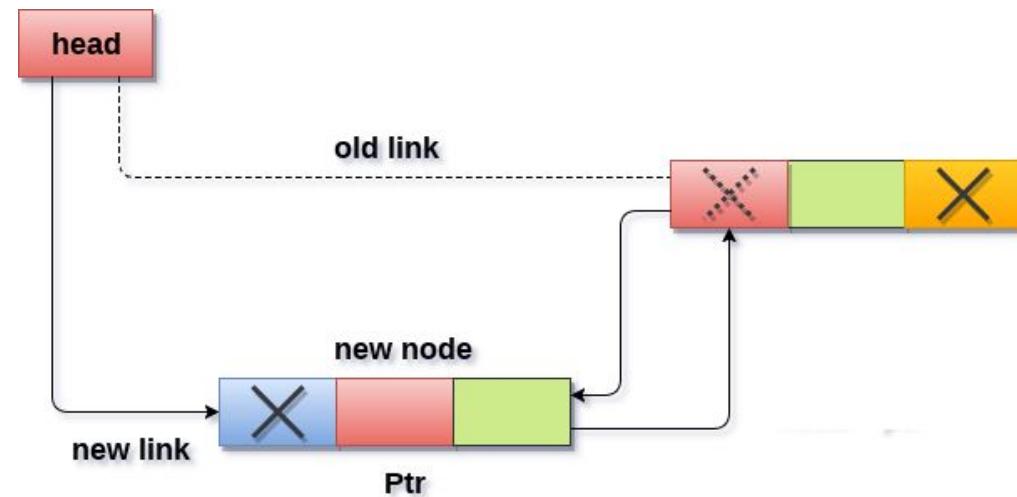
- If the list is not empty, follow the steps given below:

`newnode -> right = start;`

`start -> left = newnode;`

`start = newnode;`

Double linked list: inserting node in the beginning



`newnode -> right = start;`

`start -> left = newnode;`

`start = newnode;`

Double linked list: inserting node at the end

- The following steps are to be followed to insert a new node at the end.
 - Get the new node using `getnode()` as `newnode=getnode();`
 - If the list is empty then

`start = newnode;`

- If the list is not empty, follow the steps given below:

`temp=start;`

`while(temp->right != NULL)`

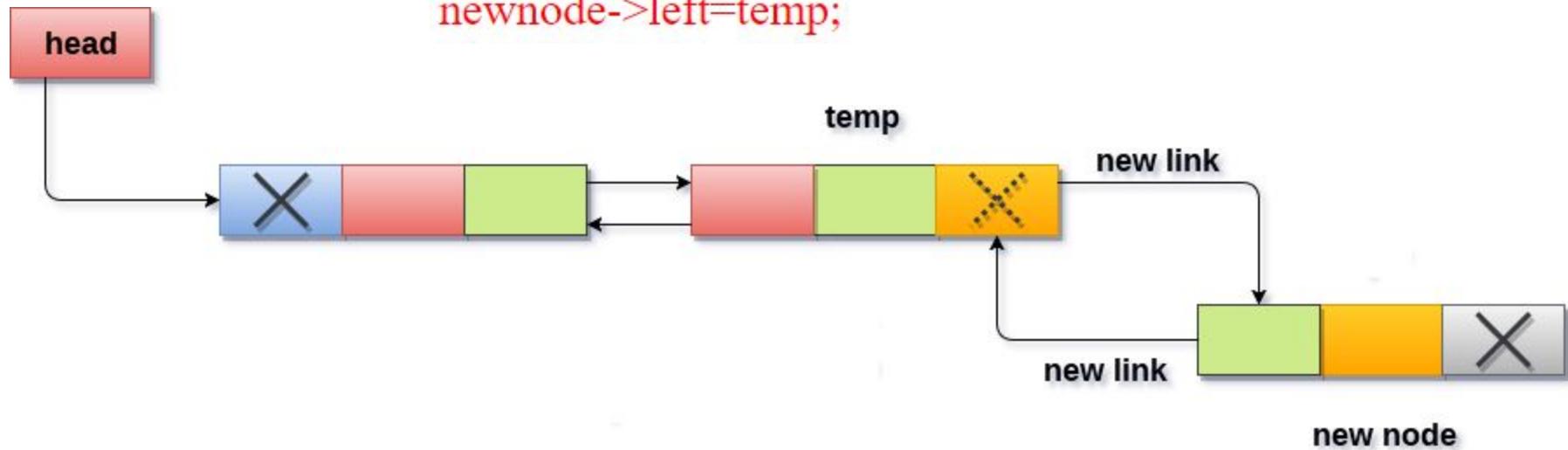
`temp=temp -> right;`

`temp->right= newnode;`

`newnode->left=temp;`

Double linked list: inserting node at the end

```
temp=start;  
while(temp->right != NULL)  
    temp=temp -> right;  
temp->right= newnode;  
newnode->left=temp;
```

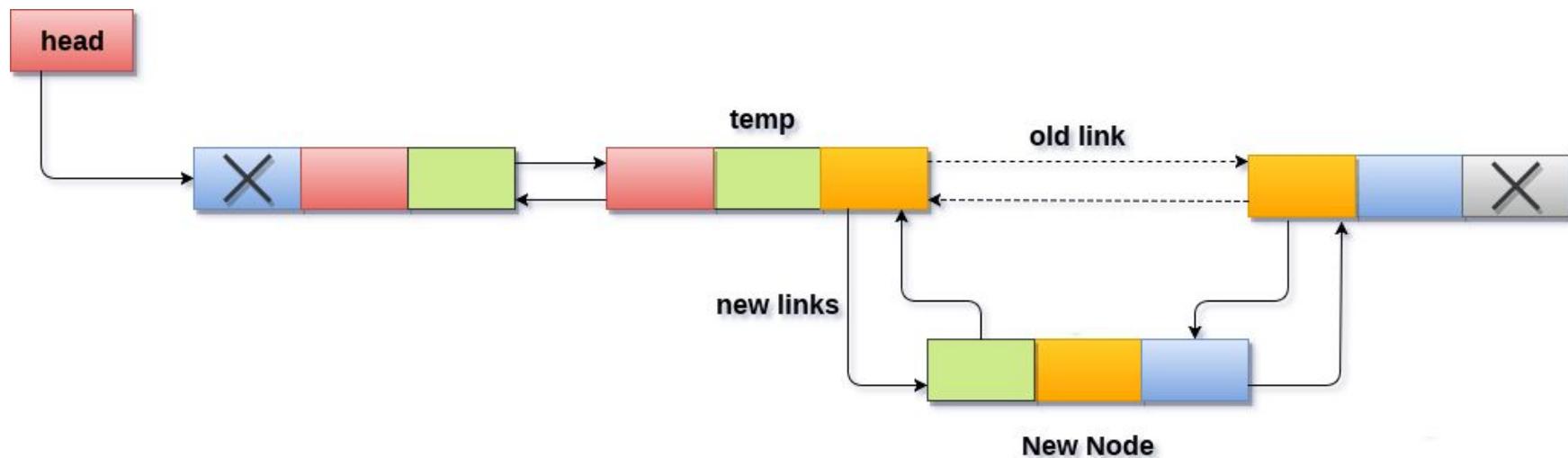


Double linked list: inserting node in the mid

- The following steps are to be followed to insert a new node at the end.
 - Get the new node using `getnode()` as `newnode=getnode();`
 - If the list is empty then
 - `start = newnode;`
 - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
 - This is done by `countnode()` function.
 - Store the starting address (which is in start pointer) in `temp` and `prev` pointers.
 - Then traverse the `temp` pointer upto the specified position followed by `prev` pointer.
 - After reaching the specified position, follow the steps:

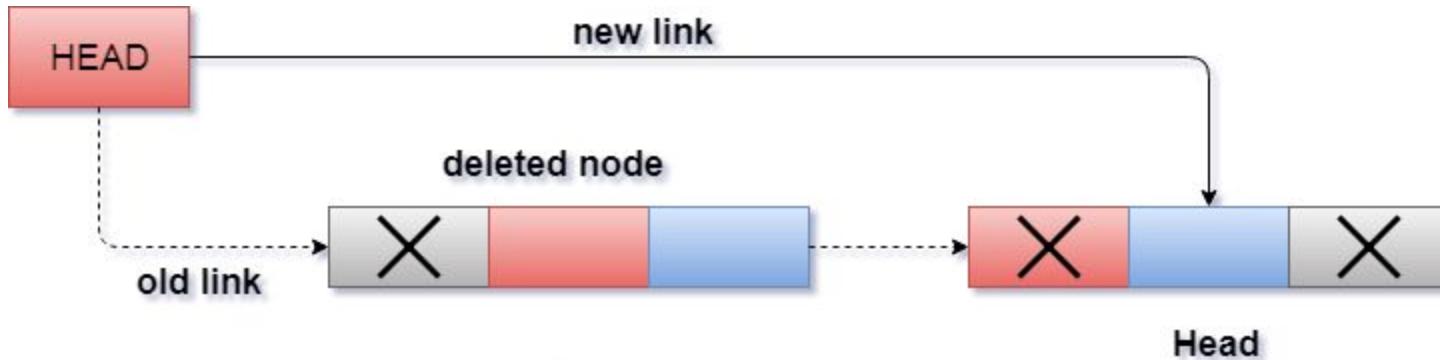
Double linked list: inserting node in the mid

```
newnode -> left = temp;  
newnode-> right = temp -> right;  
temp->right->left = newnode;  
temp->right = newnode;
```



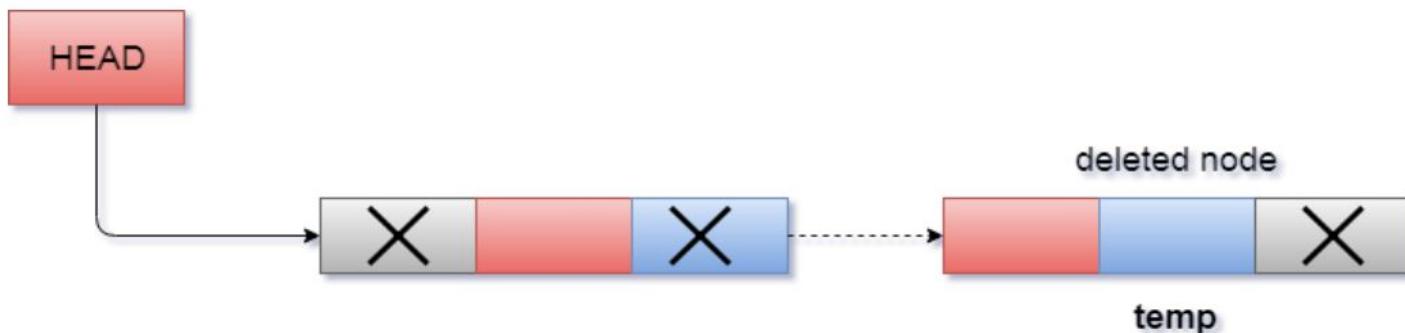
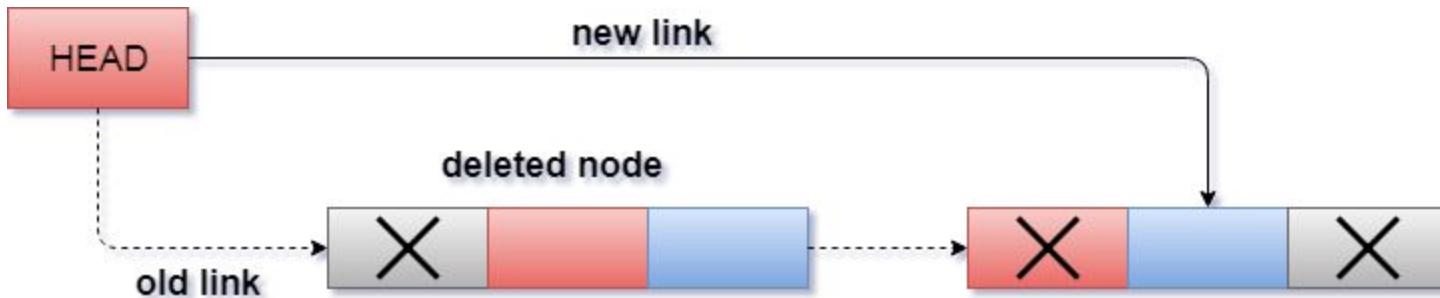
Double linked list: delete a node at the start

```
head=start;  
start=start->right;  
free(head);
```



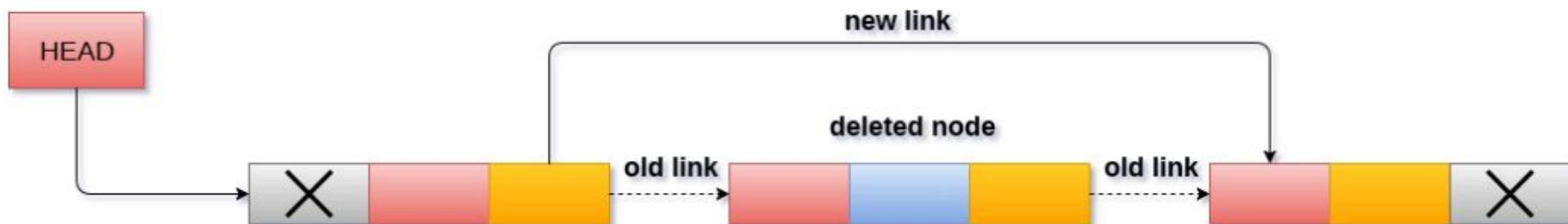
at the end

```
head= start;  
while(temp->right!=NULL)  
    temp=temp->right;  
temp->left->right =NULL;  
free(temp);
```



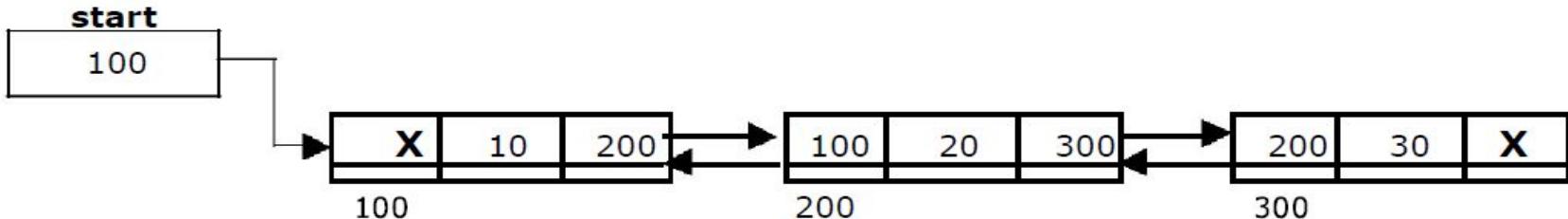
a node in between

```
if(pos > 1 && pos < nodectr)
{
    temp = start;
    i = 1;
    while(i < pos)
    {
        temp = temp -> right;
        i++;
    }
    temp -> right -> left = temp -> left;
    temp -> left -> right = temp -> right;
    free(temp);
    printf("\n node deleted..");
}
```



Double linked list: traversing

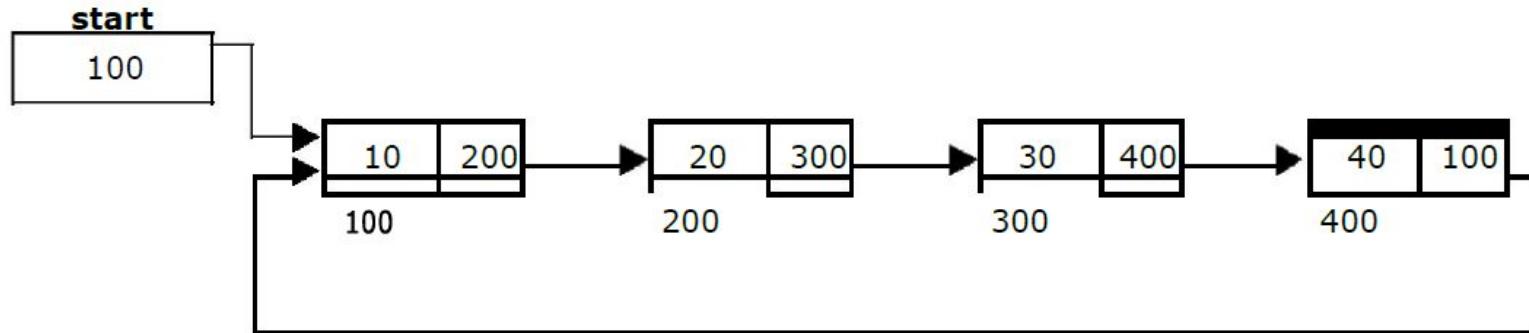
```
temp= start;  
while(temp!=NULL)  
{  
    printf("%d",temp->data);  
    temp=temp->right;  
}  
Left to Right traversing
```



```
temp= start;  
while(temp->right!=NULL)  
{  
    temp=temp->right;  
}  
while(temp!=NULL)  
{  
    printf("%d", temp->data);  
    temp=temp->left;  
}  
Right to Left traversing
```

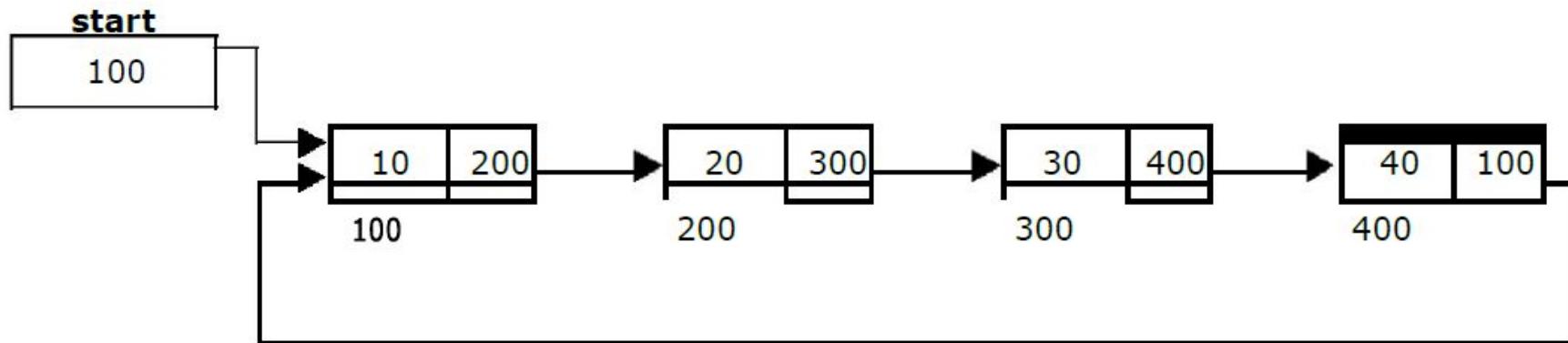
Circular Single linked list

- It is just a single linked list in which the link field of the last node points back to the address of the first node.
- A circular linked list has no beginning and no end.
- It is necessary to establish a special pointer called start pointer always pointing to the first node of the list.
- In circular linked list no **null** pointers are used, hence all pointers contain valid address.



Circular Single linked list: Basic operations

- Creation
- Insertion
- Deletion
- Traversing



Circular Single linked list: creation with ‘n’ nodes

- The following steps are to be followed to create ‘n’ number of nodes:
 - Get the new node using `getnode()` as `newnode = getnode();`
 - If the list is empty, assign new node as start;

`start = newnode;`

- If the list is not empty, follow the steps given below:

`temp=start;`

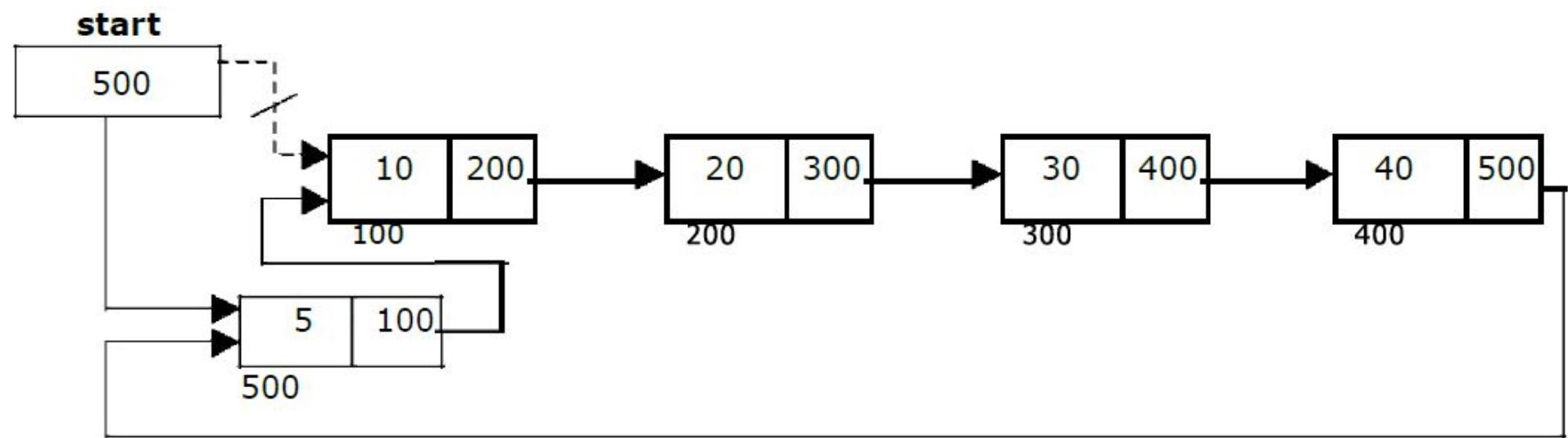
`while(temp->next!=NULL)`

`temp=temp->next;`

`temp->next=newnode;`

- Repeat the above steps ‘n’ times.
- At last set the following: `newnode->next= start;`

Circular Single linked list: Insert node at the beginning



Circular Single linked list: Insert node at the beginning

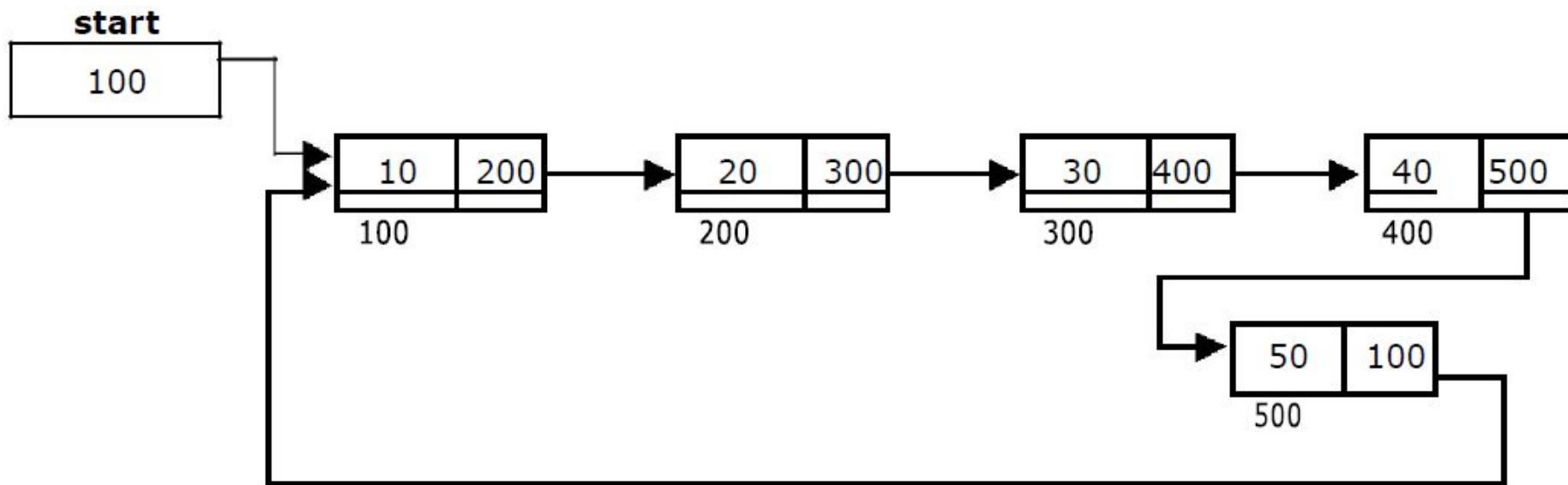
- The following steps are to be followed to insert node at beginning:
 - Get the new node using `getnode()` as `newnode = getnode();`
 - If the list is empty, assign new node as start;

```
start = newnode;  
newnode->next=start;
```

- If the list is not empty, follow the steps given below:

```
temp=start;  
while(temp->next!=start)  
    temp=temp->next;  
newnode->next=start;  
start=newnode;  
temp->next=start
```

Circular Single linked list: Insert node at the last



Circular Single linked list: Insert node at the last

- The following steps are to be followed to insert node at beginning:
 - Get the new node using `getnode()` as `newnode = getnode();`
 - If the list is empty, assign new node as start;

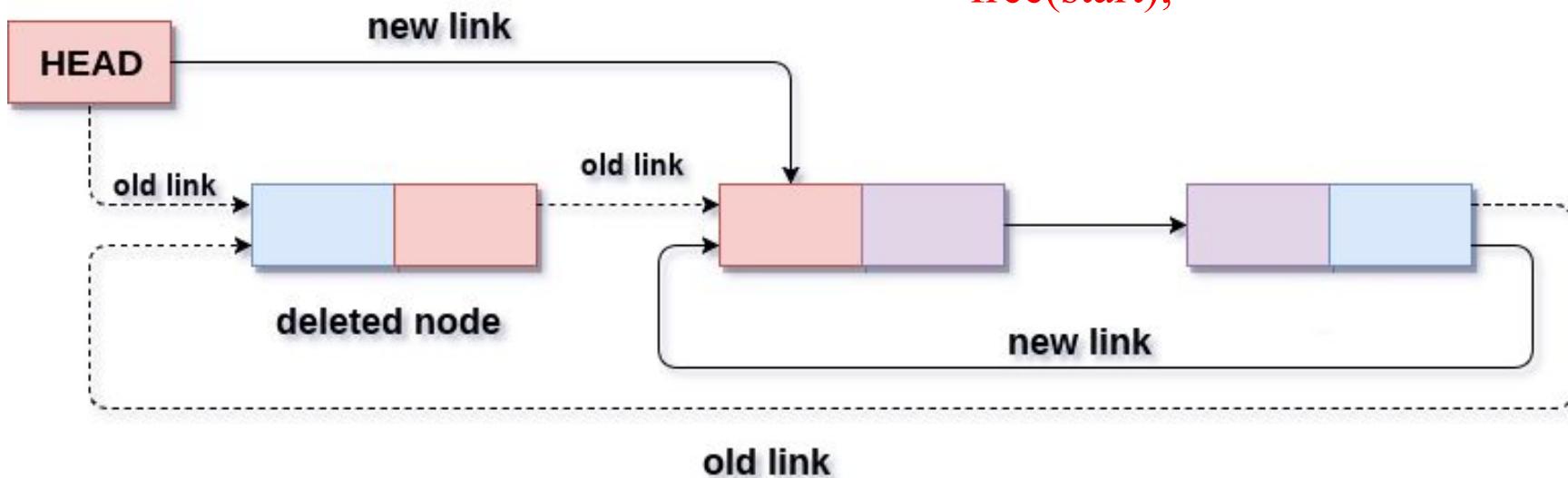
```
start = newnode;  
newnode->next=start;
```

- If the list is not empty, follow the steps given below:

```
temp=start;  
while(temp->next!=start)  
    temp=temp->next;  
newnode->next=start;  
temp->next=newnode;
```

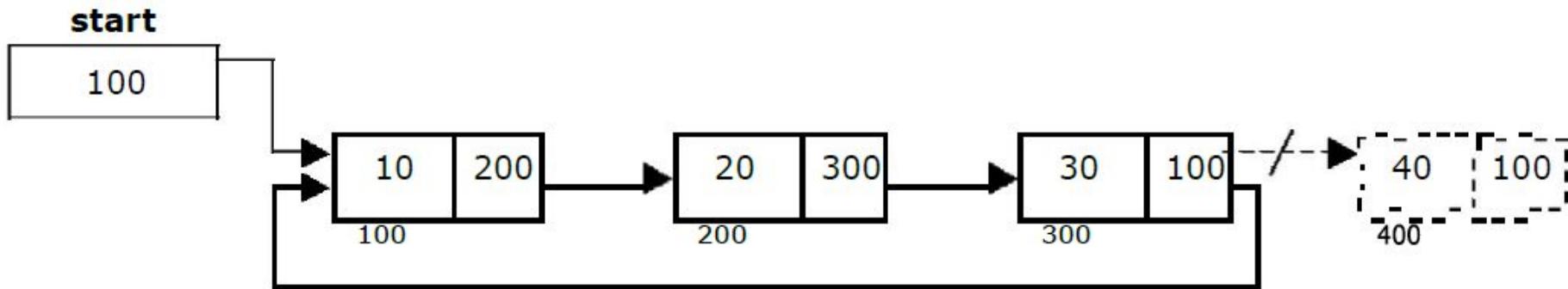
delete node at the beginning

```
temp= start=head;  
while(temp->next!=start)  
{  
    temp=temp->right;  
}  
head=head->next;  
temp->next=head;  
free(start);
```



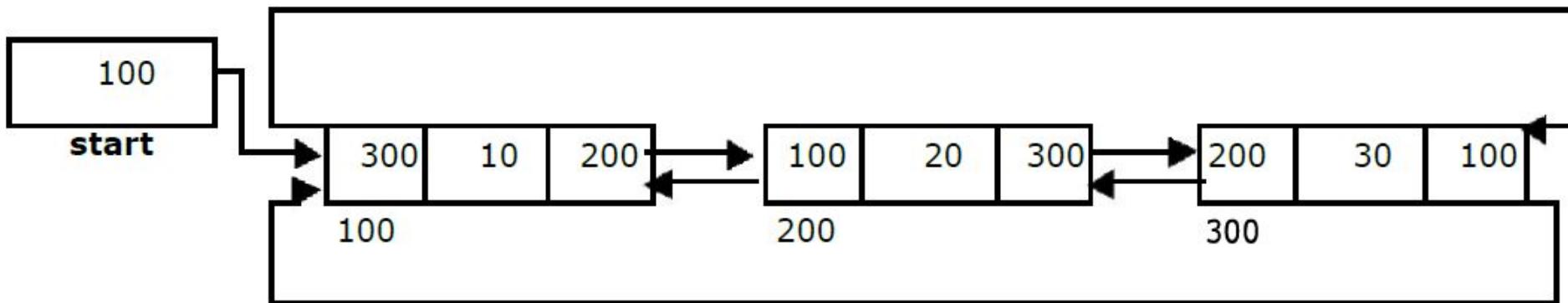
delete node at the end

```
temp= prev=start;  
while(temp->next!=start)  
{  
    prev=temp;  
    temp=temp->right;  
}  
prev->next=start;  
free(temp);
```



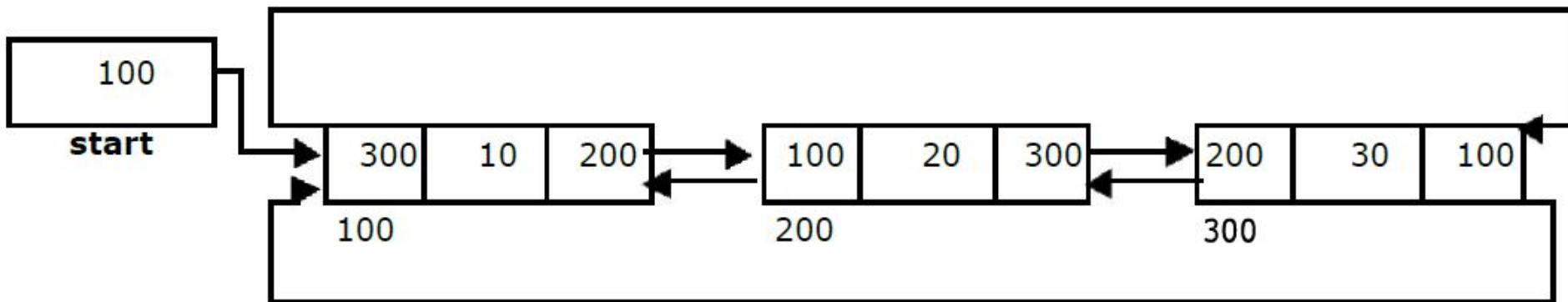
Circular Double linked list

- A circular double linked list has both successor pointer and predecessor pointer in circular manner.
- The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list.
- In circular double linked list the right link of the right most node points back to the start node and left link of the first node points to the last node.



Circular Double linked list: Operations

- Creation
- Insertion
- Deletion
- Traversing



Circular Doubly linked list: creating ‘n’ nodes

- The following steps are to be followed to create ‘**n**’ number of nodes:
- Get the new node using `getnode()` as `newnode = getnode();`
- If the list is empty, then do the following

```
start = newnode;  
newnode -> left = start;  
newnode ->right = start;
```

- If the list is not empty, follow the steps given below:

```
newnode -> left = start -> left;  
newnode -> right = start;  
start -> left->right = newnode;  
start -> left = newnode;
```

- Repeat the above steps ‘**n**’ times.

Circular Double linked list: insert a node at beginning

Get the new node using getnode()

```
newnode=getnode();
```

If the list is empty, then

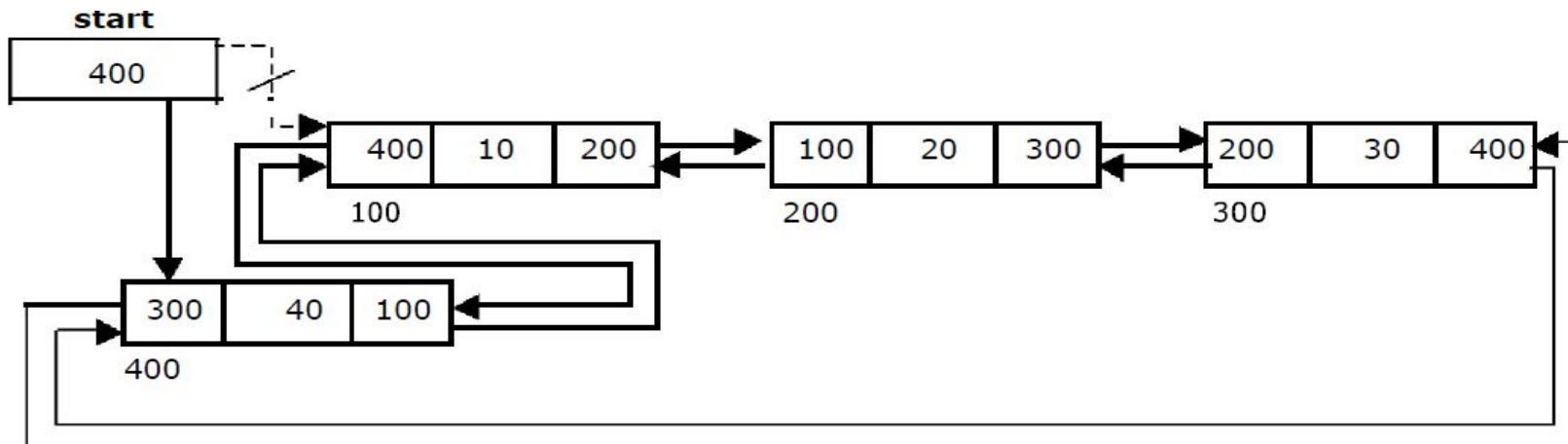
```
start = newnode;
```

```
newnode -> left = start;
```

```
newnode -> right = start;
```

If the list is not empty, follow the steps given below:

```
newnode -> left = start -> left;  
newnode -> right = start; start -  
> left -> right = newnode; start  
> left = newnode;  
start = newnode;
```



Circular Double linked list: insert a node at end

Get the new node using getnode()

```
newnode=getnode();
```

If the list is empty, then

```
start = newnode;
```

```
newnode -> left = start;
```

```
newnode -> right = start;
```

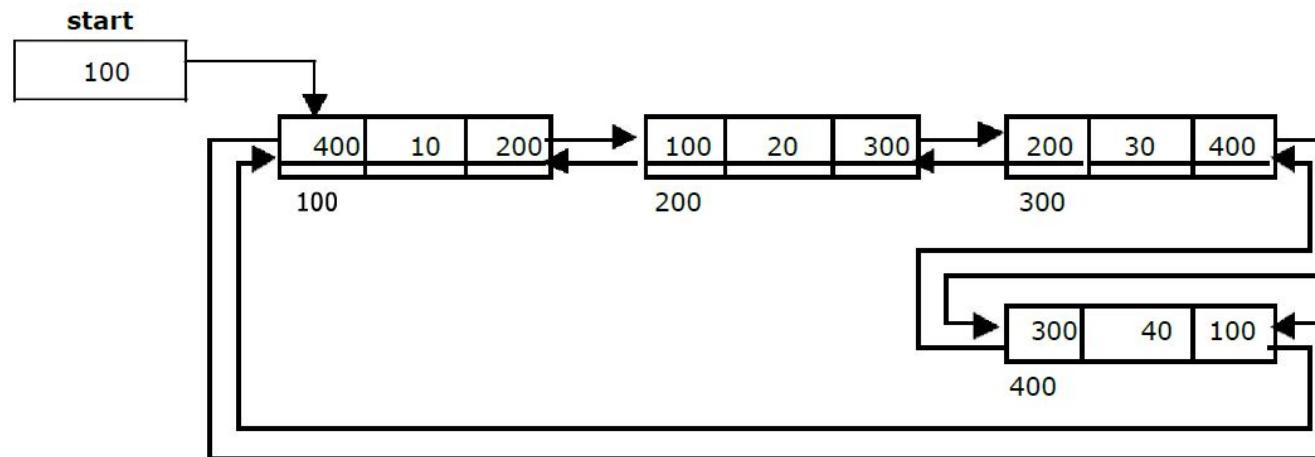
If the list is not empty follow the steps given below:

```
newnode -> left = start -> left;
```

```
newnode -> right = start; start -
```

```
> left -> right = newnode; start
```

```
-> left = newnode;
```



Circular Double linked list: insert a node in between

After reaching the specified position, follow the steps given below:

newnode -> left = temp;

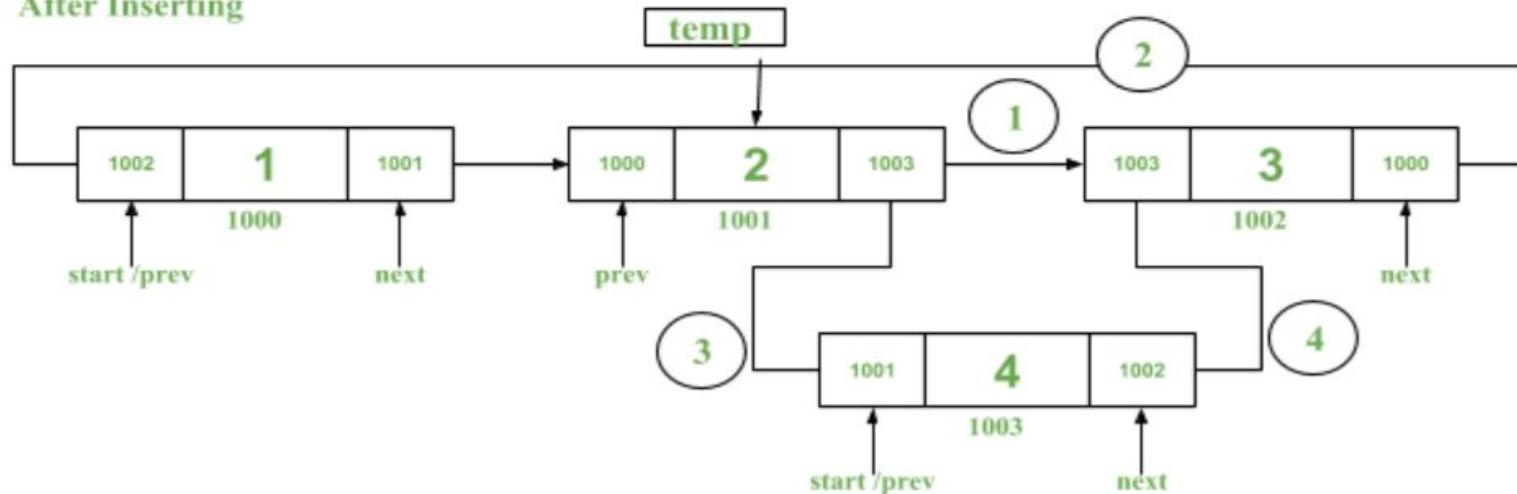
newnode -> right = temp -> right;

temp -> right -> left = newnode;

temp -> right = newnode;

nodectr++;

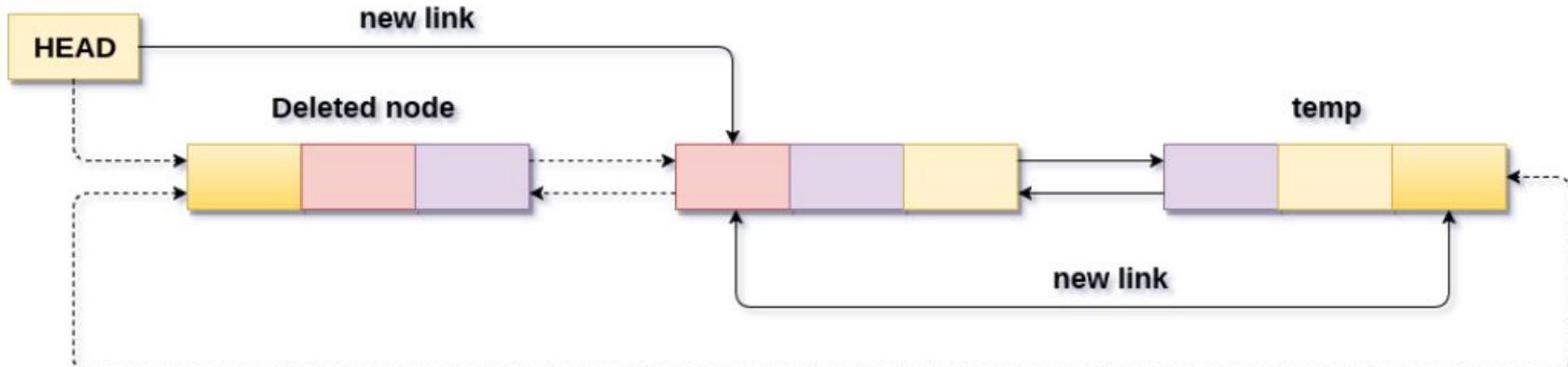
After Inserting



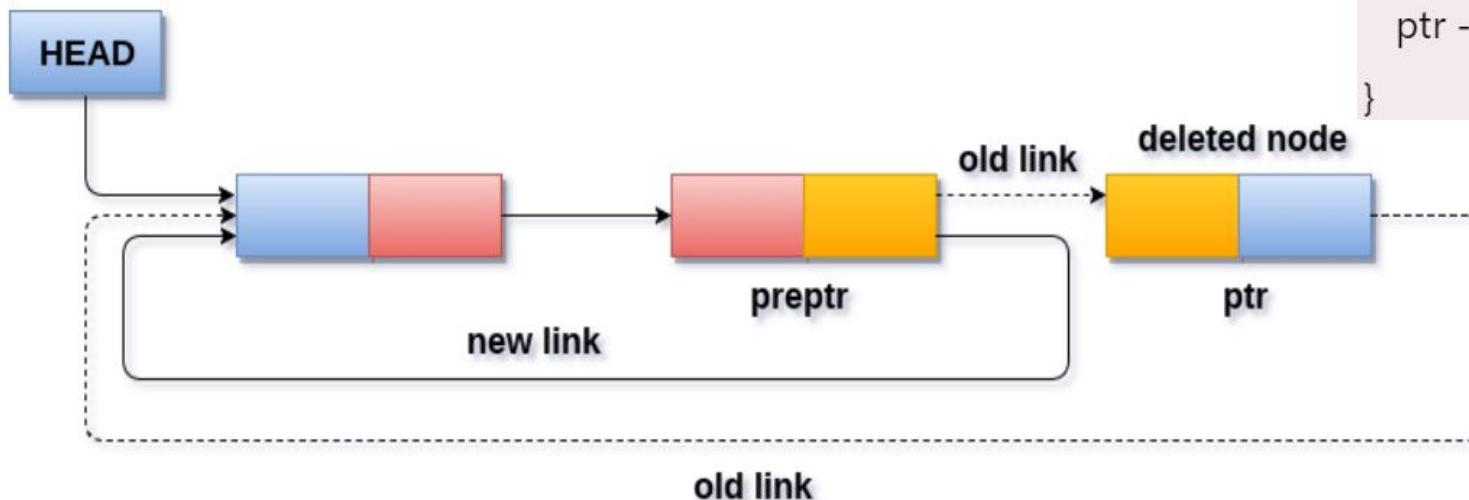
Circular Double linked list: delete a node in start

If the list is not empty, follow the steps given below:

```
temp = start;  
start = start -> right;  
temp -> left -> right = start;  
start -> left = temp -> left;
```

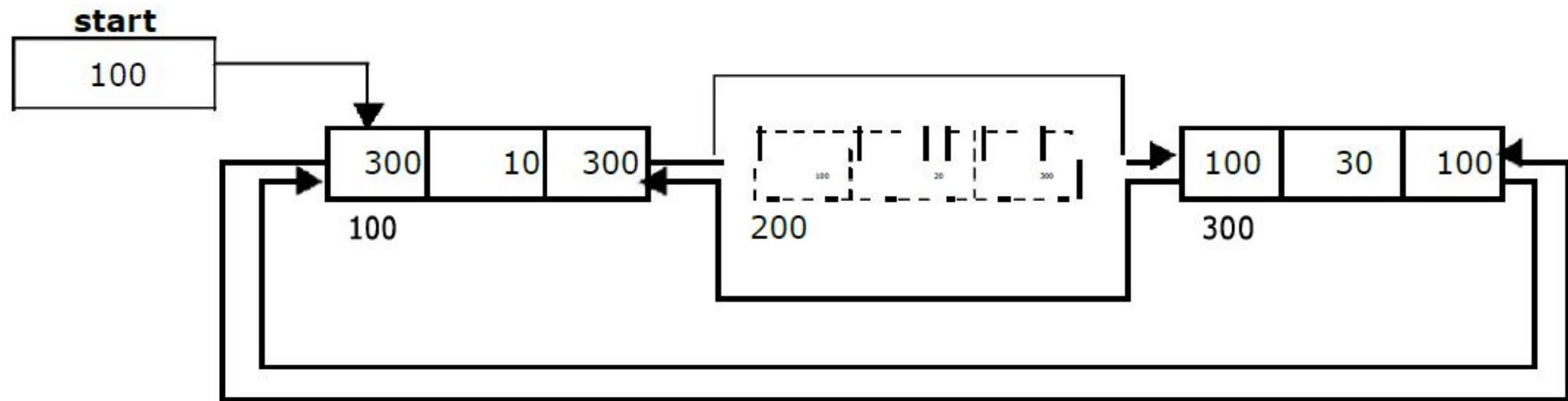


Circular Double linked list: delete a node in end



```
while(temp->next != head)
{
    temp = temp->next;
}
temp->next = ptr;
ptr->prev=temp;
head->prev = ptr;
ptr->next = head;
}
```

Circular Double linked list: delete a node after a specified position



Circular Doubly linked list: delete a node after a specified position

```
if(pos > 1 && pos < nodectr)
{
    temp = start;
    i = 1;
    while(i < pos)
    {
        temp = temp -> right ;
        i++;
    }
    temp -> right -> left = temp -> left;
    temp -> left -> right = temp -> right;
    free(temp);
    printf("\n node deleted..");
    nodectr--;
}
```

Linked List complexity

Operation	SLL (Singly Linked List)	DLL (Doubly Linked List)	CSLL (Circular Singly Linked List)	CDLL (Circular Doubly Linked List)
Insertion at Head	O(1)	O(1)	O(1)	O(1)
Insertion at Tail	O(n)	O(n) (unless tail pointer is maintained: O(1))	O(n) (unless tail pointer is maintained: O(1))	O(n) (unless tail pointer is maintained: O(1))
Insertion at Middle (after given node)	O(1)	O(1)	O(1)	O(1)
Insertion at Specific Position (by index)	O(n)	O(n)	O(n)	O(n)
Deletion at Head	O(1)	O(1)	O(1)	O(1)
Deletion at Tail	O(n)	O(1)	O(n)	O(1)
Deletion from Middle (by reference to a node)	O(1)	O(1)	O(1)	O(1)
Deletion at Specific Position (by index)	O(n)	O(n)	O(n)	O(n)
Search (by value)	O(n)	O(n)	O(n)	O(n)
Traversal (Forward)	O(n)	O(n)	O(n)	O(n)
Traversal (Backward)	Not possible	O(n)	Not possible	O(n)
Reversing the List	O(n)	O(n)	O(n)	O(n)

Stacks and Queues

Stacks and Queues

- The linear lists and linear arrays discussed in the previous unit allowed us to insert and delete elements at any place in the list:
 - at the beginning
 - at the end
 - in the middle
- But, there are several situations in which we want to restrict the insertion and deletion so that they can take place:
 - only at the beginning
 - or, at the end
 - but, not in the middle
- Stacks and Queues are useful in such scenarios.

Stack definition

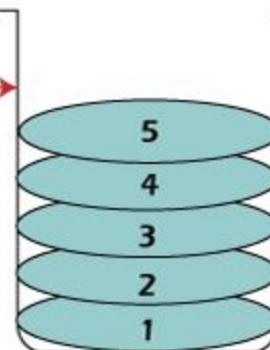
- A stack is a linear structure in which items may be added or removed only at one end.
- In stack, last item to be added is the first item to be removed.
 - Hence, it is called as **Last-In-First-Out (LIFO)** list.

TOP →



Stack of Coins

TOP →



Stack of Plates

TOP →



Can of Tennis Balls



Stack of Books

Queue definition

- A stack is a linear structure in which items may be added only at one end and items may be removed only at other end.
- In queue, first item to be added is the first item to be removed.
 - Hence, it is called as **First-In-First-Out (FIFO)** list.

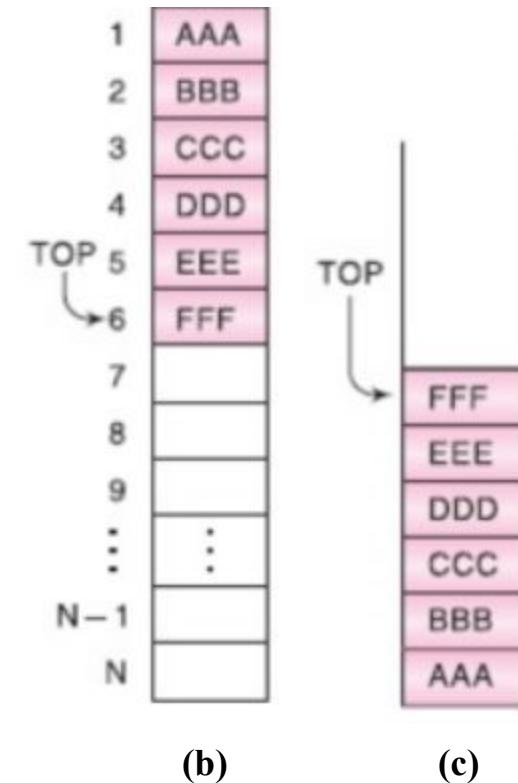
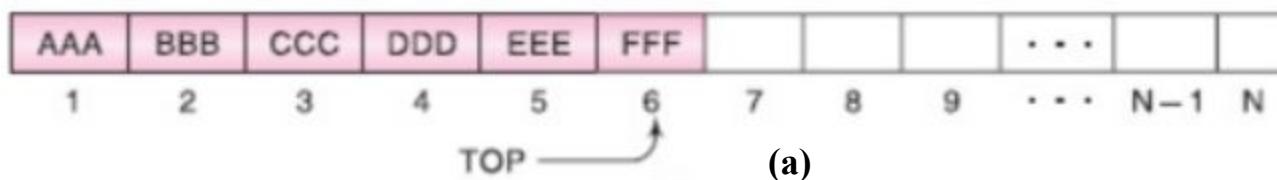


Stack Data Structure

- A stack of elements in which an element may be inserted or deleted only at one end, called the **top** of the stack.
- This means that elements are removed from the stack in reverse order.
- **Basic operations** associated with stacks are:
 - **Push:** is the term used to insert an item into a stack
 - **Pop:** is the term used to remove an item from the stack
 - **IsEmpty:** Check if the stack is empty
 - **IsFull:** Check if the stack is full
 - **Peek:** Get the value of peak element without removing it.
- The terms “**Push and Pop**” are used only with stacks and not with any other data structures.

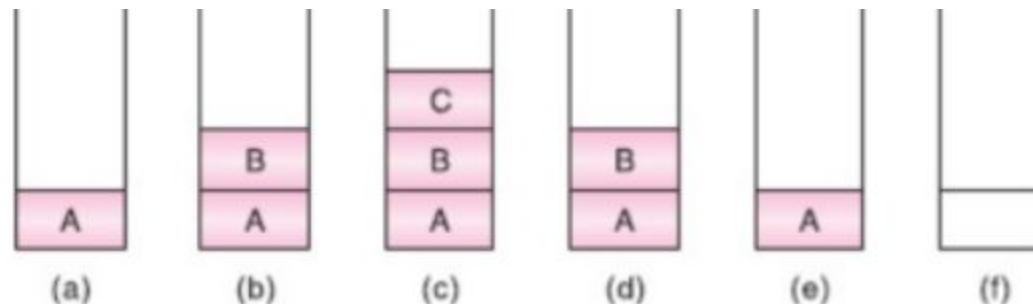
Stack example

- Consider we have a stack as:
STACK: AAA, BBB, CCC, DDD, EEE, FFF
- The different ways of picturing the stack as follows:
 - Regardless of the way a stack is described, its underlying property is that insertion and deletion can occur only at the TOP.
 - For example:** EEE can not be deleted before FFF



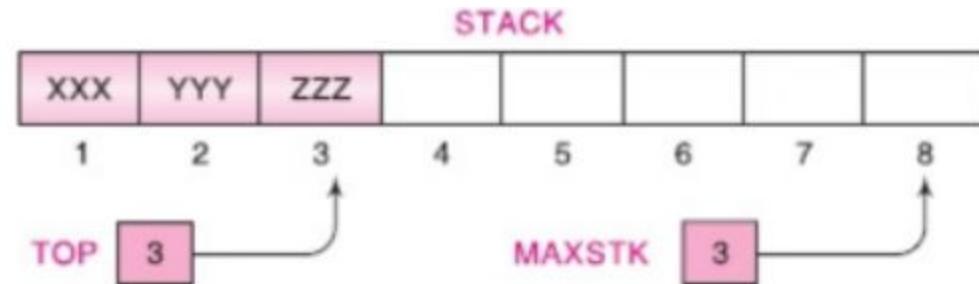
Stack to postponed decision

- Suppose you have written a C program in which we have started the execution of program with main() function (in figure main() is shown as A). But, in between we encountered the other functions in the manner as shown in figure below.
- Observe that, at each step (inserting or deleting functions from stack) of the processing in the figure, stack automatically maintains the order that is required to complete the program.



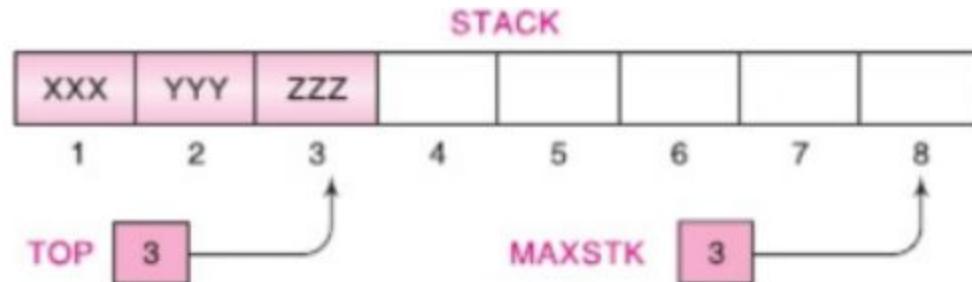
Array representation of Stack

- Stacks can be represented using a linear array.
 - First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack.
 - Then, starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion.
- **TOP** is a **pointer** which contains the location of top element of the stack.
- **MAXSTK** is a **variable** which gives the maximum number of element that can be held in stack.



Array representation of Stack Contd..

- The condition **TOP =0 or TOP= NULL** will indicate the stack is empty.
- In the figure below, **TOP=3**, that means there are **three elements in the stack**.
- Since, **MAXSTK** is **8**, there is a **room for 5 more items** in the stack.
- To perform the **PUSH or POP** operation on the stack we always check conditions Overflow and Underflow respectively.
 - **Overflow:** checks whether there is a room for new item in the stack.
 - **Underflow:** checks whether there is an element in the stack to be deleted.



Working of Stack Data Structure

- **The operations of stack work as follows:**
 - A pointer called **TOP** is used to keep track of the top element in the stack.
 - When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing **TOP == -1**.
 - On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
 - On popping an element, we return the element pointed to by TOP and reduce its value.
 - Before pushing, we check if the stack is already full.
 - Before popping, we check if the stack is already empty.

Working of Stack Data Structure

TOP = -1



empty stack

TOP = 0
stack[0] = 1



push

TOP = 1
stack[1] = 2



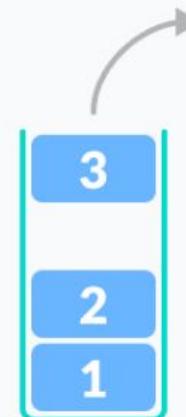
push

TOP = 2
stack[2] = 3



push

TOP = 1
return stack[2]



pop

Stack operation: PUSH operation algorithm

PUSH(STACK, TOP, MAXSTK, ITEM)

This procedure pushes an ITEM onto a stack.

1. [Stack already filled?]

If TOP = MAXSTK, then: Print: OVERFLOW, and Return.

2. Set TOP := TOP + 1. [Increases TOP by 1.]

3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]

4. Return.

Stack operation: POP operation algorithm

POP(STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

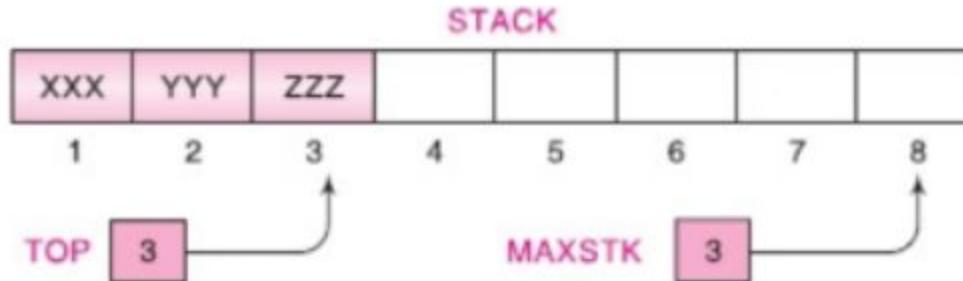
1. [Stack has an item to be removed?]
If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP := TOP – 1. [Decreases TOP by 1.]
4. Return.

Stack operation: PUSH operation simulation

- Consider the stack below:

We simulate the operation **PUSH (STACK, WWW)**

1. Since $\text{TOP} = 3$, control is transferred to Step 2.
2. $\text{TOP} = 3 + 1 = 4$.
3. $\text{STACK}[\text{TOP}] = \text{STACK}[4] = \text{WWW}$.
4. Return.

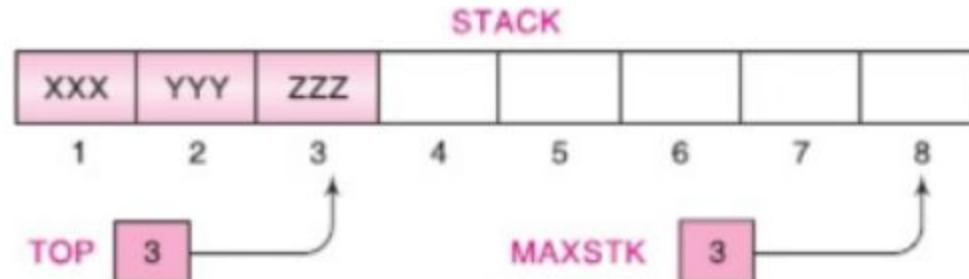


Stack operation: POP operation simulation

- Consider the stack below:

We simulate the operation **POP (STACK, ZZZ)**

1. Since $\text{TOP} = 3$, control is transferred to Step 2.
2. $\text{ITEM} = \text{ZZZ}$.
3. $\text{TOP} = 3 - 1 = 2$.
4. Return.



Stack implementation using Array in C

```
#include<stdio.h>
#include<stdlib.h>

int n, top = -1, *stack;

void push(int x){
    if(top==n) return;
    stack[++top]=x;
}

int pop(){
    if(top==-1) return -1;
    return stack[top--];
}

int peek(){
    if(top==-1) return -1;
    return stack[top];
}

void display(){
    for(int i=top ; i>-1 ; i--)
        printf("%d ",stack[i]);
    printf("\n\n");
}

int main(){
    n = 10;
    printf("Initializing the stack with size 10\n\n");
    stack = (int*)malloc(n*sizeof(int));

    printf("Pushing elements into the stack\n1\n2\n3\n\n");
    push(1);
    push(2);
    push(3);

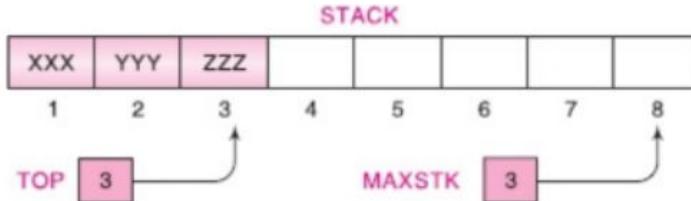
    printf("Displaying elements of the stack -\n");
    display();

    printf("The top of the stack = %d\n\n",peek());
    printf("Pop the top of the stack = %d\n\n",pop());
    printf("Pop the top of the stack = %d\n\n",pop());
    printf("Displaying elements of the stack -\n");
    display();

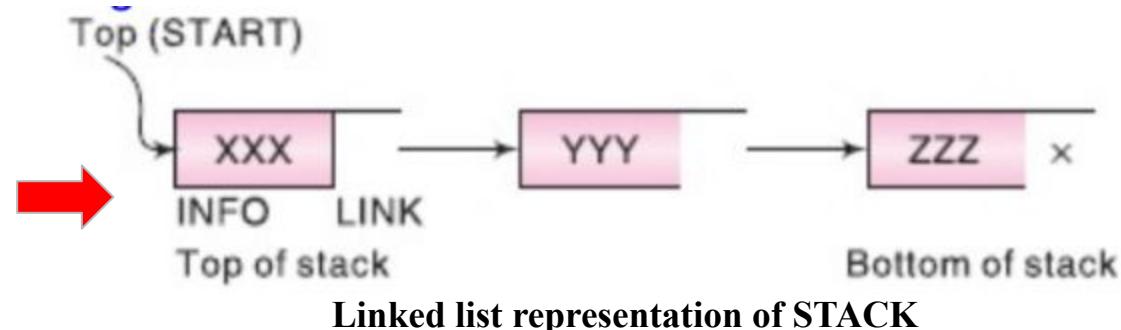
    return 0;
}
```

Linked List representation of stack

- We will discuss the linked representation of **STACK** using **Single Linked List**.
- The **INFO** fields of the nodes hold the elements of the stack.
- The **LINK** fields hold the pointers to the neighbouring elements in the stack.
- The **START** pointer behave as a **TOP** pointer of the stack.
- The **NONE** pointer of the last node in the list signals the bottom of the stack.



Array representation of STACK



Linked list representation of STACK

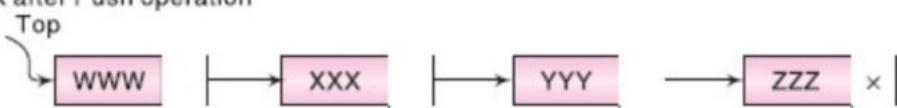
Linked List representation of stack: PUSH and POP

- A **PUSH** operation into a **STACK** is accomplished by inserting a node into the front or start of the list.
- Similarly, a **POP** operation is undertaken by deleting the node pointed by **START** pointer.

Push 'WWW' into STACK
STACK before Push operation:

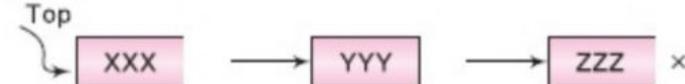


STACK after Push operation

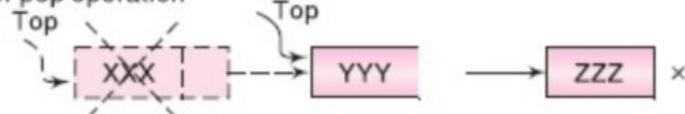


PUSH operation in STACK

Pop from STACK
STACK before pop operation:



STACK after pop operation



POP operation in STACK

Linked List representation of stack

- The array representation of **STACK** required to maintain two variables **TOP** and **MAXSTK** to check the **Overflow** (**TOP=MAXSTK**) and **Underflow** (**TOP=0**) before **PUSH** and **POP** operations respectively.
- In linked list representation, we no longer use maintain these variables.
- There is no limit on the capacity of the linked stack hence it can support as many **PUSH** operations as we want, provided memory is available.
-

Linked List STACK operations

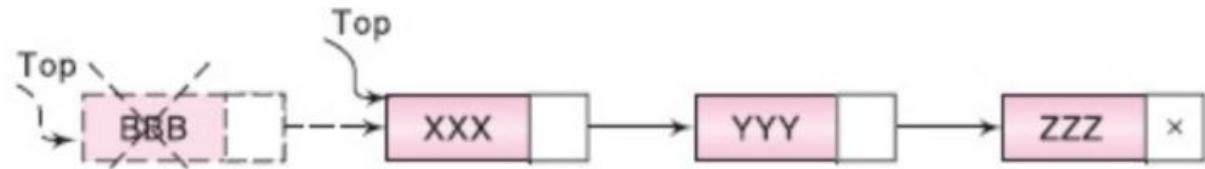
Suppose the initial stack is:



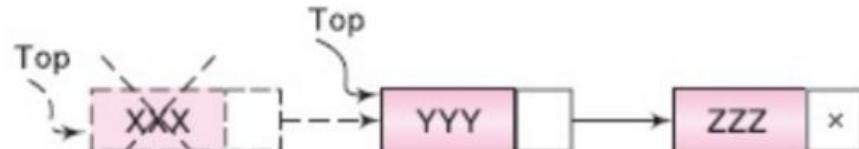
Push BBB into stack:



Pop BBB from stack:



Pop XXX from stack:



Linked list implementation of STACK in C

Creating a node structure:

```
#include <stdio.h>
#include <stdlib.h>

// Structure to create a node with data and next pointer
struct Node {
    int data;
    struct Node *next;
};
Node* top = NULL;
```

Linked list implementation of STACK in C

Push operation function:

```
// Push() operation on a stack
void push(int value) {
    struct Node *newNode;
    newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value; // assign value to the node
    if (top == NULL) {
        newNode->next = NULL;
    } else {
        newNode->next = top; // Make the node as top
    }
    top = newNode; // top always points to the newly created node
    printf("Node is Inserted\n\n");
}
```

Linked list implementation of STACK in C

Display the STACK element:

```
void display() {
    // Display the elements of the stack
    if (top == NULL) {
        printf("\nStack Underflow\n");
    } else {
        printf("The stack is \n");
        struct Node *temp = top;
        while (temp->next != NULL) {
            printf("%d--->", temp->data);
            temp = temp->next;
        }
        printf("%d--->NULL\n\n", temp->data);
    }
}
```

Linked list implementation of STACK in C

Main function:

```
int main() {
    int choice, value;
    printf("\nImplementation of Stack using Linked List\n");
    while (1) {
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("\nEnter your choice : ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\nEnter the value to insert: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
                printf("Popped element is :%d\n", pop());
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
                break;
            default:
                printf("\nWrong Choice\n");
        }
    }
}
```

Applications of Stack: Polish Notation

- One application of stack is in evaluation of arithmetic expression.
- An arithmetic expression consists of operands and operators.

$$X = A / B + C * D - F * G / Q$$

- A **polish** mathematician suggested a notation called **Polish notation**, which gives two alternatives to represent an arithmetic expression. **The notations are prefix and postfix notations.**
- The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.
- Hence parenthesis are not required while writing expressions in Polish notation.

Polish Notation: Prefix and Postfix

- **Prefix notation:**
 - The prefix notation is a notation in which the operator is written before the operands. **For example, + AB.**
- **Postfix notation:**
 - The postfix notation is a notation in which the operator is written after the operands. **For example, AB +**
- **Infix notation:**
 - The infix notation is what we come across in our general mathematics, where the operator is written in-between the operands. **For example, A+B.**

Polish Notation: Problem with infix notation

- Suppose we to evaluate the following arithmetic equation written in infix notation:

$$2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$$

- First we evaluate the exponentiations to obtain

$$8 + 5 * 4 - 12 / 6$$

- Then we evaluate multiplication and division to obtain 8+20-2.
- Last, we evaluate the addition and subtraction to get the result as 26.
- Observe that the **expression was evaluated 3 times** to take care of the precedence level.
- It makes the infix notation evaluation complex for system.

Polish and Reverse Polish Notations

- Polish notation, refers to the notation in which the operator symbol is placed before its two operands.
- For example:

$$(A+B)/(C-D) = [+AB]/[-CD] = / + A B - C D$$

- The fundamental property of polish notation is that we never need parentheses.
- Reverse polish notation is opposite of polish notation in which the operator is placed after its two operands.
- For example:

$$(A+B)/(C-D) = [AB+] / [CD-] = AB+ CD- /$$

Computer way of solving Arithmetic expression

- The computer usually evaluates an arithmetic expression written in infix notation in two steps:
 - First, it converts the expression into postfix notation
 - then , it evaluates the postfix notation
- In each step stack is the main tool to perform the task.
- The following order of precedence we would be following:

OPERATOR	PRECEDENCE	VALUE
Exponentiation (\$ or \uparrow or \wedge)	Highest	3
\ast , $/$	Next highest	2
$+$, $-$	Lowest	1

Conversion: Infix to Postfix- Algo

1. Scan the infix expression from left to right.
2. There are four possibility while scanning input that it would be either “**left parenthesis**”, “**right parenthesis**”, **operand**, or **operator**. We will perform the following task based on the input scanned:
 - a. If the scanned symbol is “**left parenthesis**”, **push** it onto the stack.
 - b. If the scanned symbol is “**right parenthesis**”, then **pop** all the items from the stack and place them in postfix expression till we get the matching “**left parenthesis**”.

Note: Do not keep parentheses on the postfix expression. Whenever, matching left parenthesis and right parenthesis will occur in stack they will be vanished but elements between those parentheses will be popped from the stack and insert in postfix expression.

Conversion: Infix to Postfix- Algo contd..

- c. If the scanned symbol is “**operand**”, then place it directly into postfix expression (output).
- d. If the scanned symbol is “**operator**”, then do the following:
 - i. If precedence of the operator which is on the top of the stack is greater than or equal to the precedence of scanned operator
 - 1. Remove all operators from the stack and place them in the postfix expression.
 - 2. Also, push the scanned operator on the **stack**.
 - ii. Else, push the scanned operator **directly** onto the **stack**.

Conversion: Infix to Postfix- Example

Convert $((A - (B + C)) * D) \uparrow (E + F)$ infix expression to postfix form:

Conversion: Infix to Postfix- Example

Convert $((A - (B + C)) * D) \uparrow (E + F)$ infix expression to postfix form:

Sl. No.	Symbol	Postfix expression	Stack	Stack Operation
1	-	-	empty	-
2	((Push '('
3	(((Push '('
4	A	A	((-
5	-	A	((-	Push '-'
6	(A	((- (Push '('
7	B	AB	((- (-
8	+	AB	((- (+	Push '+'
9	C	ABC	((- (+	-

Conversion: Infix to Postfix- Example

Convert $((A - (B + C)) * D) \uparrow (E + F)$ infix expression to postfix form:

Sl. No.	Symbol	Postfix expression	Stack	Stack Operation
10	-	ABC	((- (+	-
11)	ABC+	((-	Pop '+' and vanish '(' & ')
12)	ABC+ -	(Pop '-' and vanish '(' & ')
13	*	ABC+ -	(*	-
14	D	ABC+ - D	(*	-
15)	ABC+ - D *		Pop '*' and vanish '(' & ')
16	\uparrow	ABC+ - D *	\uparrow	Push ' \uparrow '
17	(ABC+ - D *	$\uparrow ($	Push '('

Conversion: Infix to Postfix- Example

Convert $((A - (B + C)) * D) \uparrow (E + F)$ infix expression to postfix form:

Sl. No.	Symbol	Postfix expression	Stack	Stack Operation
18	-	ABC+ - D *	↑(-
19	E	ABC+ - D * E	↑(-
20	+	ABC+ - D * E	↑(+	Push ‘+’
21	F	ABC+ - D * E F	↑(+	-
22)	ABC+ - D * E F +	↑	Pop ‘+’ and vanish ‘(’ & ‘)’
23	End of string	ABC+ - D * E F + ↑		The input is now empty, pop everything from stack and keep in postfix expression until it is empty.

Conversion: Infix to Postfix- Example2

Convert $a+b*c+ (d*e+f)*g$ infix expression to postfix form:

Conversion: Infix to Postfix- Example2

Convert $a+b*c+ (d*e+f)*g$ infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	
*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(a b c * +	+ (

Conversion: Infix to Postfix- Example2

Convert $a+b*c+ (d*e+f)*g$ infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
d	a b c * + d	+ (
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Postfix expression evaluation

- The postfix expression is evaluated easily by the use of a stack.
 - When a number is seen, it is pushed onto the stack;
 - When an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.
- When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Postfix expression evaluation: example

- Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

Postfix expression evaluation: example

- Evaluate the postfix expression: $6\ 5\ 2\ 3\ +\ 8\ *\ +\ 3\ +\ *$

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

Postfix expression evaluation: example

- Evaluate the postfix expression: $6\ 5\ 2\ 3\ +\ 8\ *\ +\ 3\ +\ *$

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as $8 * 5 = 40$ is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and $40 + 5 = 45$ is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes $45 + 3 = 48$ is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result $6 * 48 = 288$ is pushed

Quicksort: a stack application

- Quicksort is an algorithm of the divide and conquer type.
 - That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets.
- Let's understand this “reduction step” with the help of an example:

Consider A is the following list of 9 numbers:

10	16	8	12	15	6	3	9	5	
----	----	---	----	----	---	---	---	---	--

Following steps will be followed to sort the list using Quicksort.

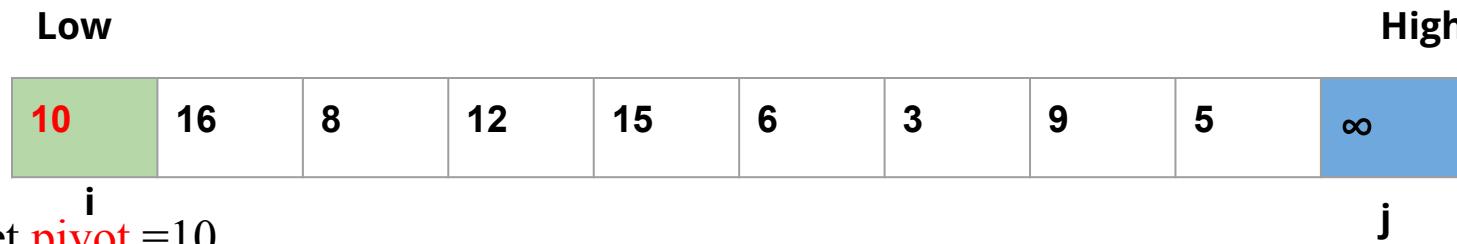
Quicksort working

Low										High
10	16	8	12	15	6	3	9	5	∞	

Set pivot = 10

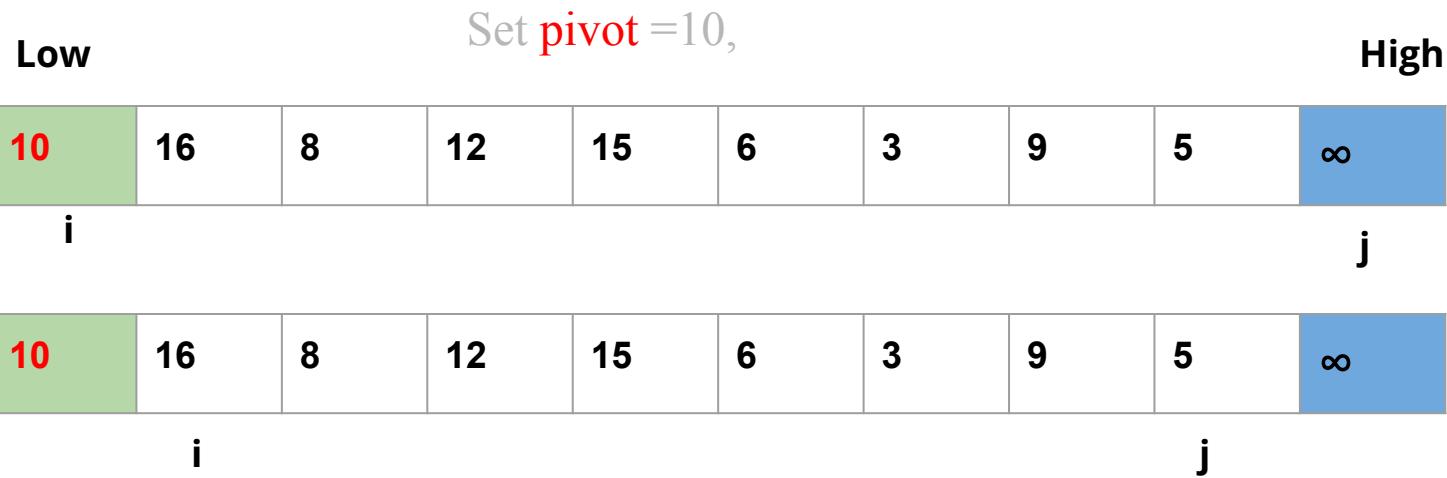
- Set the value of low, high and pivot element.
 - The Quicksort works in such a way that every time pivot element is sorted for that list.
 - We say that pivot element is sorted
 - if all elements left to the pivot element is smaller than pivot element.
 - all elements right to the pivot element is greater than pivot element.

Quicksort working Contd..



initialize **i** = low and **j** = high

- Repeat Step 1 to Step 3 unless **i > j**.
 - **Step 1:** Start searching an element from left to right starting from low and find the first greater number than pivot. For this we will use variable **i**.
 - **Step 2:** Start searching an element from right to left starting from high and find the first smaller number than pivot. For this we will use variable **j**.
 - **Step 3:** Swap **i** and **j**.



- Increment **i** until **i > pivot** and also decrement **j** until **j < pivot**.

Left to Right: $16 > 10$, and Right to Left: $5 < 10$

Swap the values of i and j.

Set pivot =10,

Low

High

10	16	8	12	15	6	3	9	5	∞
i								j	

Set i =low
and j =high

10	16	8	12	15	6	3	9	5	∞
i							j		

i++, for i>pivot
j--, for j<pivot

10	5	8	12	15	6	3	9	16	∞
i							j		

For i< j do:
Swap values of i
and j

Low

Set pivot =10,

High

10	16	8	12	15	6	3	9	5	∞
i								j	

Set i =low
and j =high

10	16	8	12	15	6	3	9	5	∞
i							j		

i++, for i>pivot
j--, for j<pivot

10	5	8	12	15	6	3	9	16	∞
i							j		

For i<j do:
Swap values of i
and j

10	5	8	12	15	6	3	9	16	∞
i							j		

i++, for i>pivot
j--, for j<pivot

Low

High										
10	5	8	9	15	6	3	12	16	∞	

i

j

For $i < j$ do:
Swap values of i
and j

Low

High										
10	5	8	9	15	6	3	12	16	∞	

i

j

i++, for $i >$ pivot
j--, for $j <$ pivot

Low

High										
10	5	8	9	3	6	15	12	16	∞	

i

j

For $i < j$ do:
Swap values of i
and j

Low

High										
10	5	8	9	3	6	15	12	16	∞	

j

i

i++, for $i >$ pivot
j--, for $j <$ pivot

Low

High										
6	5	8	9	3	10	15	12	16	∞	

Here $i > j$ do:
Swap value of j
with pivot

Low										High
6	5	8	9	3	10	15	12	16	∞	

- Observe the position of 10:
 - All elements left to 10 are smaller than 10
 - All elements right to 10 are greater than 10.
 - **Hence now 10 is sorted.**

Low

High

6	5	8	9	3	10	15	12	16	∞
---	---	---	---	---	----	----	----	----	----------

Sublist 1

6	5	8	9	3	10	15	12	16	∞
---	---	---	---	---	----	----	----	----	----------

i

j

Set pivot = 6

6	5	8	9	3
---	---	---	---	---

i

j

6	5	3	9	8
---	---	---	---	---

i

j

6	5	3	9	8
---	---	---	---	---

j

i

3	5	6	9	8
---	---	---	---	---

Sublist 2

Set pivot = 15

15	12	16	∞
----	----	----	----------

i

j

15	12	16	∞
----	----	----	----------

j

i

12	15	16	∞
----	----	----	----------

Low

Set pivot = 10

High

10	16	8	12	15	6	3	9	5	∞
----	----	---	----	----	---	---	---	---	----------

6	5	8	9	3	10	15	12	16	∞
---	---	---	---	---	----	----	----	----	----------

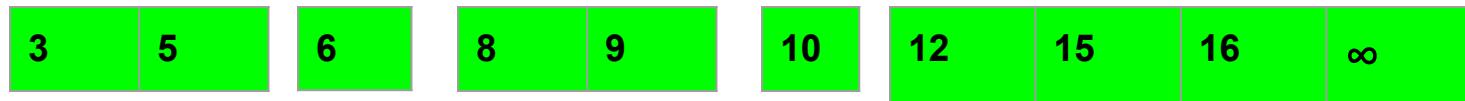
6	5	8	9	3	10	15	12	16	∞
---	---	---	---	---	----	----	----	----	----------

3	5	6	9	8	10	12	15	16	∞
---	---	---	---	---	----	----	----	----	----------

3	5	6	9	8	10	12	15	16	∞
---	---	---	---	---	----	----	----	----	----------

3	5	6	8	9	10	12	15	16	∞
---	---	---	---	---	----	----	----	----	----------

- All sorted sublists are merged to get the final sorted list:



Quicksort Algorithm

```
quicksort(low, high)
{
    if(low<high)
    {
        j=partition(low, high);
        quicksort(low, j);
        quicksort(j+1, high);
    }
}
```

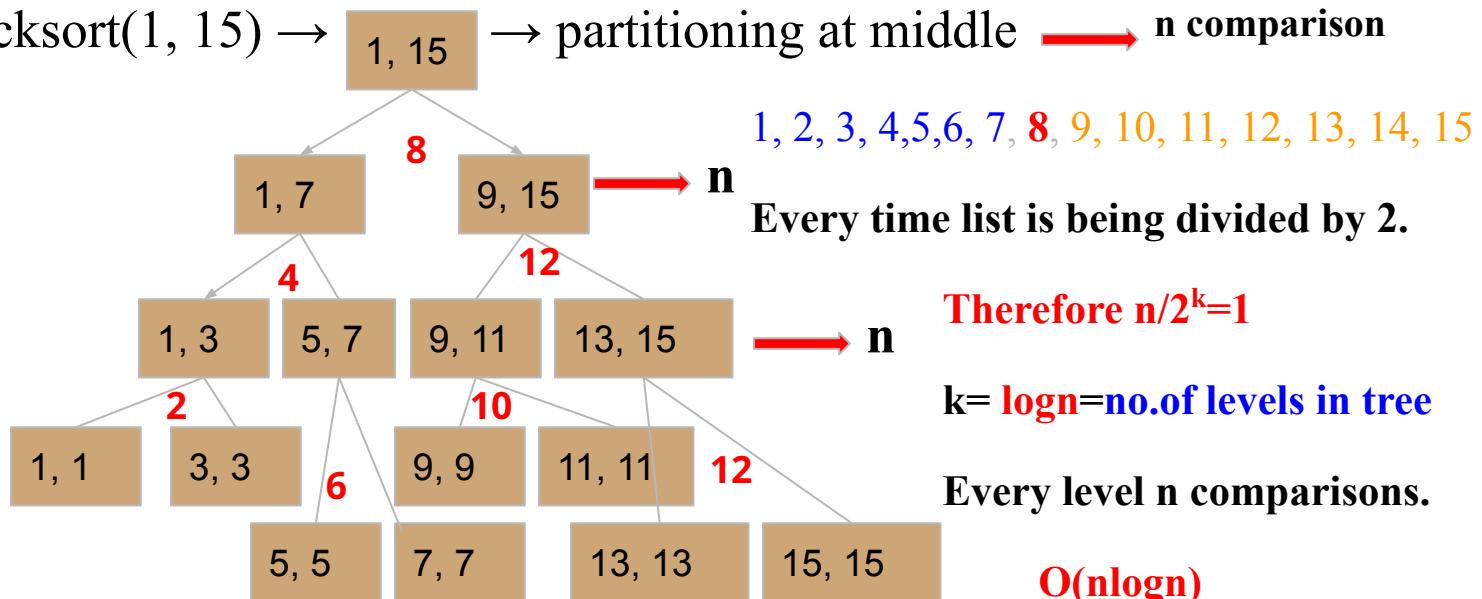
```
Partition(low, high)
{
    pivot=stack[i];
    i=low, j=high;
    while(i<j)
    {
        do
        {
            i++;
        }while(stack[i]<= pivot)

        do
        {
            j--;
        }while(stack[j]>=pivot)
        if (i<j)
            swap(stack[i],stack[j]);
    }
    swap(stack[low], stack[j]);
    return j; //partitioning position
}
```

Quicksort Analysis

Suppose we have list of 15 elements: 1, 2, 3, 4, ..., 15

If we call quicksort(1, 15) → $1, 15$ → partitioning at middle → n comparison



Quicksort Analysis: Best case

- We saw if the partition is done in the middle then **the complexity of Quicksort is $O(n\log n)$** .
 - Because everytime we partition the list in 2 equal halves sublist.
 - This is also the **Best case** complexity of Quicksort algorithm.
- **But, whether this best case is possible?**
 - For this, the pivot should always be the middle element.
 - So, for the best case
 - List must be sorted
 - And we always choose the pivot as median element
 - This case may not meet always hence achieving best case is not possible.

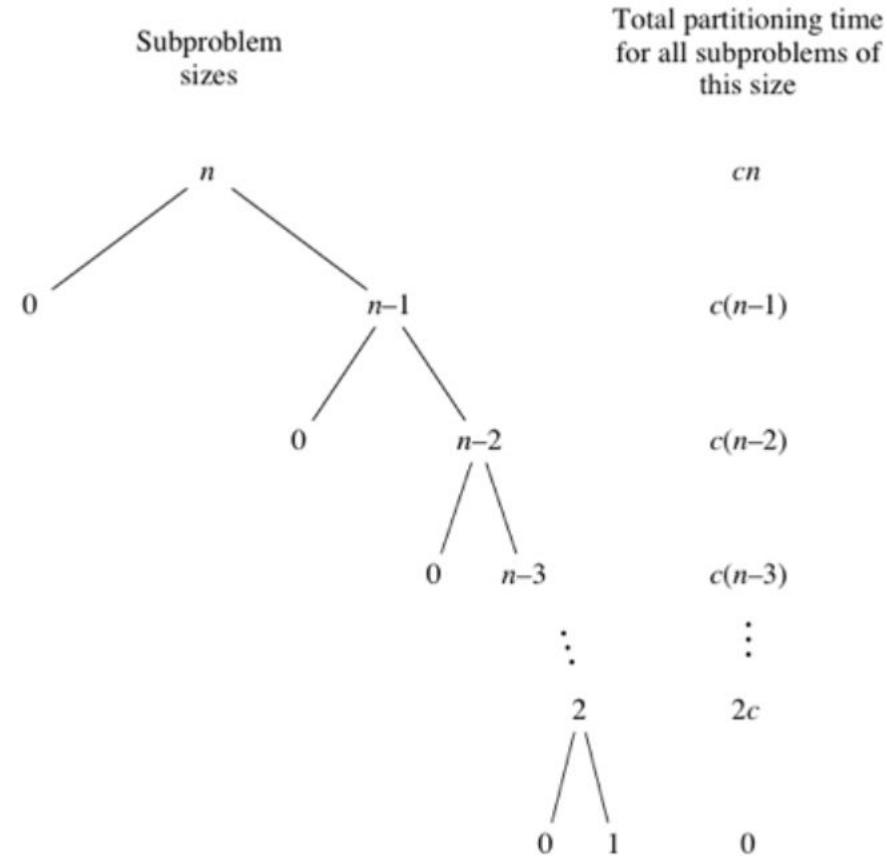
Worst case

- Worst case scenario occurs when the partition elements are always at one of the end of the sorted array.
- Total comparison:

$$n + (n-1) + (n-2) + \dots + 2 + 1 + 0 = n(n+1)/2$$

This is $O(n^2)$

Therefore Worst case complexity is $O(n^2)$.

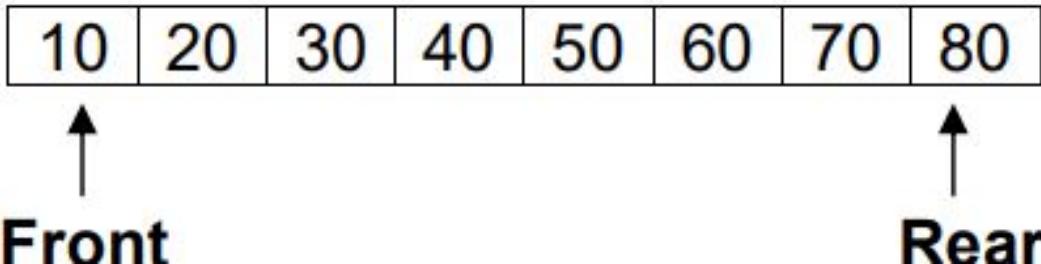


Queue

What is Queue data structure?

- Queue is a non-primitive linear data structure that permits insertion of an element at one end and deletion of an element at the other end.
- The end at which the deletion of an element take place is called front, and the end at which insertion of a new element can take place is called rear.
- The deletion or insertion of elements can take place only at the front and rear end of the list respectively.
- The first element that gets added into the queue is the first one to get removed from the list.
- Hence, Queue is also referred to as First-In-First-Out (FIFO) list.

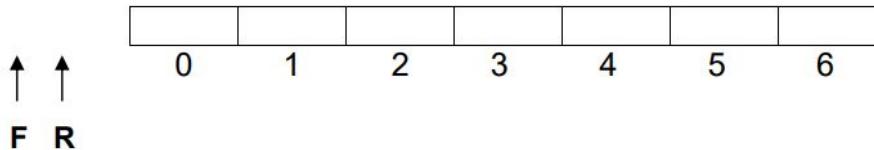
Queue representation: Front and Rear



- In the above representation of queue 10 is the first element and 80 is the last element added to the Queue.
- Similarly, 10 would be the first element to get removed and 80 would be the last element to get removed.

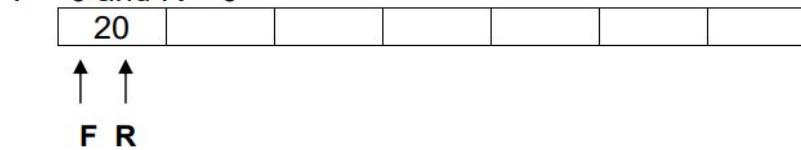
Queue representation: insertion operation

$F = -1$ and $R = -1$



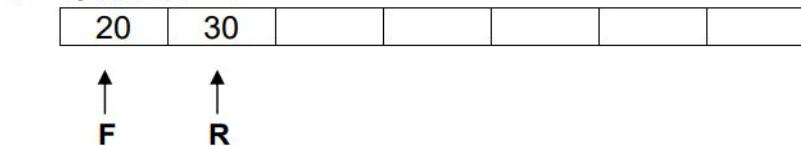
Empty queue

$F = 0$ and $R = 0$



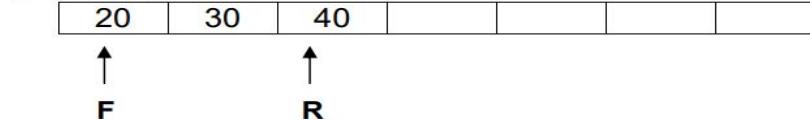
One element queue

$F = 0$ and $R = 1$



Two element queue

$F = 0$ and $R = 2$



Three element queue

Queue representation: insertion operation

- It is clear from the above insertion operations that whenever we **insert an element in the queue, the value of Rear is incremented by one**
i.e. $\text{Rear} = \text{Rear} + 1$
- Also, **during the insertion of the first element** in the queue we incremented the Front by one
i.e. $\text{Front} = \text{Front} + 1$
- **Afterwards the Front will not be changed during the entire operation.**

Queue representation: deletion operation

$$F = 0 \text{ and } R = 2$$

20	30	40				
----	----	----	--	--	--	--

↑
F

↑
R

Current queue

$$F = 1 \text{ and } R = 2$$

	30	40				
--	----	----	--	--	--	--

↑
F ↑
R

One element deleted from queue

$$F = 2 \text{ and } R = 2$$

			40			
--	--	--	----	--	--	--

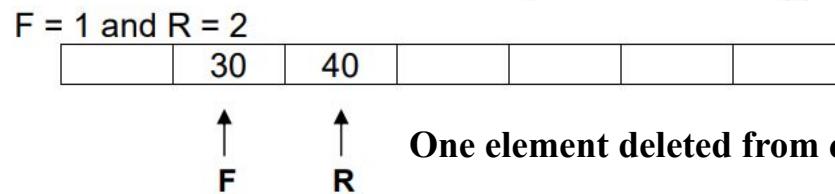
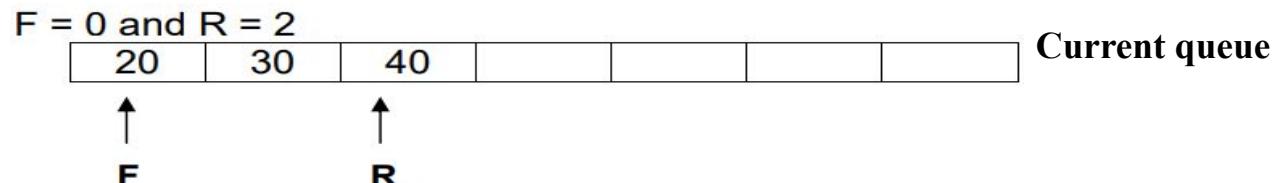
↑
F ↑
R

Two elements deleted from queue

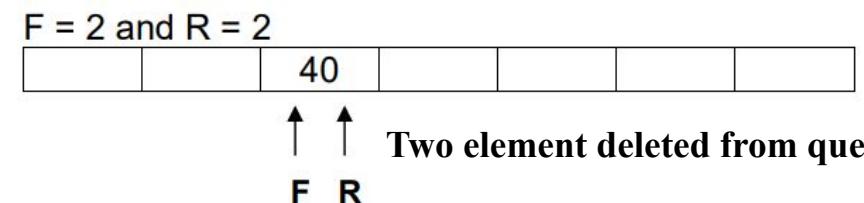
- It is clear from the above deletion operations that whenever we **delete an element in the queue, the value of Front is incremented by one**

i.e. $\text{Front} = \text{Front} + 1$

Queue representation: deletion operation

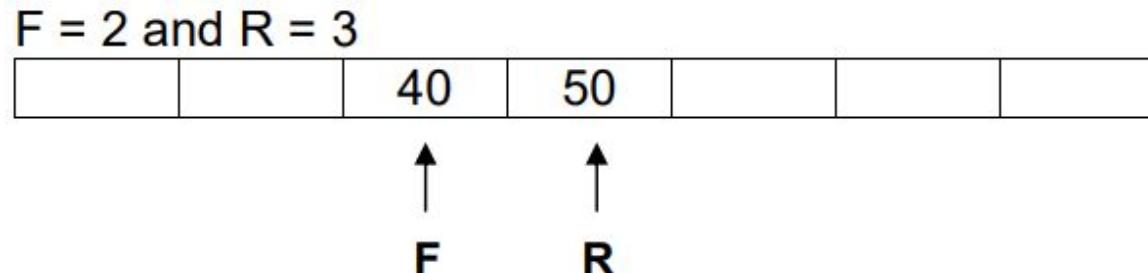


One element deleted from queue



Two elements deleted from queue

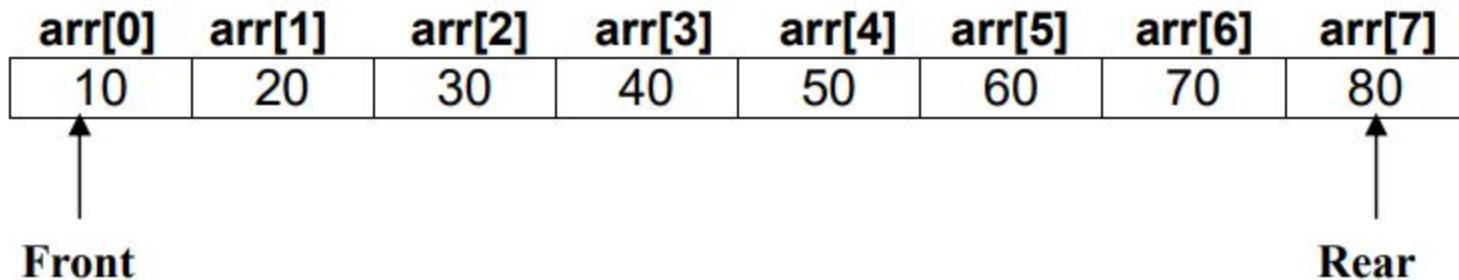
If we add one more element after deletion, the queue will look like:



Implementation of linear queue

- Queues can be implemented in two ways :
 - **Static implementation (using arrays)**
 - **Dynamic implementation (using pointers)**
- **Static implementation:**
 - Static implementation of Queue is represented by arrays.
 - If Queue is implemented using arrays, we must be sure about the exact number of elements we want to store in the queue, because we have to declare the size of the array at design time or before the processing starts.
 - In this case, the **beginning of the array** will **become the front** for the queue and the **last location of the array** will **act as rear** for the queue.

Implementation of linear queue: as array

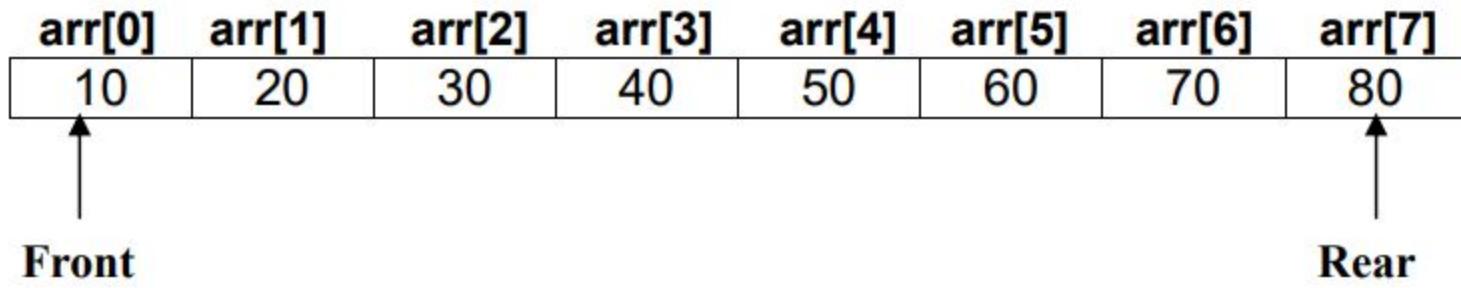


- The following relation gives the total number of elements present in the queue, when implemented using arrays :

$$\text{Total elements in queue} = \text{rear} - \text{front} + 1$$

- Also note that if $\text{front} > \text{rear}$, then there will be no element in the queue or queue is empty.

Operations on a Queue



- Three basic operations which can be performed on queue are:
 - To **insert** an element in a queue
 - To **delete** an element from a queue
 - To **traverse** elements in a queue

Queue: insertion

```
void lqinsert()
{
int num;
if(rear==MAXSIZE-1)
{
    printf("\nQueue is full (Queue overflow)");
    return;
}
printf("\nEnter the element to be inserted : ");
scanf("%d",&num);
rear++;
queue[rear]=num;
if(front==-1)
    front=0;
}
```

Step 1 :If REAR = (MAXSIZE –1) : then
 Write : “Queue Overflow” and return
 [End of If structure]
Step 2 : Read NUM to be inserted in Linear Queue.
Step 3 : Set REAR := REAR + 1
Step 4 : Set QUEUE[REAR] := NUM
Step 5 : If FRONT = -1 : then
 Set FRONT=0.
 [End of If structure]
Step 6 : Exit

- Let QUEUE[MAXSIZE] is an array for implementing the Linear Queue & NUM is the element to be inserted in linear queue.
- FRONT represents the index number of the element at the beginning of the queue
- and REAR represents the index number of the element at the end of the Queue.

Queue: deletion

```
void lqdelete()
{
if(front == -1)
{
printf("\nQueue is empty (Queue underflow)");
return;
}
int num;
num=queue[front];
printf("\nDeleted element is : %d",num);
front++;
if(front>rear)
front=rear=-1;
}
```

- Step 1 : If FRONT = -1 : then
 Write : "Queue Underflow" and return
 [End of If structure]
- Step 2 : Set NUM := QUEUE[FRONT]
- Step 3 : Write "Deleted item is : ", NUM
- Step 4 : Set FRONT := FRONT + 1.
- Step 5 : If FRONT>REAR : then
 Set FRONT := REAR := -1.
 [End of If structure]
- Step 6 : Exit
- Let QUEUE[MAXSIZE] is an array for implementing the Linear Queue & NUM is the element to be deleted from linear queue.
 - FRONT represents the index number of the element at the beginning of the queue
 - and REAR represents the index number of the element at the end of the Queue.

Linked list representation of Queue

- Let queue be a structure whose declarations looks like follows :

```
struct queue
{
    int info;
    struct queue *link;
}*start=NULL;
```

Dynamic Queue: insertion

```
void lqinsert()
{
    struct queue *ptr;
    int num;
    ptr=(struct queue*)malloc(sizeof(struct queue));
    printf("\nEnter element to be inserted in queue : ");
    scanf("%d",&num);
    ptr->info=num;
    ptr->link=NULL;
    if(front==NULL)
    {
        front=ptr;
        rear=ptr;
    }
    else
    {
        rear->link=ptr;
        rear=ptr;
    }
}
```

Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into linear queue.
Step 3 : Set PTR->INFO = NUM
Step 4 : Set PTR->LINK= NULL
Step 5 : If FRONT = NULL : then
 Set FRONT=REAR=PTR
Else
 Set REAR->LINK=PTR;
 Set REAR=PTR;
[End of If Else Structure]

Step 6 : Exit

- Let PTR is the structure pointer which allocates memory for the new node & NUM is the element to be inserted into linear queue.
- INFO represents the information part of the node and LINK represents the link or next pointer pointing to the address of next node.
- FRONT represents the address of first node, REAR represents the address of the last node.
- Initially, Before inserting first element in the queue, FRONT=REAR=NULL.

Dynamic Queue: deletion

Step 1 : If FRONT = NULL : then

 Write 'Queue is Empty(Queue Underflow)' and return.

 [End of If structure]

Step 2 : Set PTR = FRONT

Step 3 : Set NUM = PTR->INFO

Step 4 : Write 'Deleted element from linear queue is : ',NUM.

Step 5 : Set FRONT = FRONT->LINK

Step 6 : If FRONT = NULL : then

 Set REAR = NULL.

 [End of If Structure].

Step 7 : Deallocate memory of the node at the beginning of queue using PTR.

Step 8 : Exit.

- Let PTR is the structure pointer which deallocates memory of the first node in the linear queue & NUM is the element to be deleted from queue.
- INFO represents the information part of the deleted node and LINK represents the link or next pointer of the deleted node pointing to the address of next node.
- FRONT represents the address of first node, REAR represents the address of the last node.

Dynamic Queue: deletion

- Let **PTR** is the structure pointer which deallocates memory of the first node in the linear queue & **NUM** is the element to be deleted from queue.
- INFO** represents the information part of the deleted node and **LINK** represents the link or next pointer of the deleted node pointing to the address of next node.
- FRONT** represents the address of first node, **REAR** represents the address of the last node.

```
void lqdelete()
{
if(front==NULL)
{
printf("\nQueue is empty (Queue underflow)");
return;
}
struct queue *ptr;
int num;
ptr=front;
num=ptr->info;
printf("\nThe deleted element is : %d",num);
front=front->link;
if(front==NULL)
rear=NULL;
free(ptr);
}
```

Linear queue limitation

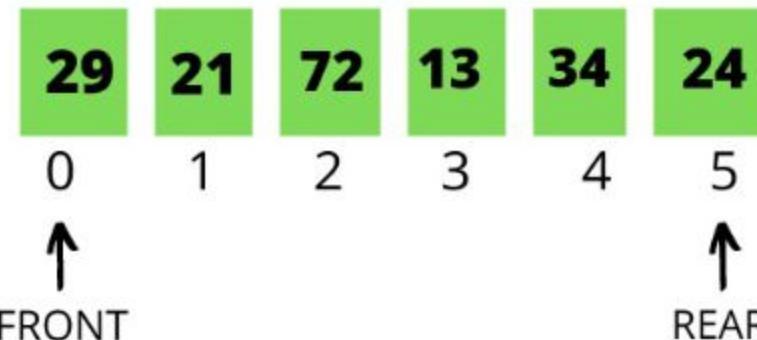
- Consider a queue of size 6 is declared to maintain 6 elements.
- Initially, when queue is empty, we have

Front = Rear = -1;

- Then we insert 6 elements from rear end as follows:

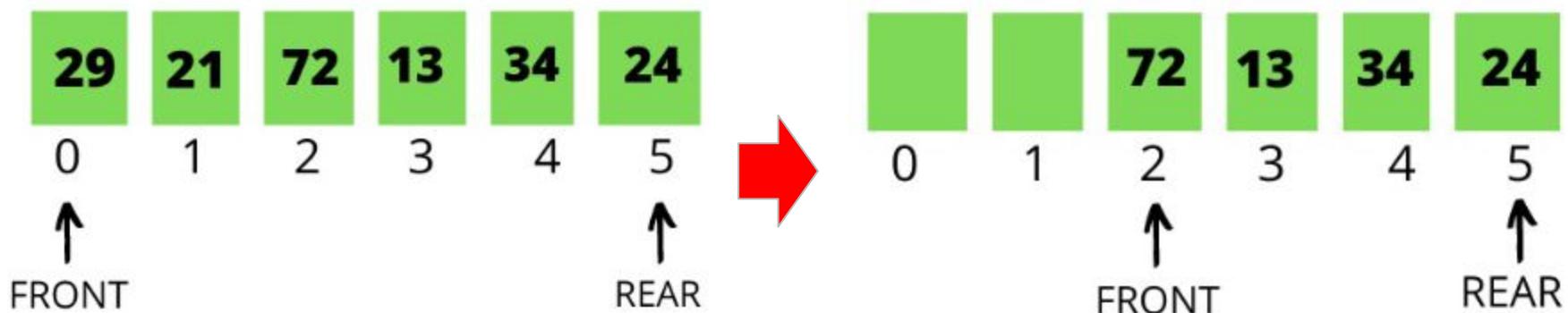
Front = 0

Rear=5



Linear queue limitation Contd..

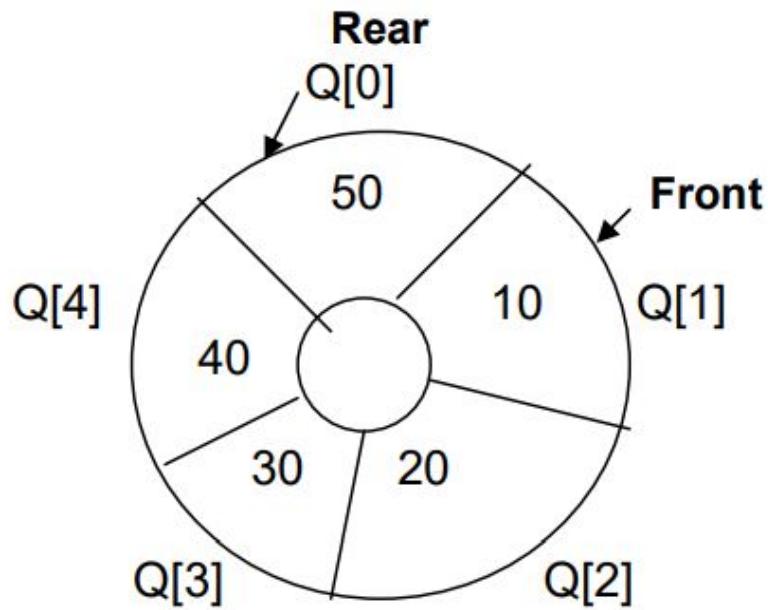
- Now say we need to delete 2 elements from the queue and hence it becomes as follows:



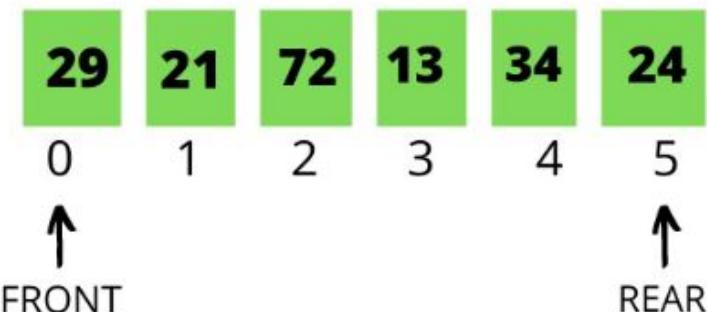
- Further if we want to enqueue 100, although 2 positions are free in the queue, it can not be performed. This is where linear queue lags.

Linear queue limitation: Solution

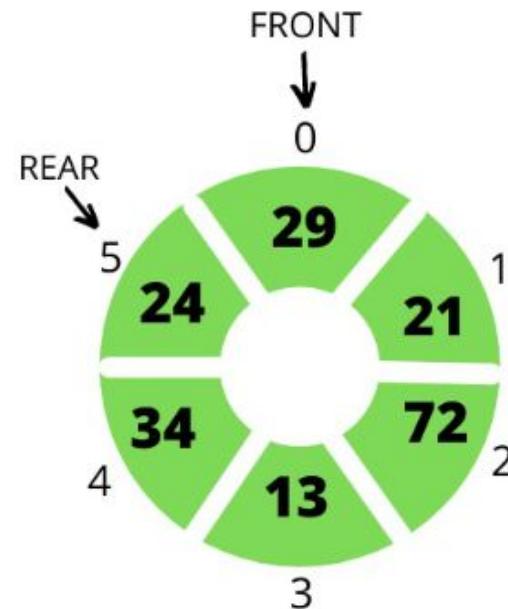
- Limitation of linear queue can be addressed using circular queue.
- Circular Queue is just a variation of the linear queue in which front and rear-end are connected to each other to optimize the space wastage of the Linear queue and make it efficient.



Linear queue Vs Circular queue: after enqueue

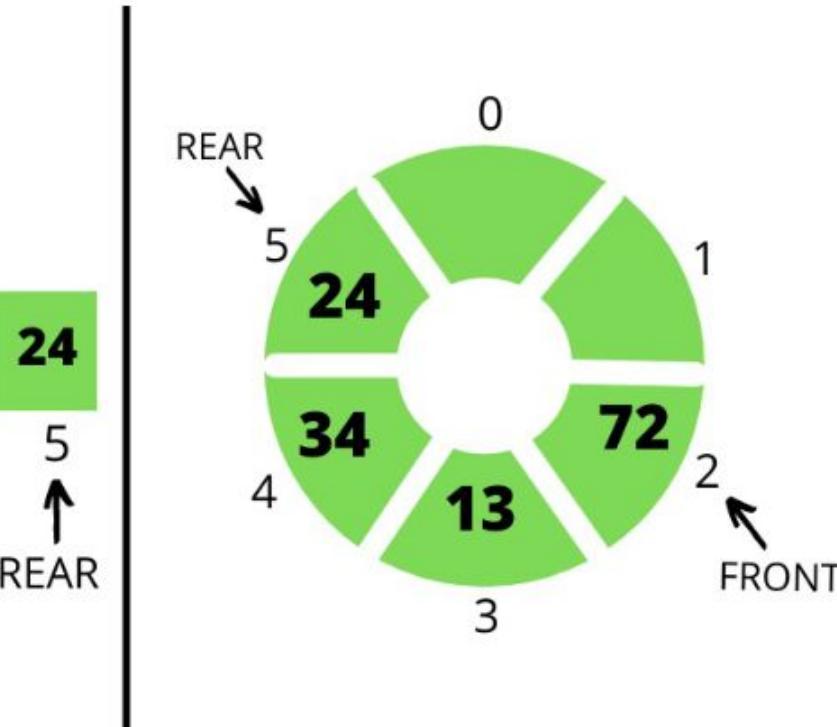
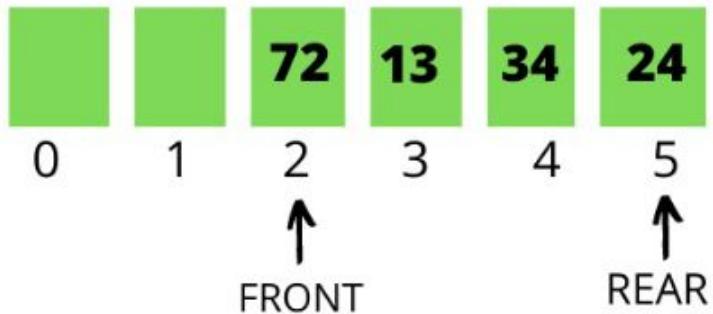


LINEAR QUEUE

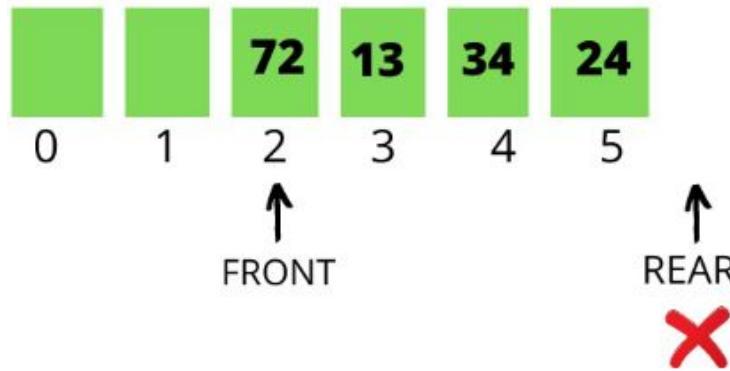


CIRCULAR QUEUE

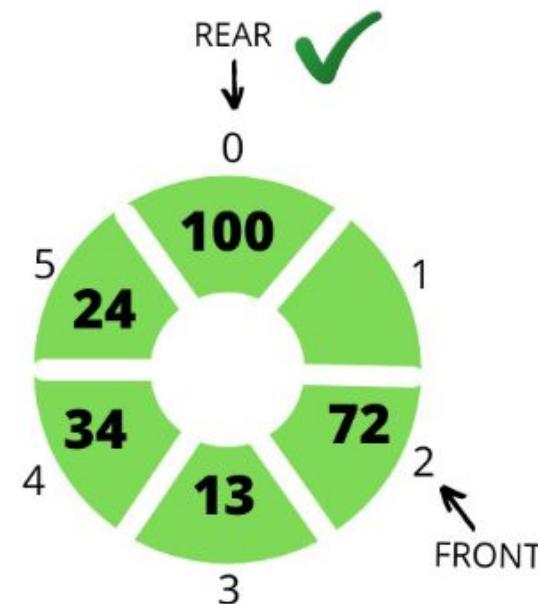
Linear queue Vs Circular queue: after dequeuing 2 elements



Linear queue Vs Circular queue: enqueue an element 100



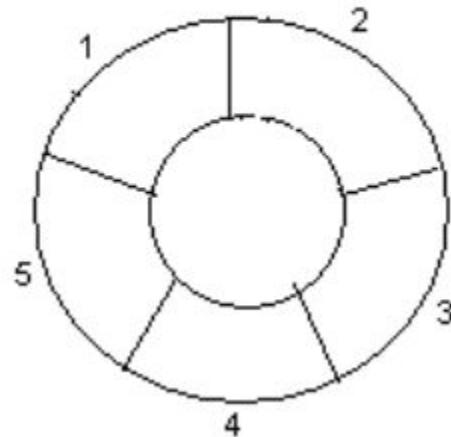
Insertion not possible !



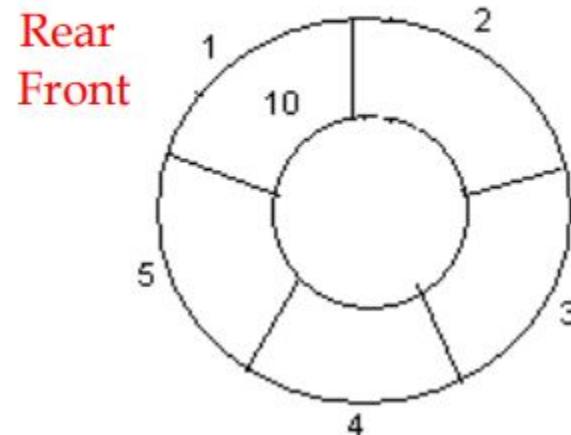
Insertion possible !

Circular queue: operations example

1. Initially, Rear = 0, Front = 0.

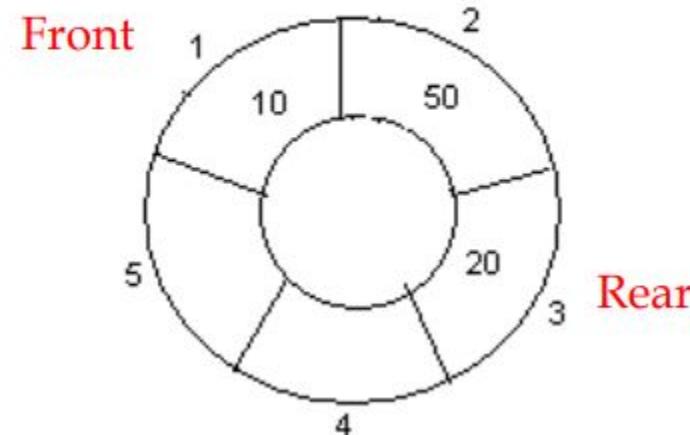
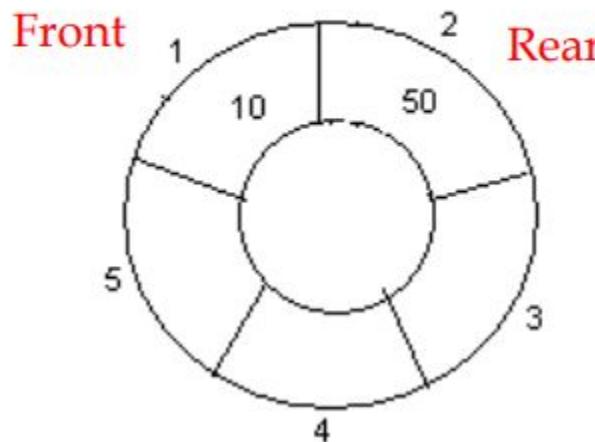


2. Insert 10, Rear = 1, Front = 1.



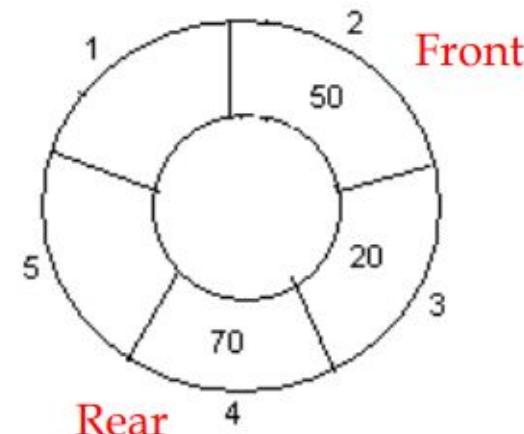
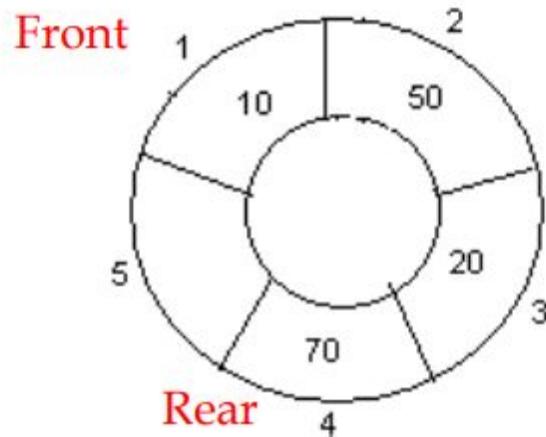
Circular queue: operations example

3. Insert 50, Rear = 2, Front = 1. 4. Insert 20, Rear = 3, Front = 1



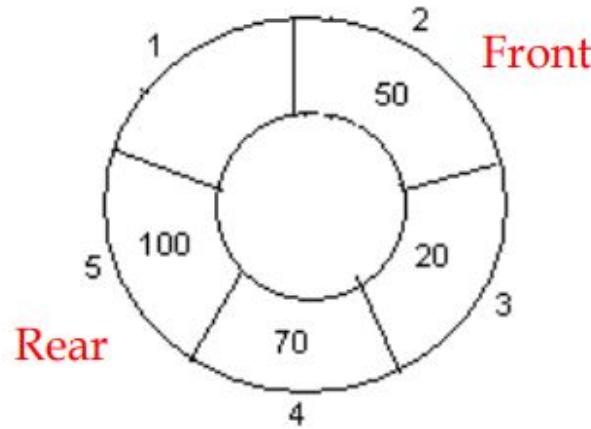
Circular queue: operations example

5. Insert 70, Rear = 4, Front = 1. 6. Delete front, Rear = 4, Front = 2.

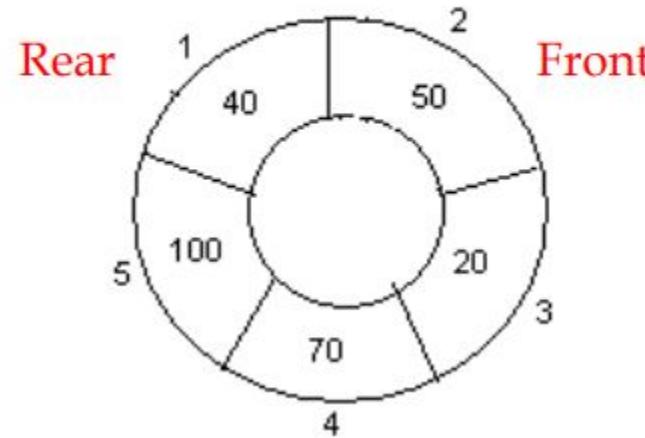


Circular queue: operations example

7. Insert 100, Rear = 5, Front = 2.

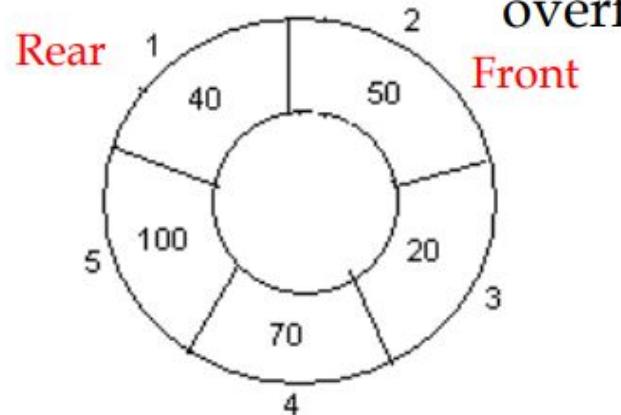


8. Insert 40, Rear = 1, Front = 2.

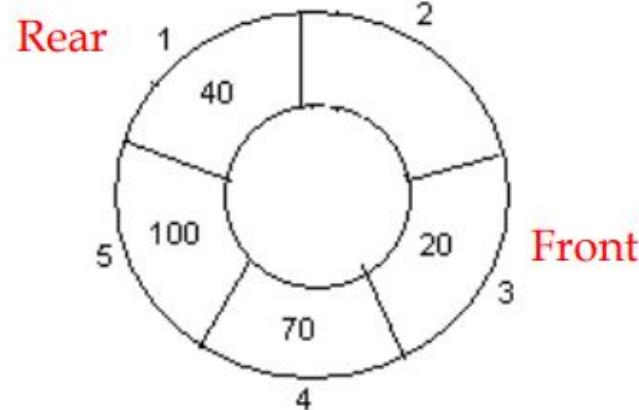


Circular queue: operations example

9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.

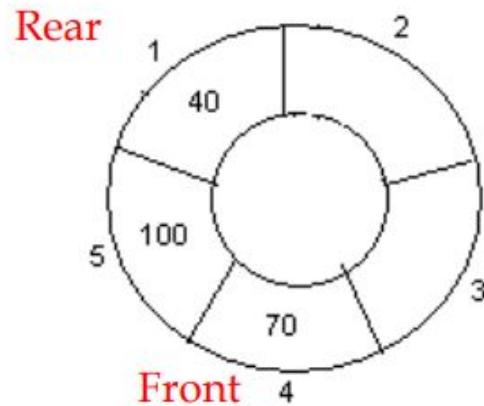


10. Delete front, Rear = 1, Front = 3.

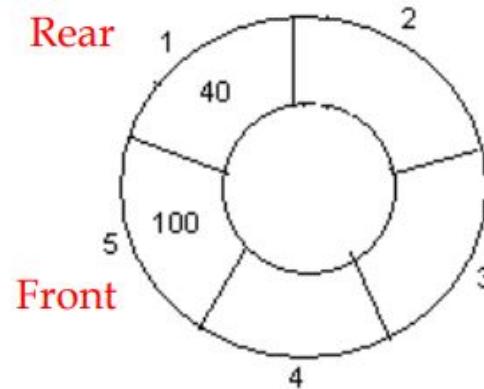


Circular queue: operations example

11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



Circular Queue: insertion

```
#include <stdio.h>
#define N 5
int queue[N];
int front=-1;
int rear=-1;
int enqueue(int x)
{
    if (front== -1 && rear == -1)
    {
        front=rear=0;
        queue[rear]=x;
    }
    else if((rear+1)%N==front)
    {
        printf("Circular queue is full");
    }
    else
    {
        rear=(rear+1)%N;
        queue[rear]=x;
    }
}
```

- Step 1 : If FRONT = (REAR + 1) % MAXSIZE : then
 Write : "Queue Overflow" and return.
 [End of If structure]
- Step 2 : Read NUM to be inserted in Circular Queue.
- Step 3 : If FRONT= -1 : then
 Set FRONT = REAR =0.
 Else
 Set REAR=(REAR + 1) % MAXSIZE.
 [End of If Else structure]
- Step 4 : Set CQUEUE[REAR]=NUM;
- Step 5 : Exit

- Let **CQUEUE[MAXSIZE]** is an array for implementing the Circular Queue, where **MAXSIZE** represents the max. size of array.
- **NUM** is the element to be inserted in circular queue, **FRONT** represents the index number of the element at the beginning of the queue and **REAR** represents the index number of the element at the end of the Queue.

Circular Queue: deletion

- Let CQUEUE[MAXSIZE] is an array for implementing the Circular Queue, where MAXSIZE represents the max. size of array.
- NUM is the element to be deleted from circular queue, FRONT represents the index number of the first element inserted in the Circular Queue and REAR represents the index number of the last element inserted in the Circular Queue.

Step 1 : If FRONT = - 1 : then

 Write : "Queue Underflow" and return.

 [End of If Structure]

Step 2 : Set NUM = CQUEUE[FRONT].

Step 3 : Write 'Deleted element from circular queue is : ',NUM.

Step 4 : If FRONT = REAR : then

 Set FRONT = REAR = -1;

Else

 Set FRONT = (FRONT + 1) % MAXSIZE.

Step 5 : Exit

Circular Queue: deletion

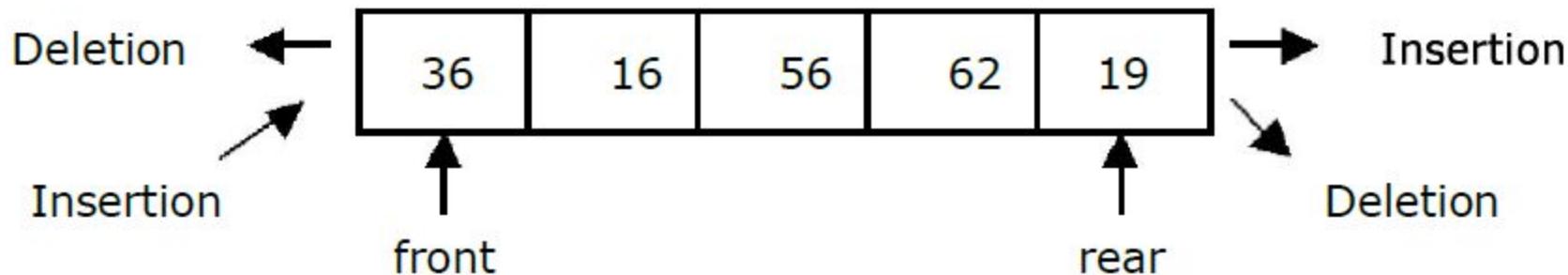
- Let CQUEUE[MAXSIZE] is an array for implementing the Circular Queue, where MAXSIZE represents the max. size of array.
- NUM is the element to be deleted from circular queue, FRONT represents the index number of the first element inserted in the Circular Queue and REAR represents the index number of the last element inserted in the Circular Queue.

```
void cqdelete()
{
int num;
if(front== -1)
{
printf("\nQueue is Empty (Queue underflow)");
return;
}
num=cqueue[front];
printf("\nDeleted element from circular queue is : %d",num);

if(front==rear)
front=rear=-1;
else
front=(front+1)%MAXSIZE;
}
```

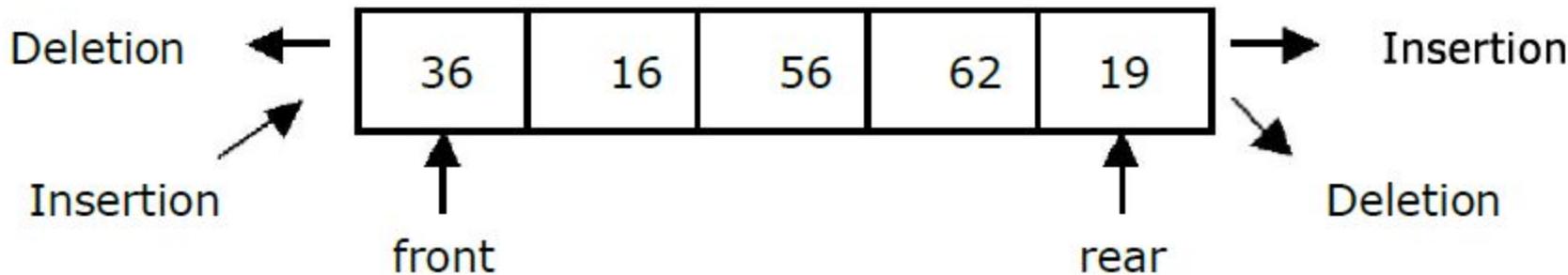
Double ended queue: Deque

- Till now we saw that a queue in which we insert items at one end and from which we remove items at the other end.
- Now, we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue.
- This data structure is a **deque**. The word deque is an acronym derived from **double-ended queue**.



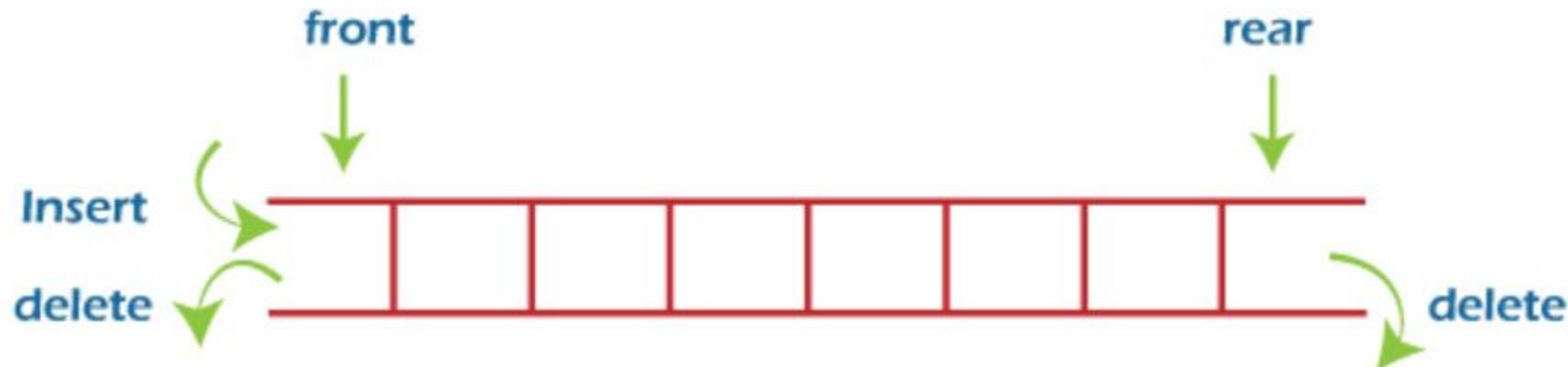
Double ended queue: Deque

- Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule.
- Types of deque:**
 - Input restricted deque
 - Output restricted deque



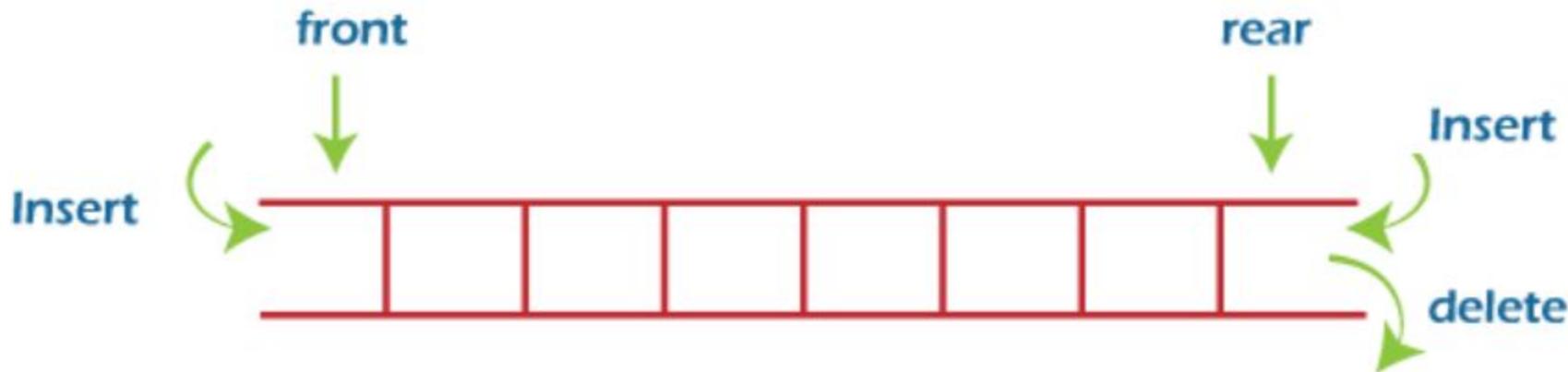
Deque: input restricted

- In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



Deque: output restricted

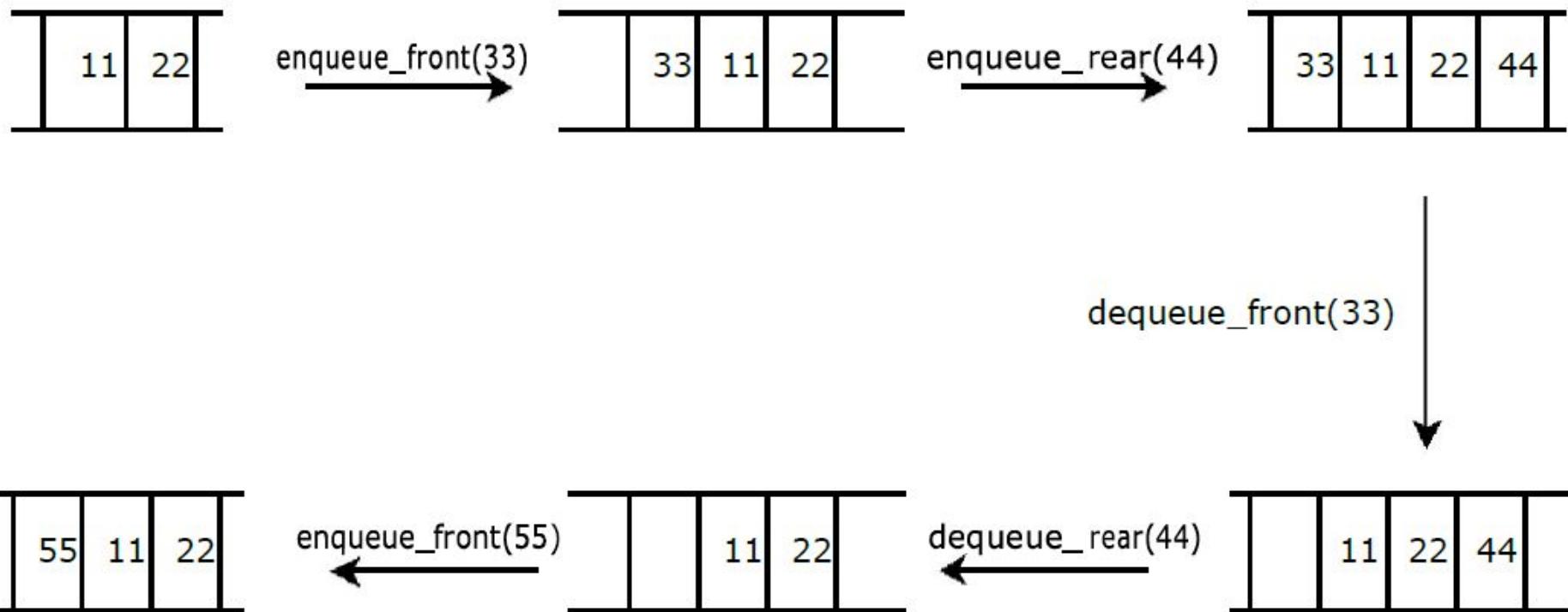
- In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Deque: operations

- There are the following operations that can be applied on a deque -
 - Insertion at front
 - Insertion at rear
 - Deletion at front
 - Deletion at rear
- Other operations which can be performed on deque are:
 - Get the front item from the deque (peek)
 - Get the rear item from the deque (peek)
 - Check whether the deque is full or not (Overflow)
 - Checks whether the deque is empty or not (underflow)

Deque: operations examples

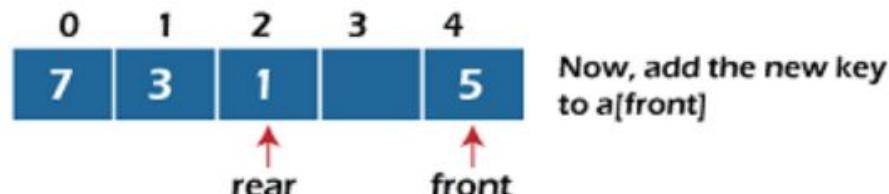
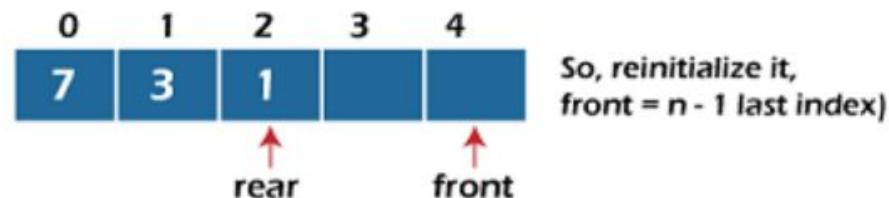
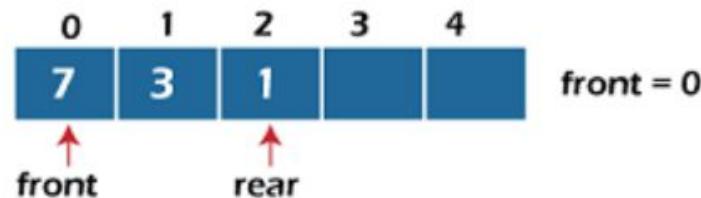


Deque: Insertion at front end

- In this operation, the element is inserted from the front end of the queue.
- Before implementing the operation, we first have to check whether the queue is full or not.
- If the queue is not full, then the element can be inserted from the front end by using the below conditions -
 - If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
 - Otherwise, check the position of the front if the front is less than 1 (**front < 1**), then reinitialize it by **front = n - 1**, i.e., the last index of the array.

Deque: Insertion at front end

- If the queue is not full, then the element can be inserted from the front end by using the below conditions -
 - If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
 - Otherwise, check the position of the front if the front is less than 1 (**front < 1**), then reinitialize it by **front = n - 1**, i.e., the last index of the array.

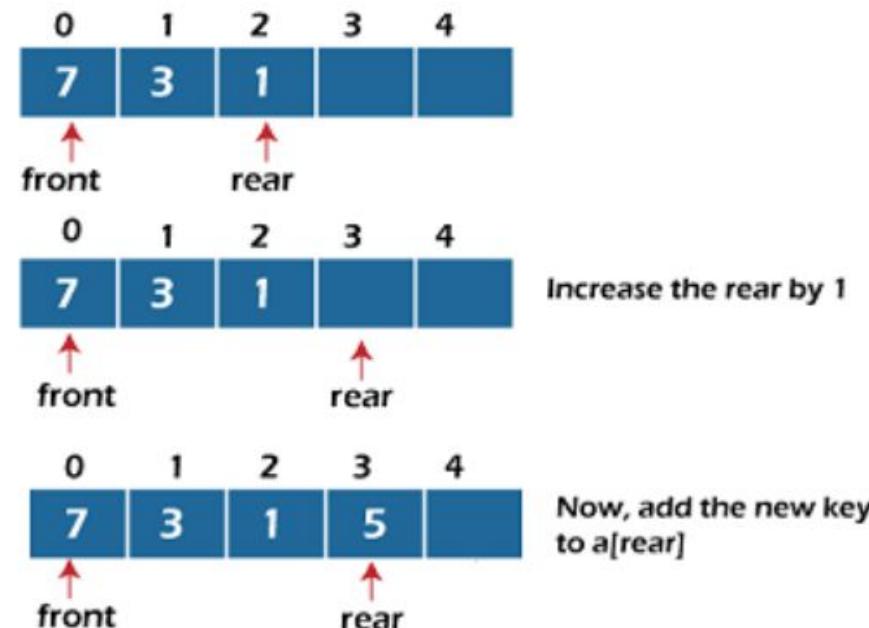


Deque: Insertion at rear end

- In this operation, the element is inserted from the rear end of the queue.
- Before implementing the operation, we first have to check again whether the queue is full or not.
- If the queue is not full, then the element can be inserted from the rear end by using the below conditions -
 - If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
 - Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.

Deque: Insertion at rear end

- If the queue is not full, then the element can be inserted from the rear end by using the below conditions -
 - If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
 - Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.

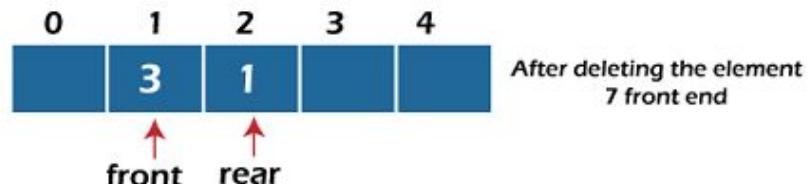
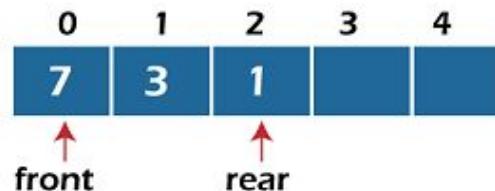


Deque: Deletion at front end

- In this operation, the element is deleted from the front end of the queue.
- Before implementing the operation, we first have to check whether the queue is empty or not.
- If the queue is empty, i.e., $\text{front} = -1$, it is the underflow condition, and we cannot perform the deletion.
- If the queue is not full, then the element can be deleted from the front end by using the below conditions -

Deque: Deletion at front end

- **If** the deque has only one element, set rear = -1 and front = -1.
- **Else if** front is at end (that means front = size - 1), set front = 0.
- **Else** increment the front by 1, (i.e., front = front + 1).

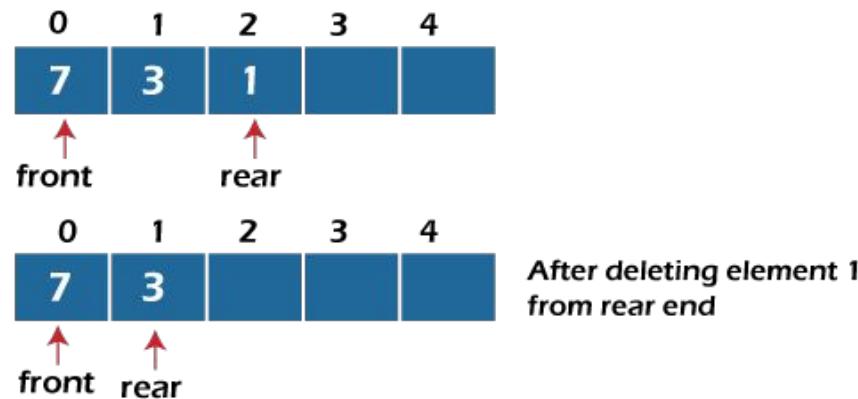


Deque: Deletion at rear end

- In this operation, the element is deleted from the rear end of the queue.
- Before implementing the operation, we first have to check whether the queue is empty or not.
- If the queue is empty, i.e., $\text{front} = -1$, it is the underflow condition, and we cannot perform the deletion.
- If queue is not empty, perform following operations:

Deque: Deletion at rear end

- **If** the deque has only one element, set rear = -1 and front = -1.
- **Else if** rear = 0 (rear is at front), then set **rear = n - 1**.
- **Else**, decrement the rear by 1 (or, rear = rear -1).



Deque: Check empty and Full condition

Check empty

- This operation is performed to check whether the deque is empty or not.
- If **front = -1**, it means that the deque is empty.

Check full

- This operation is performed to check whether the deque is full or not.
- If **front = rear + 1**, or **front = 0 and rear = n - 1** it means that the deque is full.

The time complexity of all of the above operations of the deque is $O(1)$, i.e., constant.

Deque: Applications

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Dequeue implementation

```
void display()
{
    int i=f;
    printf("\nElements in a deque are: ");

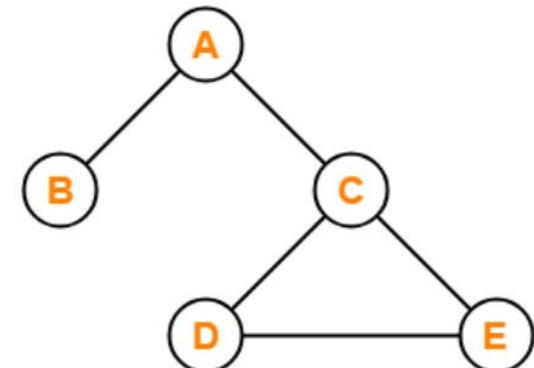
    while(i!=r)
    {
        printf("%d ",deque[i]);
        i=(i+1)%size;
    }
    printf("%d",deque[r]);
}
```

```
#define size 5
int deque[size];
int f = -1, r = -1;
// insert_front function will insert the value from the front
void insert_front(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        f=r=0;
        deque[f]=x;
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}
```

Tree

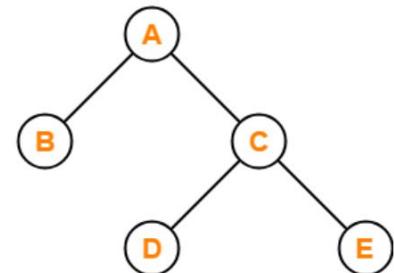
Tree

- Tree data structure may be defined as:
 - Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.
 - OR, A tree is a connected graph without any circle.
 - OR If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.
- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has exactly $(n-1)$ edges.



✗

This graph is not a Tree



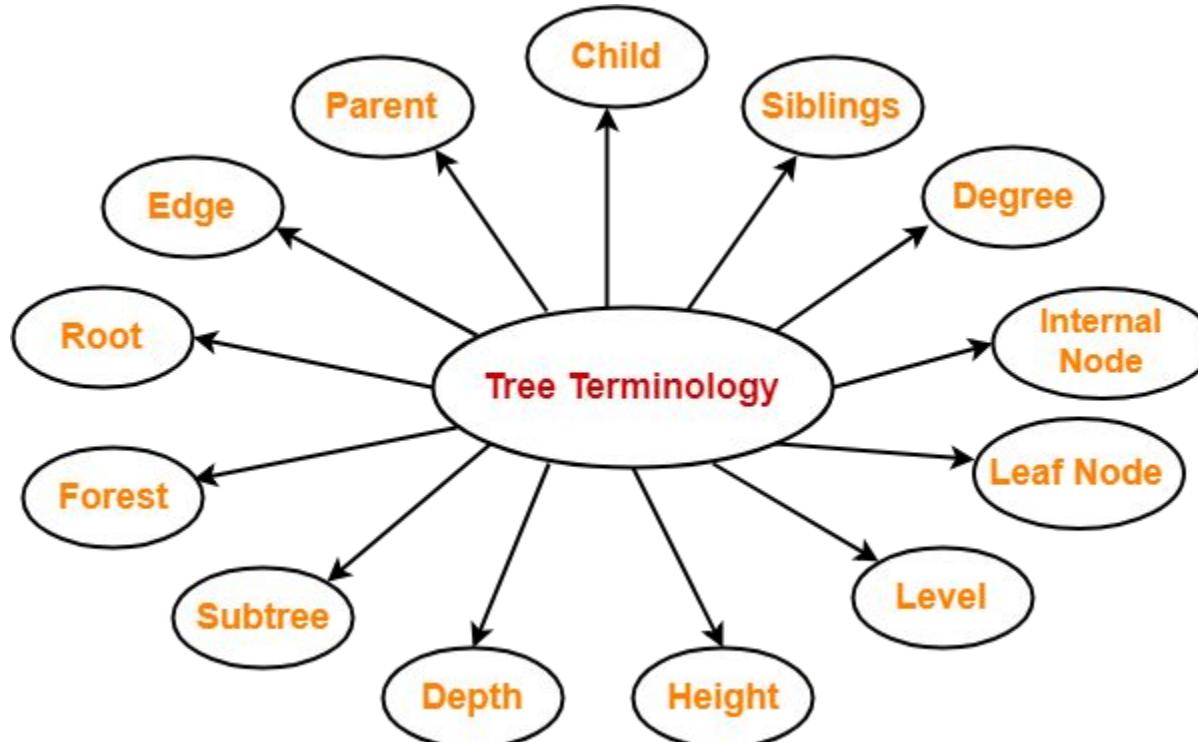
✓

This graph is a Tree

Tree Application

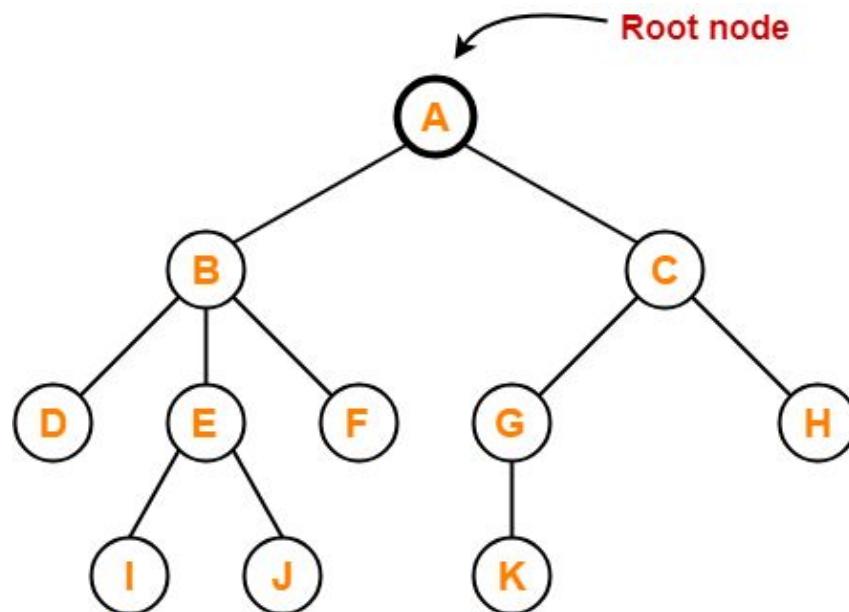
- Databases make use of the tree data structure for indexing purposes
- Tree structures are utilized by Domain Name Server (DNS)
- XML Parser also makes use of tree structures
- File Explorer or My Computer of any mobile phone or computer
- The comments on any of the questions posted on websites have comments as the child of those questions.
- The decision-based algorithms being used in machine learning work upon the principle of the algorithm of a tree structure.

Tree terminology



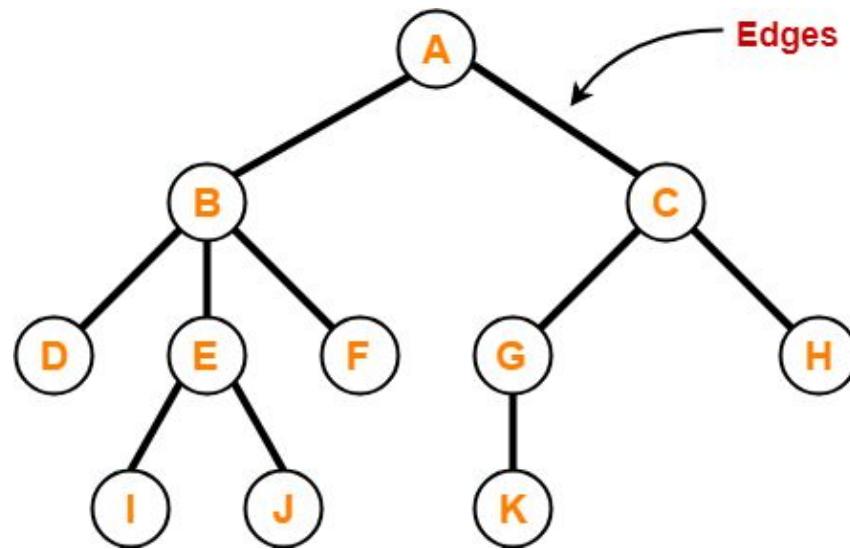
Tree terminology: Root

- The first node from where the tree originates is called as a root node.
- In any tree, there must be only one root node. We can never have multiple root nodes in a tree data structure.



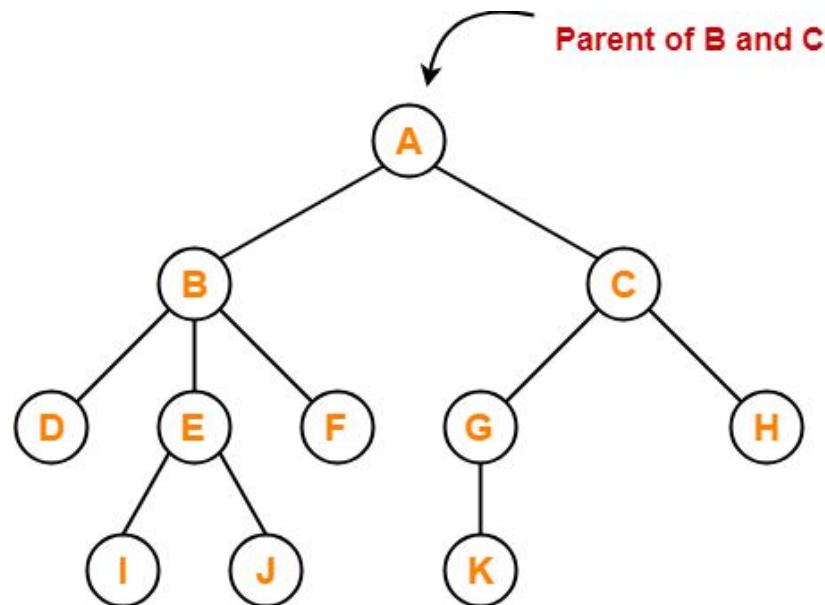
Tree terminology: Edge

- The connecting link between any two nodes is called as an edge.
- In a tree with n number of nodes, there are exactly $(n-1)$ number of edges.



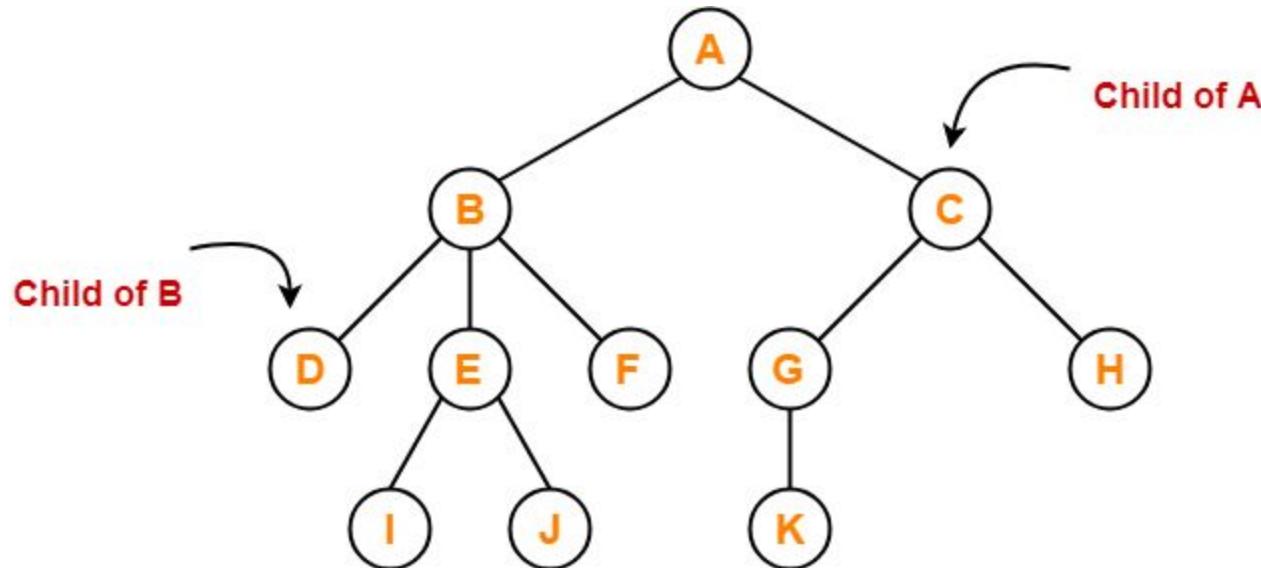
Tree terminology: Parent

- The node which has a branch from it to any other node is called as a parent node.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.



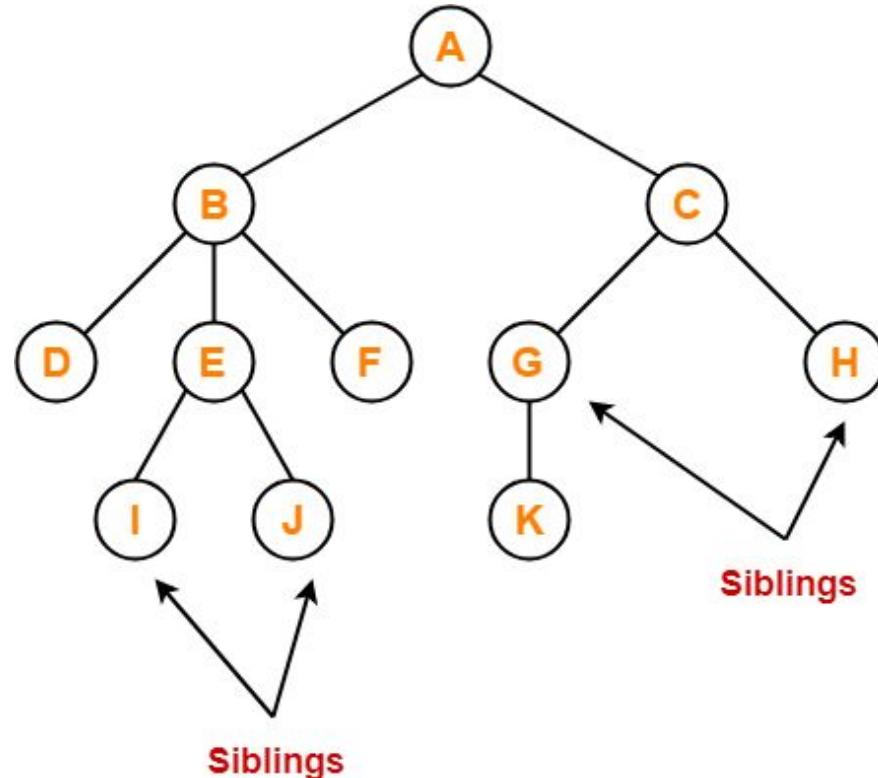
Tree terminology: Child

- The node which is a descendant of some node is called as a child node.
All the nodes except root node are child nodes.



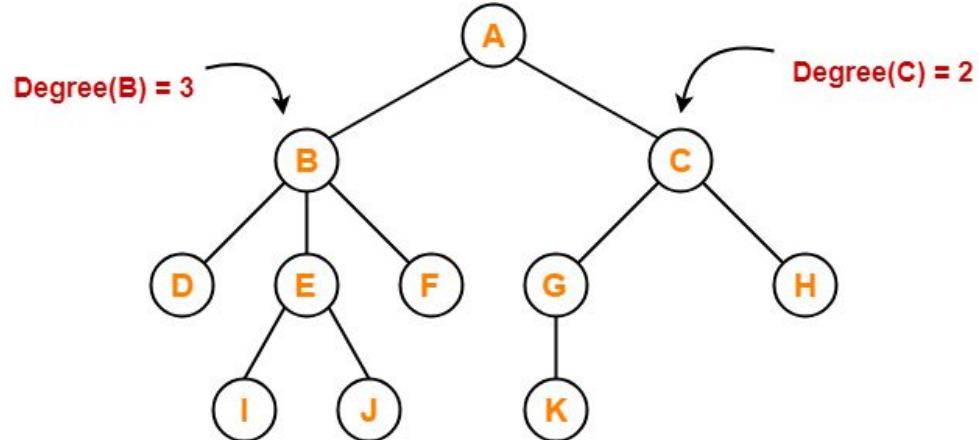
Tree terminology: Siblings

- Nodes which belong to the same parent are called as siblings.
- In other words, nodes with the same parent are sibling nodes.



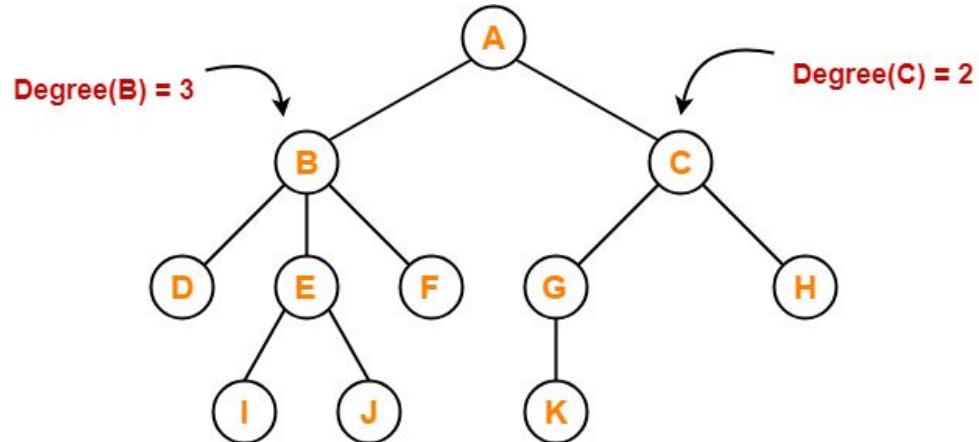
Tree terminology: Degree

- Degree of a node is the total number of children of that node.
- Degree of a tree is the highest degree of a node among all the nodes in the tree.



Tree terminology: Degree contd.

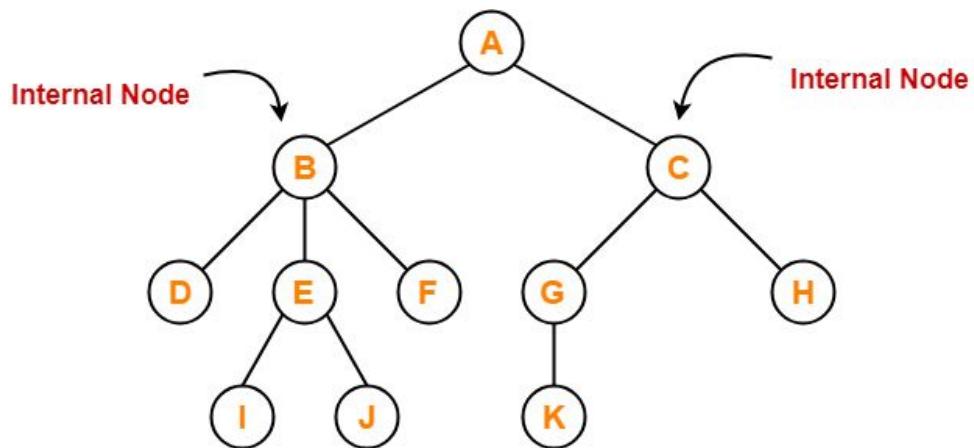
- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0



Degree of the above tree is 3

Tree terminology: Internal nodes

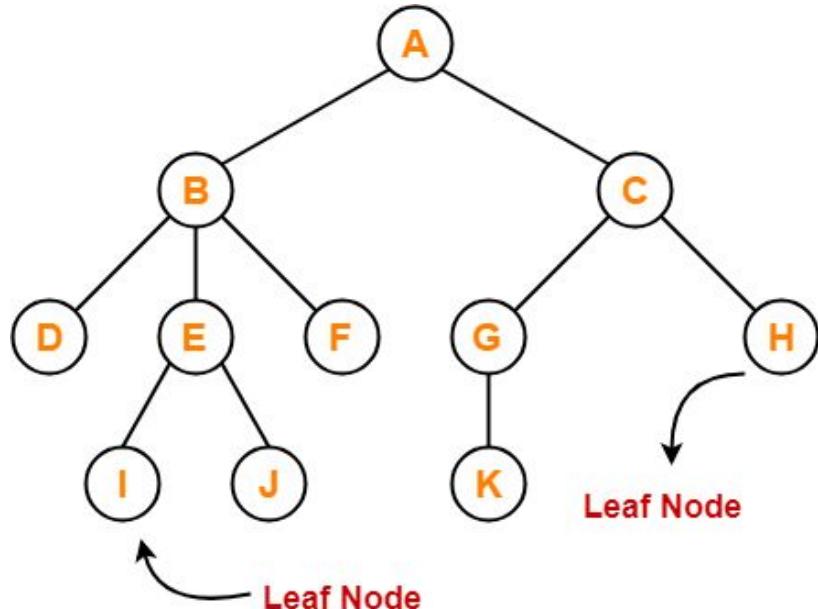
- The node which has at least one child is called as an internal node.
- Internal nodes are also called as non-terminal nodes.
- Every non-leaf node is an internal node.



Here, nodes A, B, C, E and G are internal nodes.

Tree terminology: leaf nodes

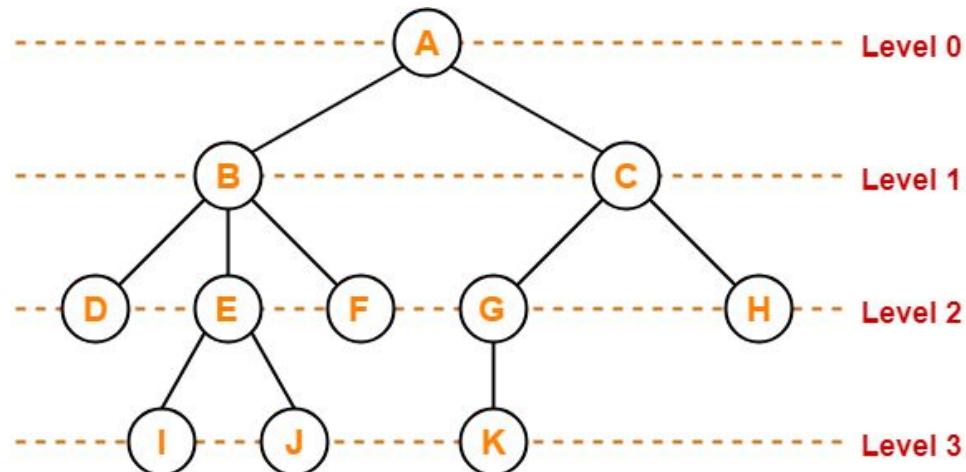
- The node which does not have any child is called as a leaf node.
- Leaf nodes are also called as external nodes or terminal nodes.



Here, nodes D, I, J, F, K and H are leaf nodes.

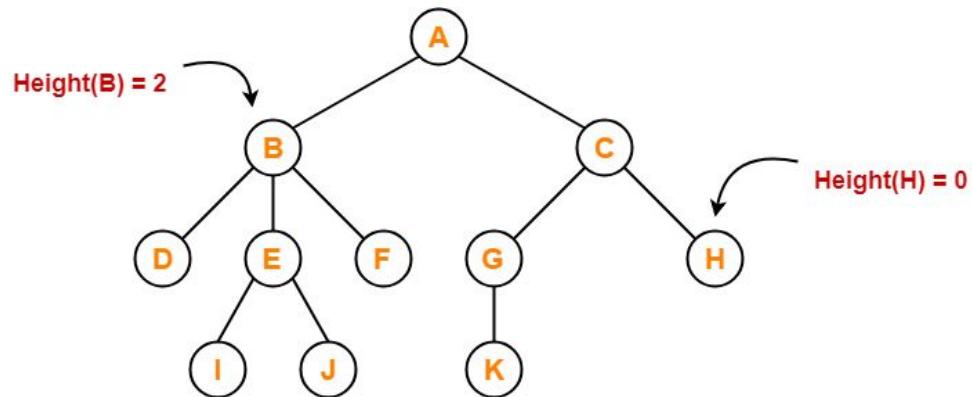
Tree terminology: levels

- In a tree, each step from top to bottom is called as level of a tree.
- The level count starts with 0 and increments by 1 at each level or step.



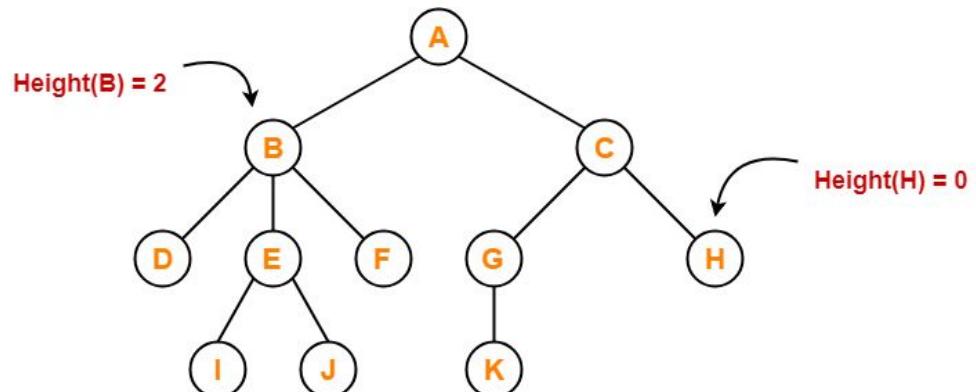
Tree terminology: Height

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.
- Height of a tree is the height of root node.
- Height of all leaf nodes = 0



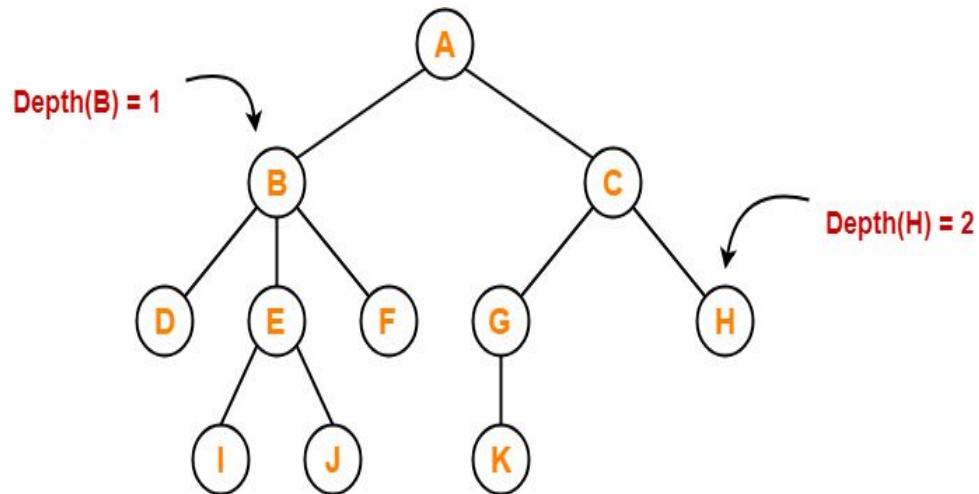
Tree terminology: Height contd..

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0



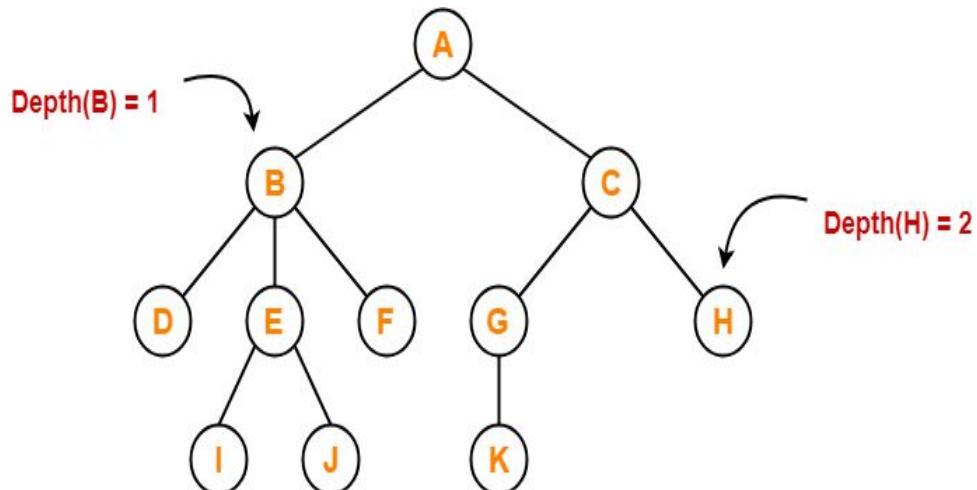
Tree terminology: Depth

- Total number of edges from root node to a particular node is called as depth of that node.
- Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.



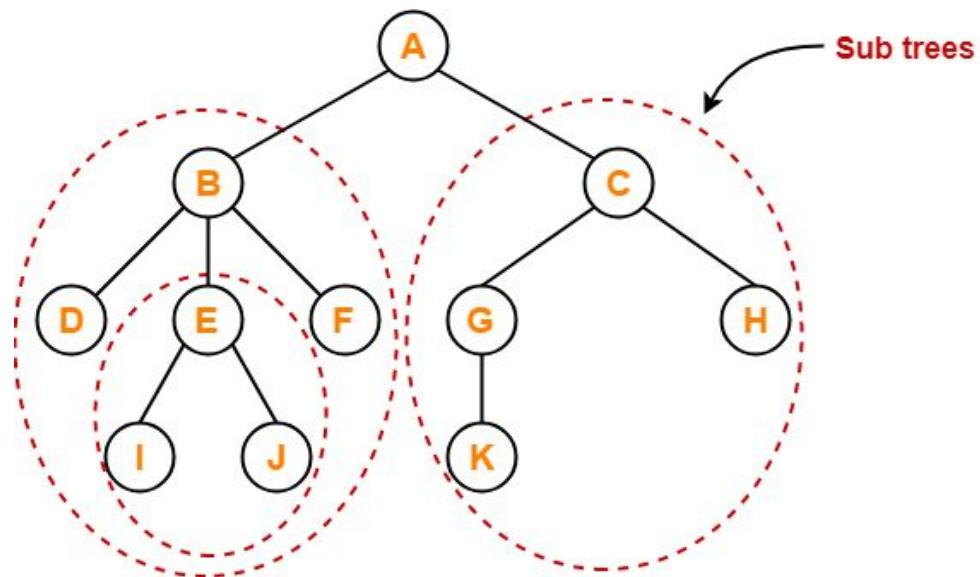
Tree terminology: Depth

- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3



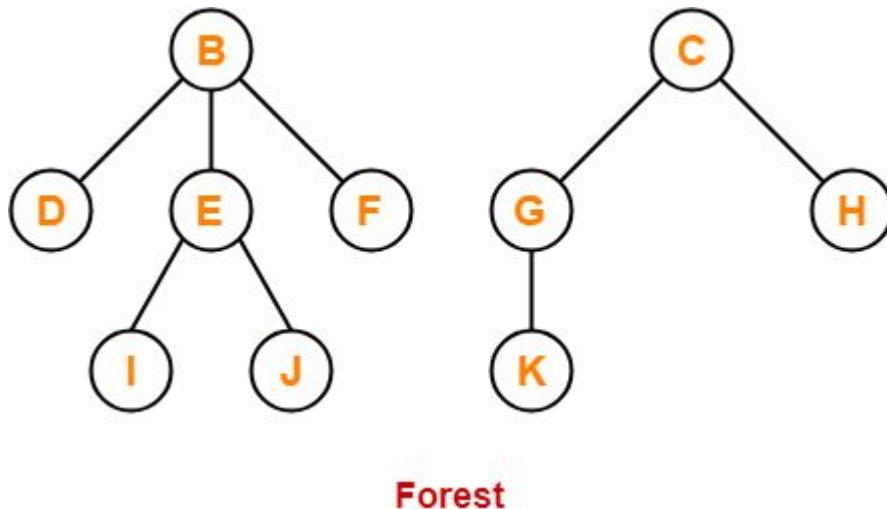
Tree terminology: Subtree

- In a tree, each child from a node forms a subtree recursively.
- Every child node forms a subtree on its parent node.



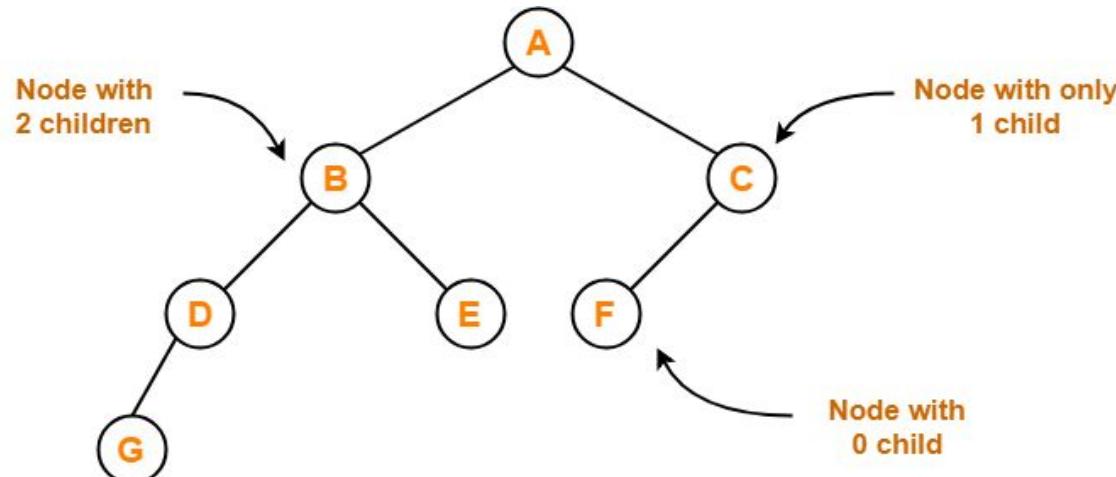
Tree terminology: Forest

- A forest is a set of disjoint trees.



Binary Tree

- Binary tree is a special tree data structure in which each node can have at most 2 children.
- Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children.



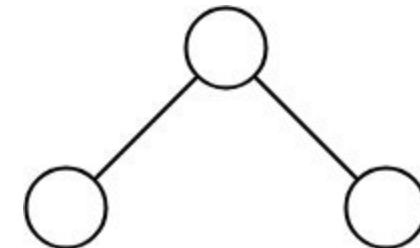
Binary Tree Example

Unlabelled Binary Tree

- A binary tree is unlabeled if its nodes are not assigned any label.

Number of different Binary Trees possible
with 'n' unlabeled nodes

$$= \frac{2^n C_n}{n + 1}$$



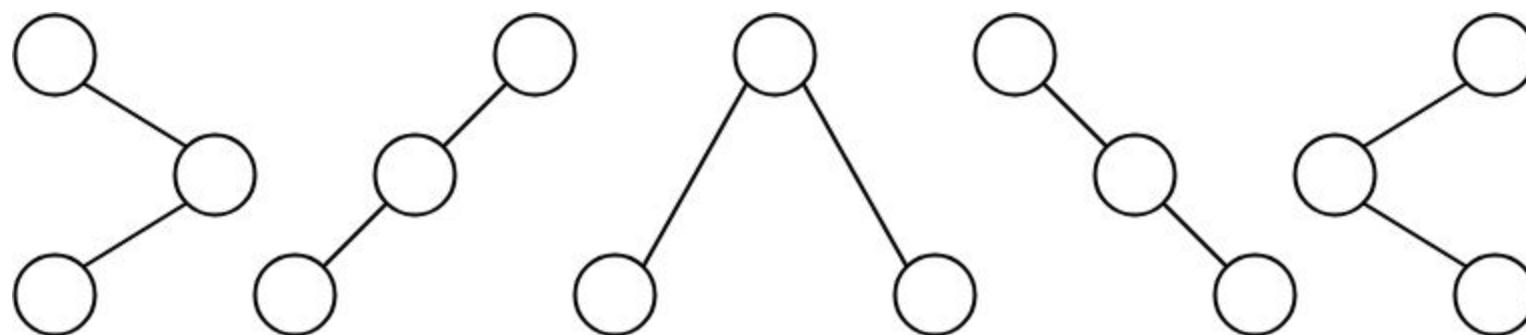
Unlabeled Binary Tree

Consider we want to draw all the binary trees possible with 3 unlabeled nodes.
Using the above formula, we have-

Number of binary trees possible with 3 unlabeled nodes =
 $2 \times 3C3 / (3 + 1) = 6C3 / 4 = 5$

Unlabelled Binary Tree

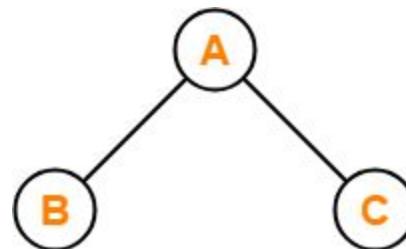
- Thus, With 3 unlabeled nodes, 5 unlabeled binary trees are possible. These unlabeled binary trees are as follows-



Binary Trees Possible With 3 Unlabeled Nodes

Labelled Binary Tree

- A binary tree is labeled if all its nodes are assigned a label.

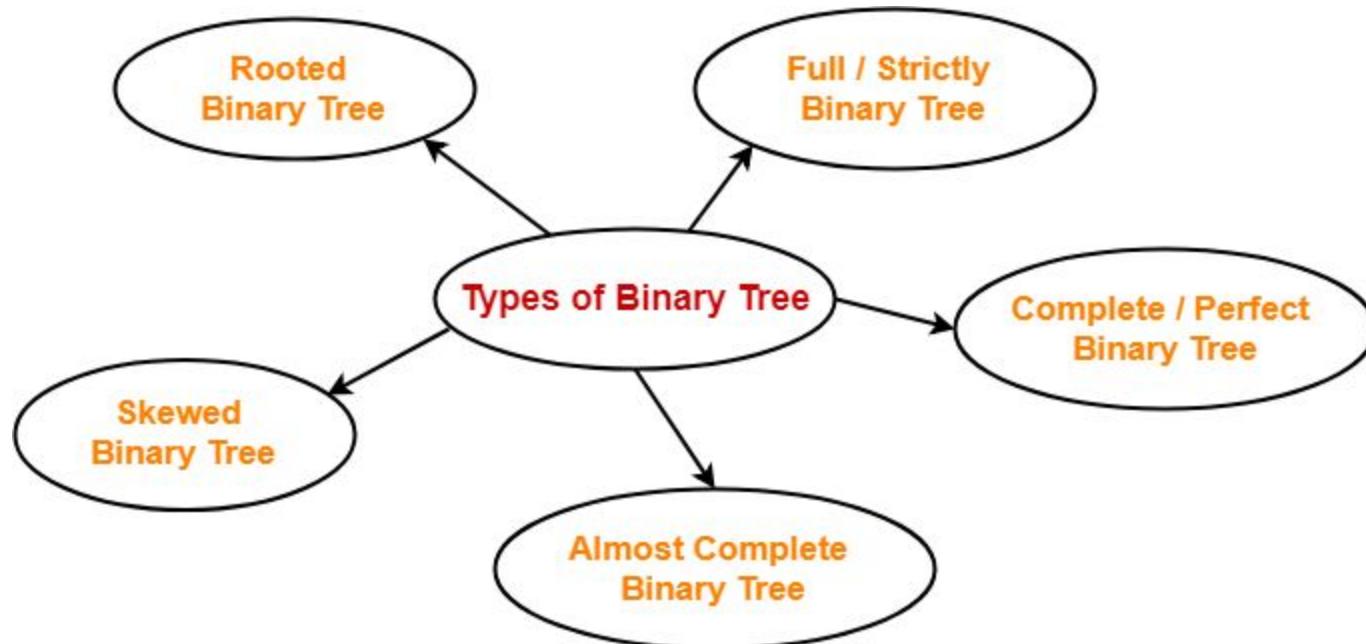


Labeled Binary Tree

Number of different Binary Trees possible
with 'n' labeled nodes

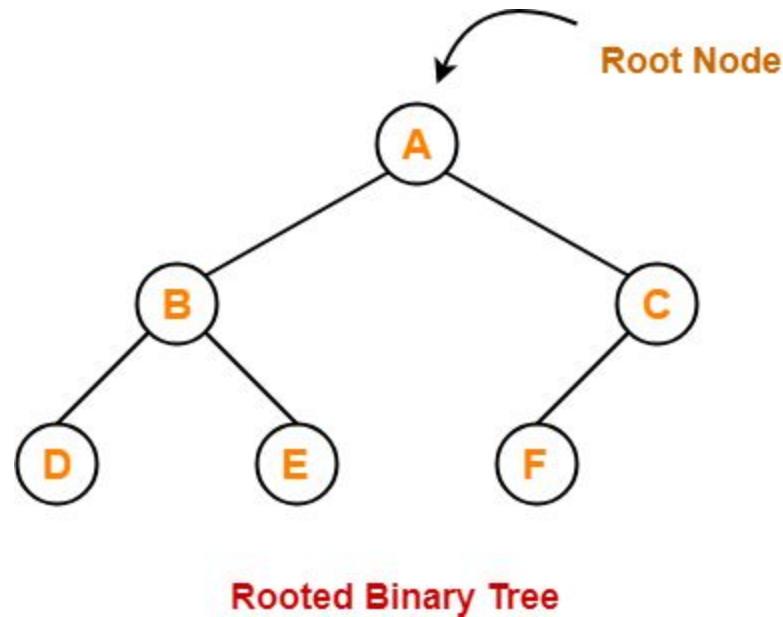
$$= \frac{2^n C_n}{n+1} \times n!$$

Binary Tree: Types



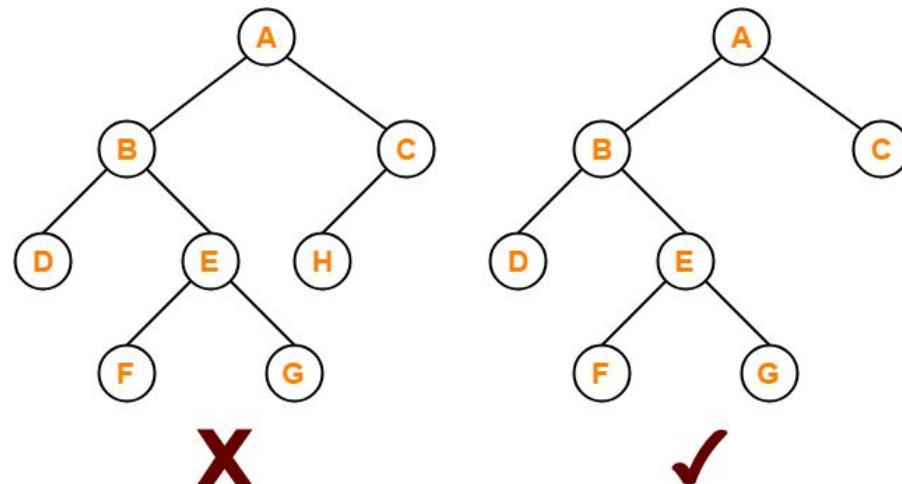
Rooted Binary Tree

- A rooted binary tree is a binary tree that satisfies the following 2 properties-
 - It has a root node.
 - Each node has at most 2 children.



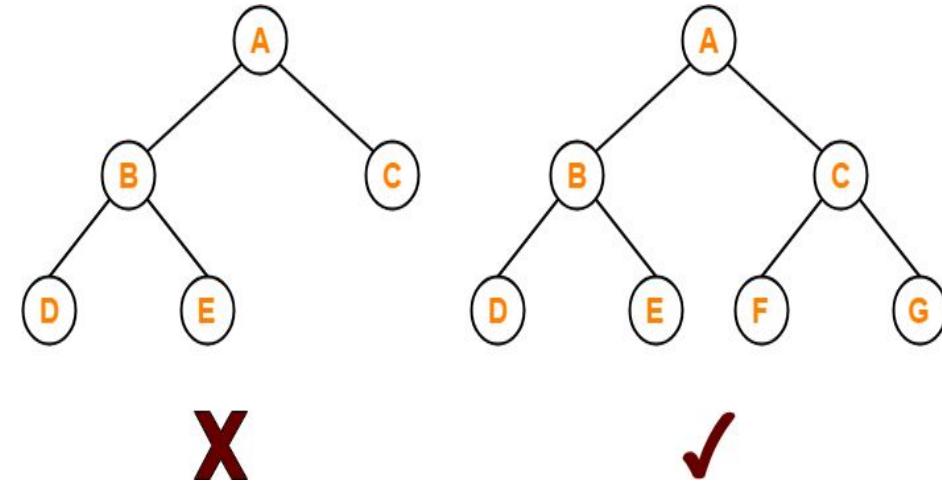
Fully/Strictly Binary Tree

- A binary tree in which every node has either 0 or 2 children is called as a Full binary tree.
- Full binary tree is also called as Strictly binary tree.



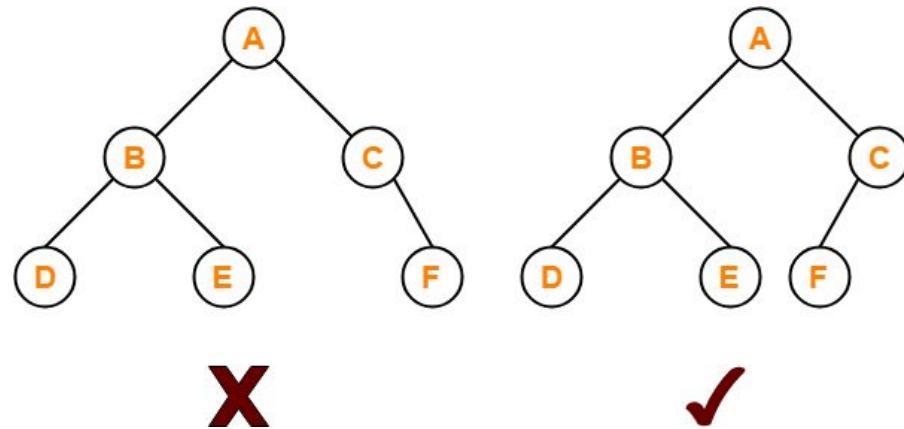
Complete/Perfect Binary Tree

- A complete binary tree is a binary tree that satisfies the following 2 properties-
 - Every internal node has exactly 2 children.
 - All the leaf nodes are at the same level.
 - Complete binary tree is also called as Perfect binary tree.



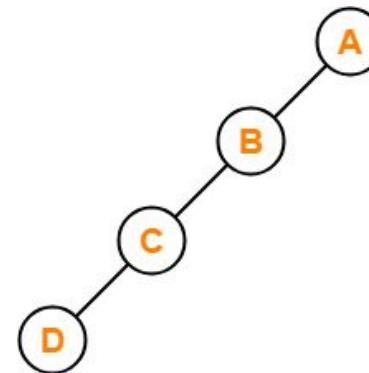
Almost Complete Binary Tree

- An almost complete binary tree is a binary tree that satisfies the following 2 properties-
 - All the levels are completely filled except possibly the last level.
 - The last level must be strictly filled from left to right.

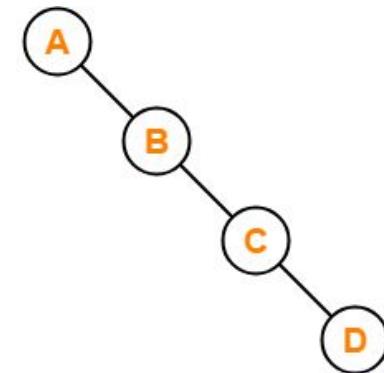


Skewed Binary Tree

- A skewed binary tree is a binary tree that satisfies the following 2 properties-
 - All the nodes except one node has one and only one child.
 - The remaining node has no child.
 - OR, A skewed binary tree is a binary tree of n nodes such that its depth is $(n-1)$.



Left Skewed Binary Tree

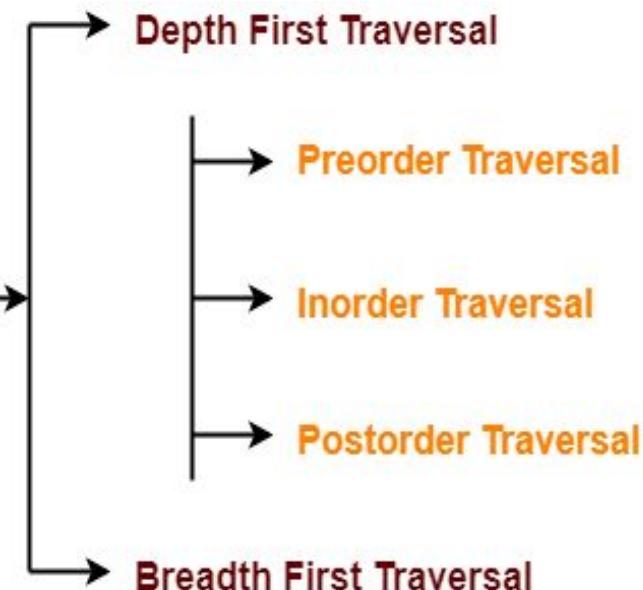


Right Skewed Binary Tree

Binary Tree: Traversal

- Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.
- Various tree traversal techniques are-

Tree Traversal Techniques



Binary Tree: Depth First Traversal

- Following three traversal techniques fall under Depth First Traversal-
 - Preorder Traversal
 - Inorder Traversal
 - Postorder Traversal

Depth First Traversal: Preorder

Algorithm-

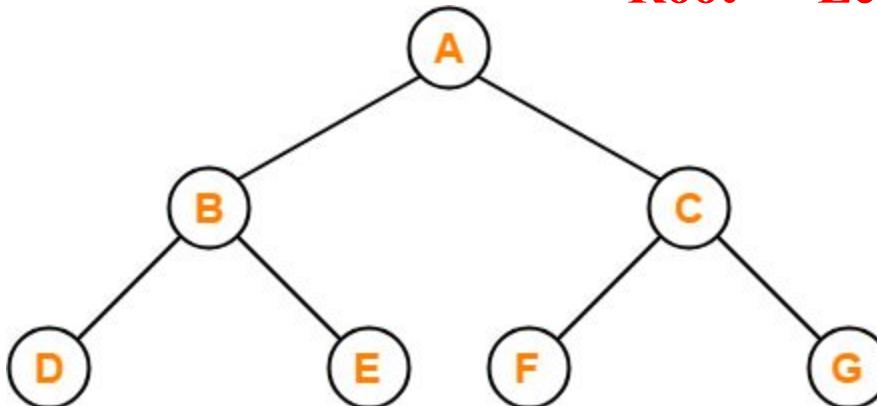
- Visit the root
- Traverse the left sub tree i.e. call Preorder (left subtree)
- Traverse the right subtree i.e. call Preorder (right subtree)

Root → Left → Right

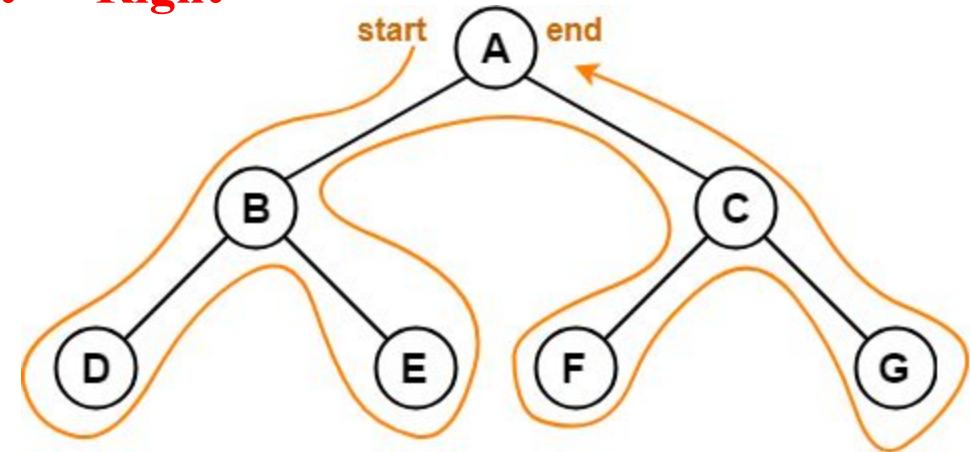
Depth First Traversal: Preorder

- Traverse the entire tree starting from the root node keeping yourself to the left.

Root → Left → Right



Preorder Traversal : A , B , D , E , C , F , G



Preorder Traversal : A , B , D , E , C , F , G

Depth First Traversal: Application

- Preorder traversal is used to get prefix expression of an expression tree.
- Preorder traversal is used to create a copy of the tree.

Depth First Traversal: Inorder

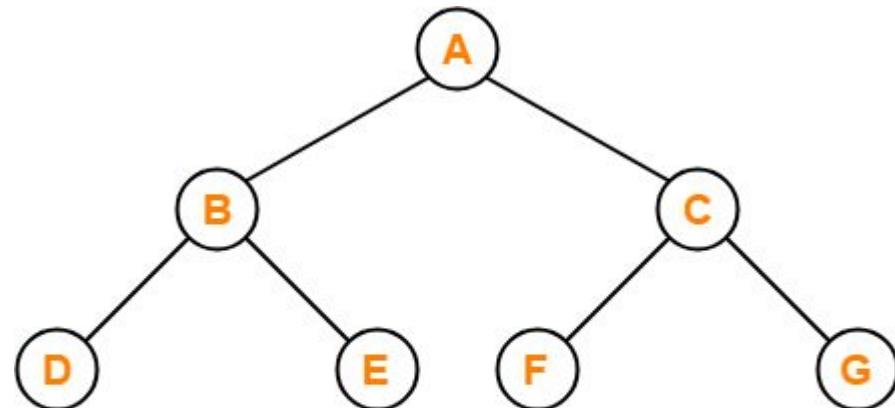
Algorithm-

- Traverse the left subtree i.e. call Inorder (left subtree)
- Visit the root
- Traverse the right subtree i.e. call Inorder (right subtree)

Left → Root → Right

Depth First Traversal: Inorder

- Traverse the entire tree starting from the root node keeping yourself to the left.
- Inorder traversal is used to get infix expression of an expression tree.



Inorder Traversal : D , B , E , A , F , C , G

Left → Root → Right

Depth First Traversal: Postorder

Algorithm-

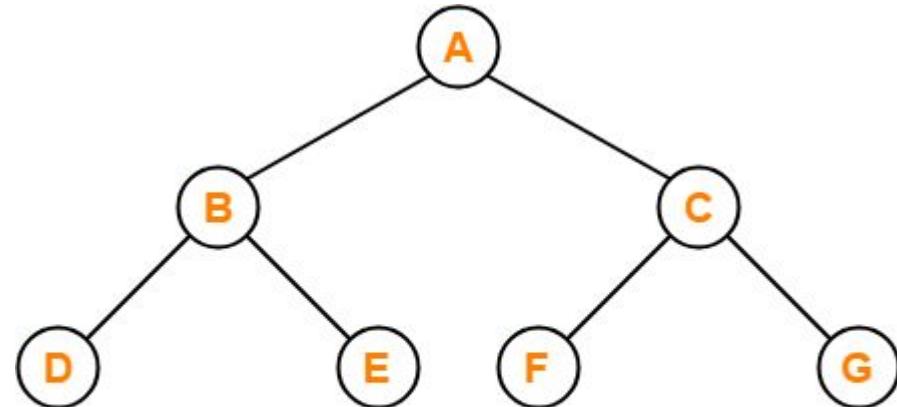
- Traverse the left subtree i.e. call Postorder (left subtree)
- Traverse the rightsub tree i.e. call Postorder (right subtree)
- Visit the root

Left → Right → Root

Depth First Traversal: Postorder

Left → Right→Root

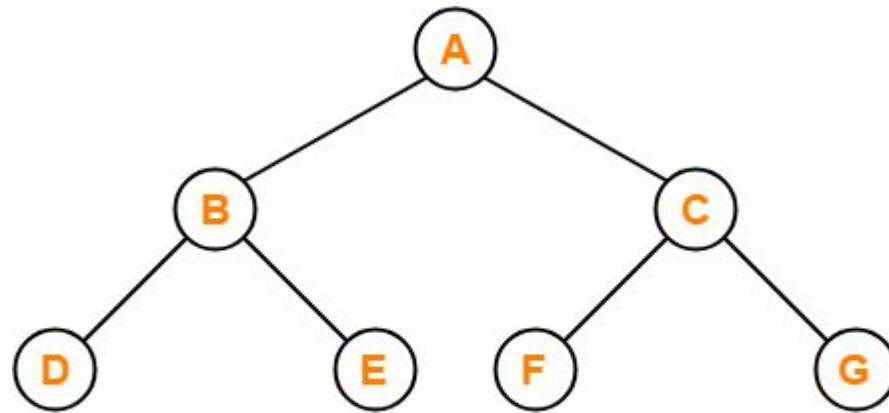
- Postorder traversal is used to get postfix expression of an expression tree.
- Postorder traversal is used to delete the tree.
- This is because it deletes the children first and then it deletes the parent.



Postorder Traversal : D , E , B , F , G , C , A

Breadth First Traversal

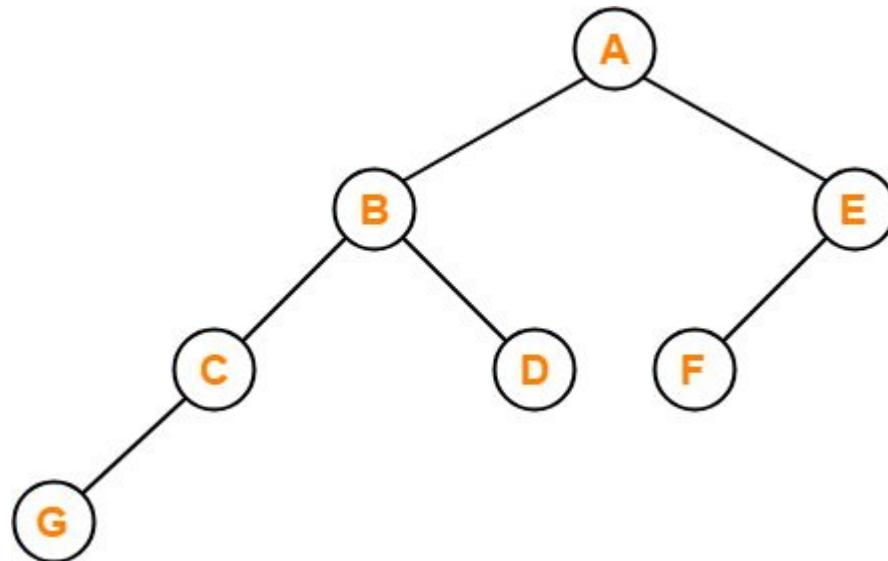
- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as Level Order Traversal.
- Level order traversal is used to print the data in the same order as stored in the array representation of a complete binary tree.



Level Order Traversal : A , B , C , D , E , F , G

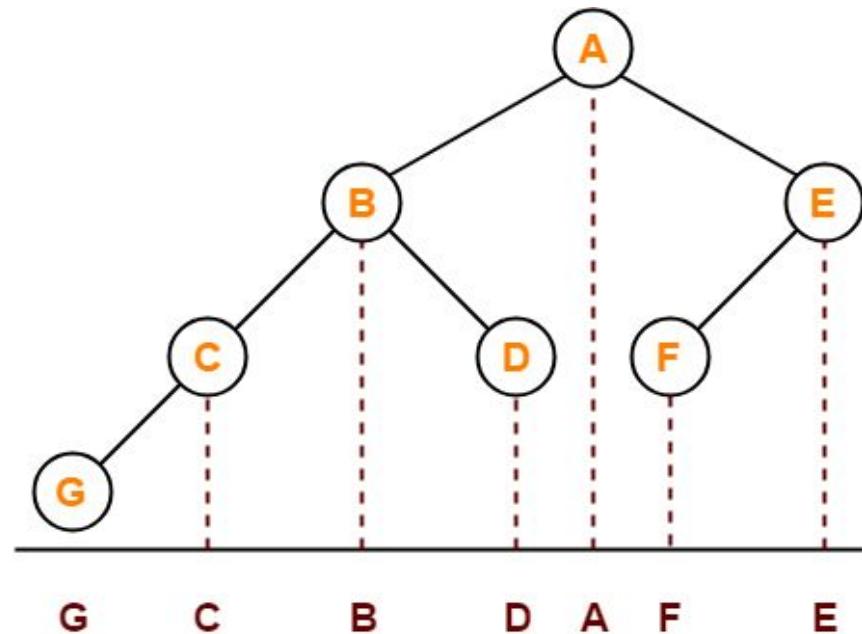
Traversal: Exercise 1

- Perform the inorder traversal for the given tree.



Traversal: Exercise 1: solution

- Perform the inorder traversal for the given tree.



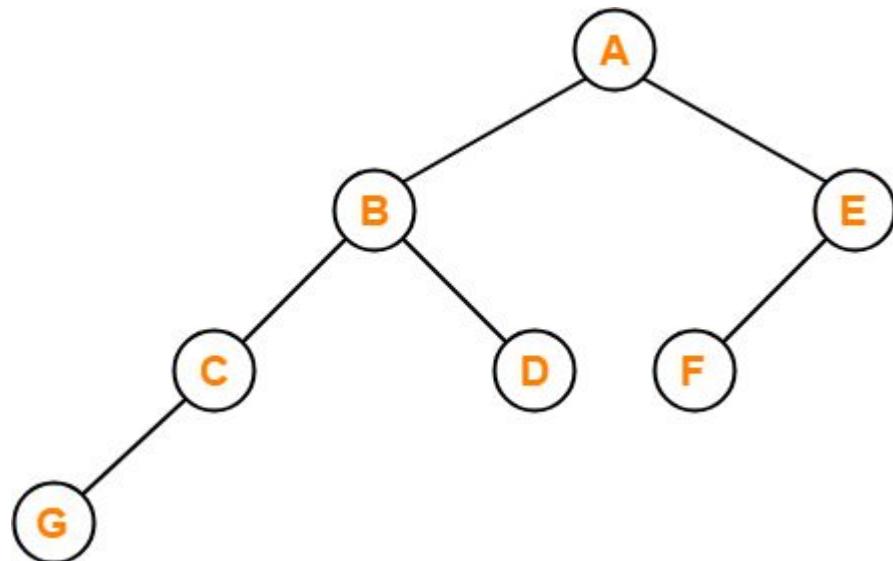
Inorder Traversal : G , C , B , D , A , F , E

Traversal: Exercise 2

- Perform the postorder traversal for the given tree.
- Solution:

Postorder Traversal :

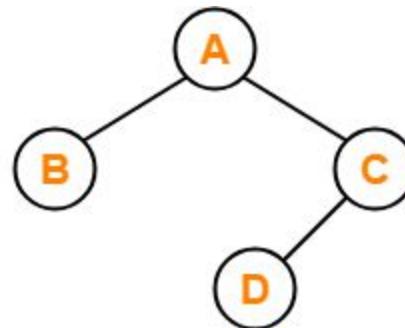
G , C , D , B , F , E , A



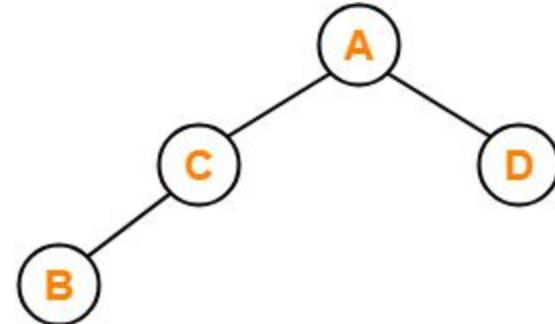
Traversal: Exercise 3

- Which of the following binary trees has its inorder and preorder traversals as BCAD and ABCD respectively

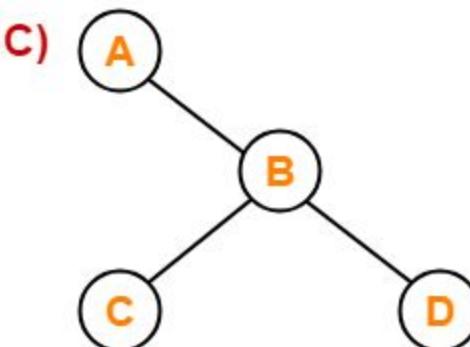
A)



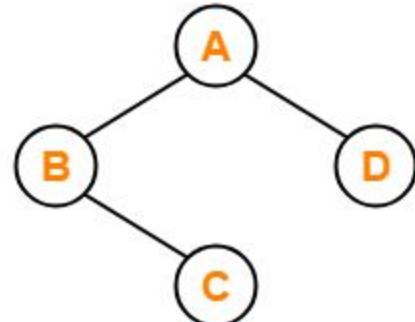
B)



C)



D)



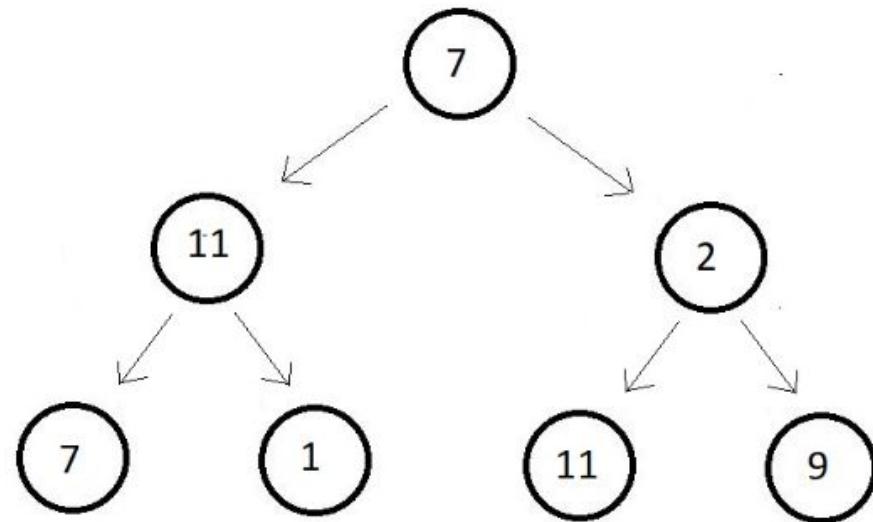
Binary Tree representation

- A binary tree data structure is represented using two methods.
- Those methods are as follows...

Array Representation

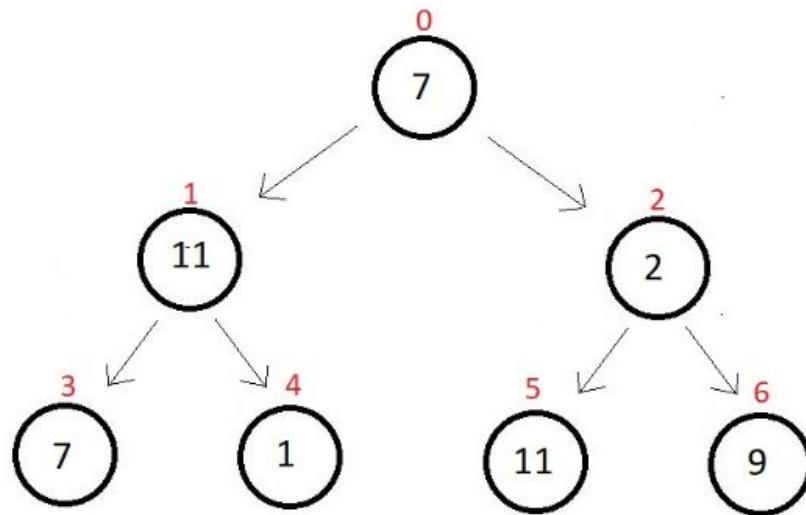
Linked List Representation

- Suppose we have a binary tree with 7 nodes.



Binary Tree representation

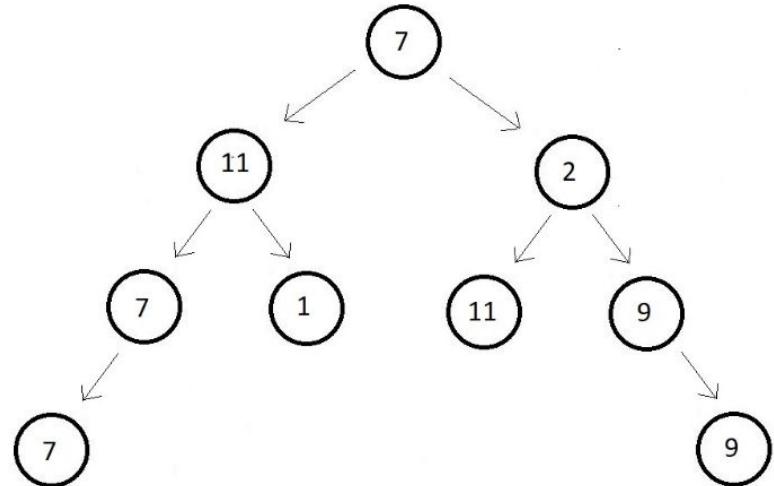
- we traverse each level starting from the root node and from left to right and mark them with the indices these nodes would belong to.
- And now we can simply make an array of length 7 and store these elements at their corresponding indices.



0	1	2	3	4	5	6
7	11	2	7	1	11	9

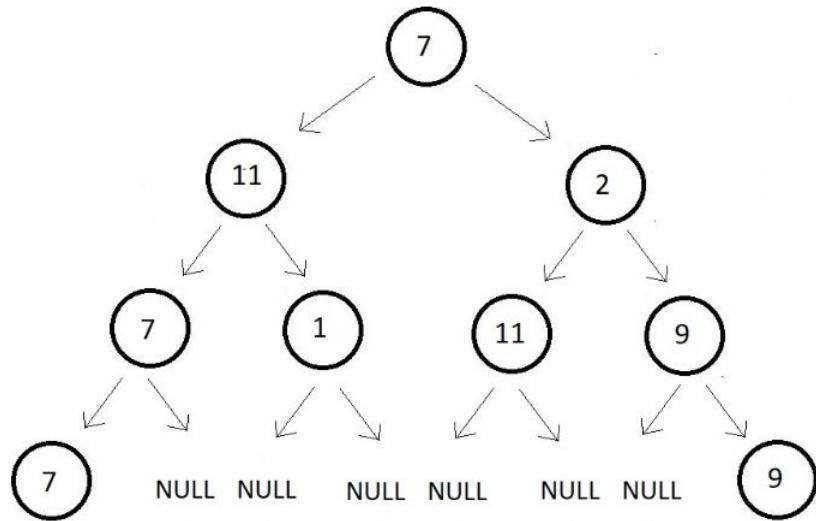
Binary Tree representation

- But, how do we store the nodes of this binary tree?
- Here, while traversing we get stuck at the 8th index.
- We don't know if declaring the last node as the 8th index element makes it a general representation of the tree or not.
- So, we simply make the tree perfect ourselves. We first assume the remaining vacant places to be NULL.



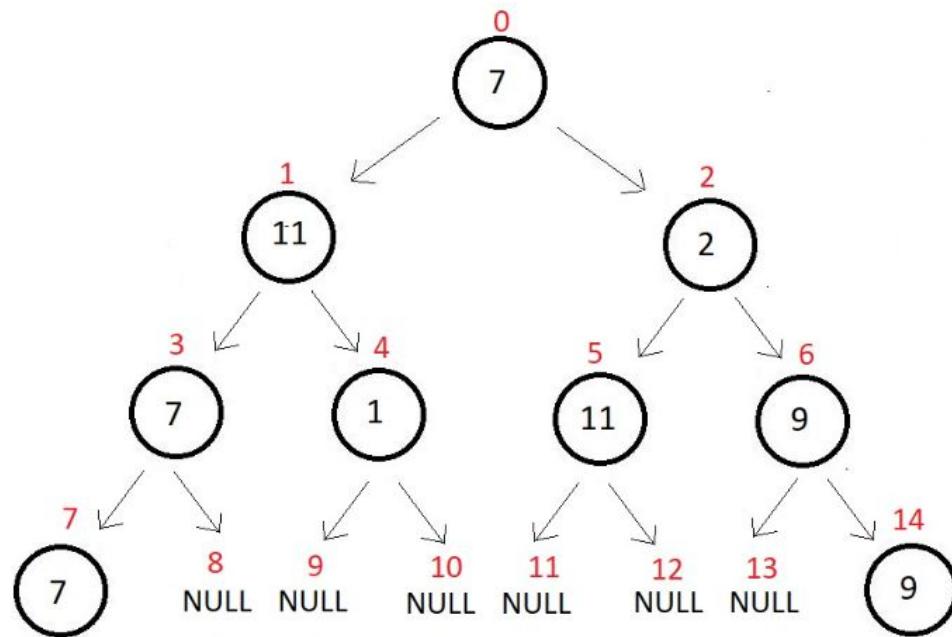
Binary Tree representation

- But, how do we store the nodes of this binary tree?
- Here, while traversing we get stuck at the 8th index.
- We don't know if declaring the last node as the 8th index element makes it a general representation of the tree or not.
- So, we simply make the tree perfect ourselves. We first assume the remaining vacant places to be NULL.



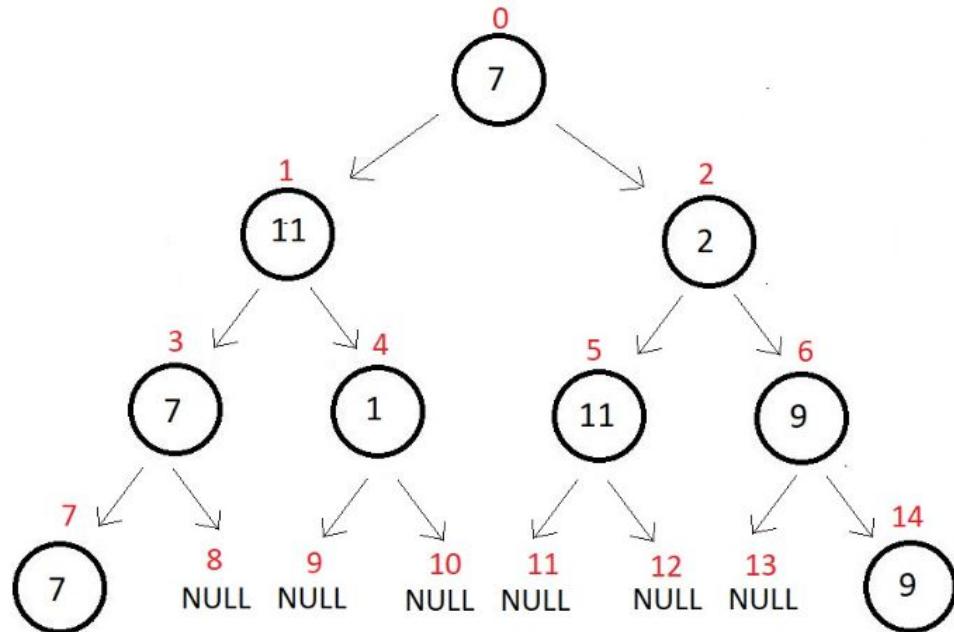
Binary Tree representation

- And now we can easily mark their indices from 0 to 14.



Binary Tree representation

- And the array representation of the tree looks something like this. It is an array of length 15.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	11	2	7	1	11	9	7							9

Binary Tree representation: using array

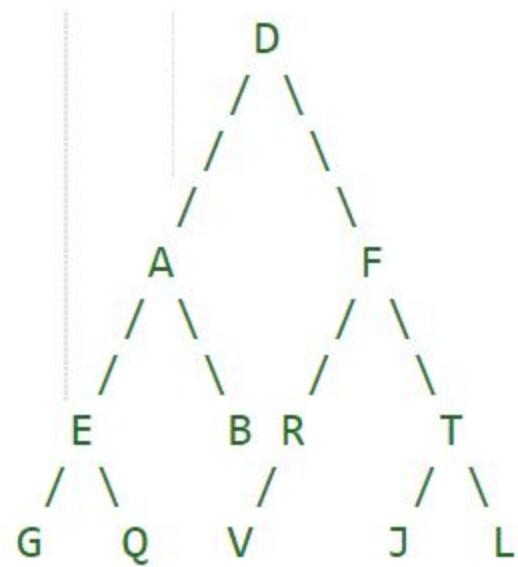
- What we do now is assign each of nodes to a specific position in the array.
- This could be done any way you like, but a particular easy and useful way is:
 - root of the tree (A): array position 1
 - root's left child (B): array position 2
 - root's right child (C): array position 3
 - ...
 - left child of node in array position K: array position $2K$
 - right child of node in array position K: array position $2K+1$

Binary Tree representation: using array

- An array can be converted into a binary tree.
 - **Parent** : Parent of a node at index lies at $(n-1)/2$ except the root node.
 - **Left Child** : Left child of a node at index n lies at $(2*n+1)$.
 - **Right Child** : Right child of a node at index n lies at $(2*n+2)$.
 - **Left Sibling** : left Sibling of a node at index n lies at $(n-1)$.
 - **Right Sibling** : Right sibling of a node at index n lies at $(n+1)$.

Binary Tree implementation: using array

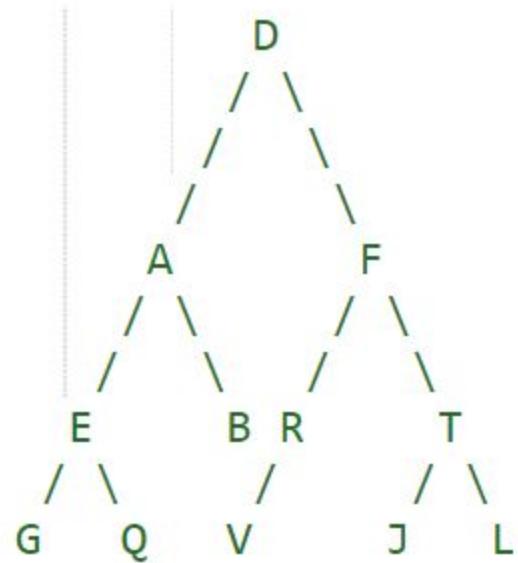
- Consider we need to create the following tree:



Binary Tree implementation: using array

- Consider we need to create the following tree:

Initialize variable for complete binary tree and declare an array **tree[]**.



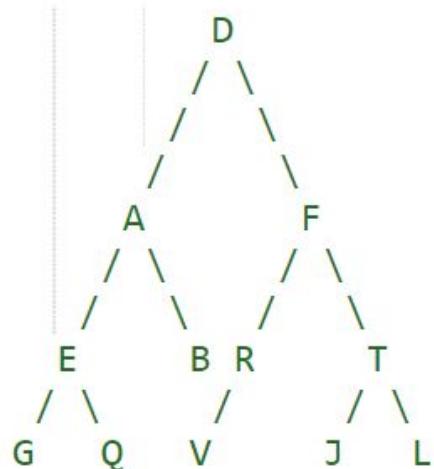
```
// variable to store maximum number of nodes  
int complete_node = 15;
```

```
// array to store the tree  
char tree[] = {'\0', 'D', 'A', 'F', 'E', 'B', 'R', 'T', 'G', 'Q', '\0', '\0', 'V', '\0', 'J', 'L'};
```

Binary Tree implementation: using array

- Consider we need to create the following tree:

Write function for right child

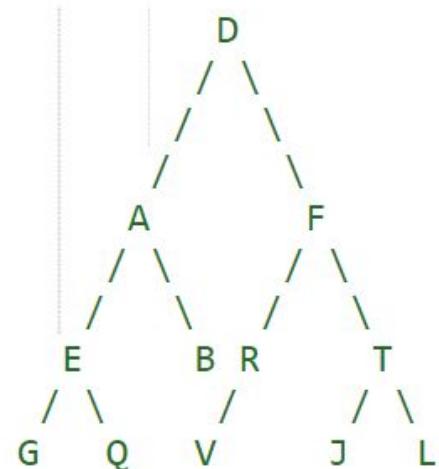


```
int get_right_child(int index)
{
    // node is not null
    // and the result must lie within the number of nodes for a complete binary tree
    if(tree[index]!='\0' && ((2*index)+1)<=complete_node)
        return (2*index)+1;
    // right child doesn't exist
    return -1;
}
```

Binary Tree implementation: using array

- Consider we need to create the following tree:

Write function for left child



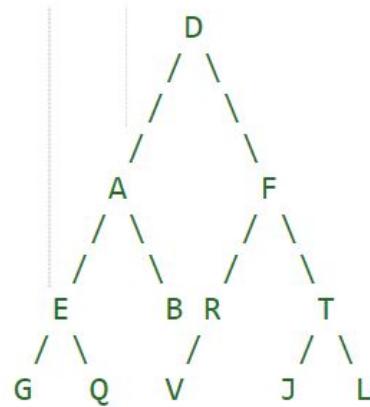
```
int get_left_child(int index)
{
    // node is not null
    // and the result must lie within the number of nodes for a complete binary tree
    if(tree[index]!='\0' && (2*index)<=complete_node)
        return 2*index;
    // left child doesn't exist
    return -1;
}
```

Binary Tree implementation: using array

- Consider we need to create the following tree:

Write function for preorder traversal

```
void preorder(int index)
{
    // checking for valid index and null node
    if(index>0 && tree[index]!='\0')
    {
        printf(" %c ",tree[index]); // visiting root
        preorder(get_left_child(index)); //visiting left subtree
        preorder(get_right_child(index)); //visiting right subtree
    }
}
```

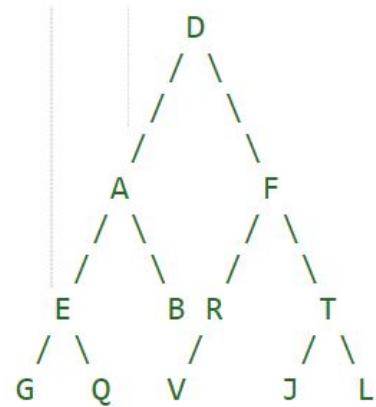


Binary Tree implementation: using array

- Consider we need to create the following tree:

Write function for postorder traversal

```
void postorder(int index)
{
    // checking for valid index and null node
    if(index>0 && tree[index]!='\0')
    {
        postorder(get_left_child(index)); //visiting left subtree
        postorder(get_right_child(index)); //visiting right subtree
        printf(" %c ",tree[index]); //visiting root
    }
}
```

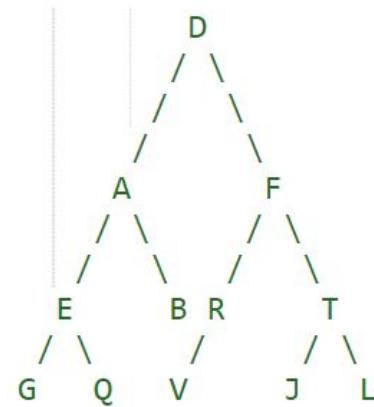


Binary Tree implementation: using array

- Consider we need to create the following tree:

Write function for inorder traversal

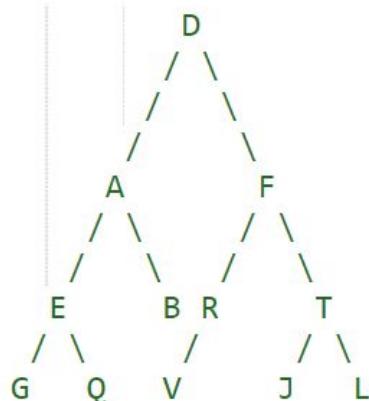
```
void inorder(int index)
{
    // checking for valid index and null node
    if(index>0 && tree[index]!='\0')
    {
        inorder(get_left_child(index)); //visiting left subtree
        printf(" %c ",tree[index]); //visiting root
        inorder(get_right_child(index)); // visiting right subtree
    }
}
```



Binary Tree implementation: using array

- Consider we need to create the following tree:

Write function for main():

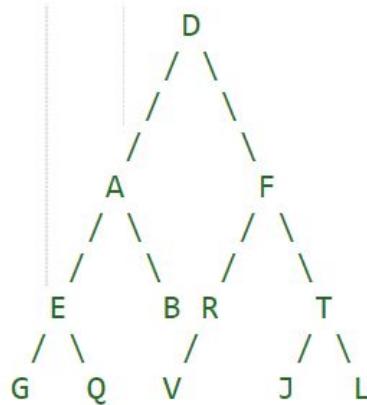


```
int main()
{
    printf("Preorder:\n");
    preorder(1);
    printf("\nPostorder:\n");
    postorder(1);
    printf("\nInorder:\n");
    inorder(1);
    printf("\n");
    return 0;
}
```

Binary Tree implementation: using array

- Consider we need to create the following tree:

Write function for main():



Preorder: DAEGQBFRVTJL

Postorder: GQEBAVRJLTFD

Inorder: GEQABDVRFJTL

Binary Tree implementation: using structure and array

```
#include<stdio.h>

typedef struct node
{
    struct node*left;
    struct node*right;
    char data;
}node;

void main()
{
    node*tree=NULL;
    char c[]={ 'A','B','C','D','E','F','\0',
               'G','\0','\0','\0','\0','\0','\0','\0','\0','\0',
               '\0','\0','\0','\0','\0','\0','\0'};

    tree=insert(c,0);
    inorder(tree);
}
```

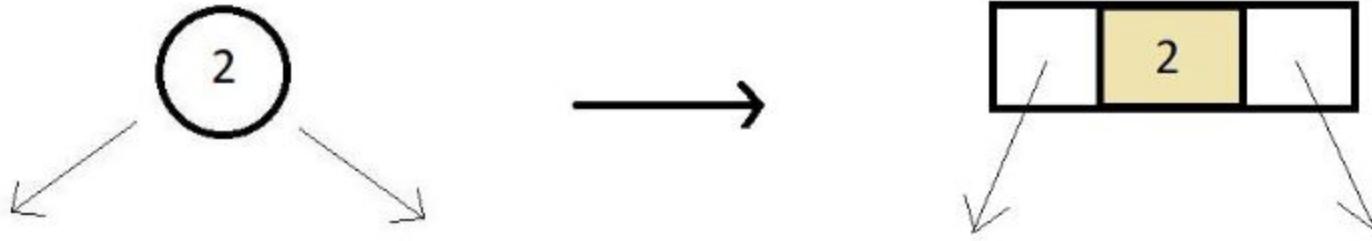
Binary Tree implementation: using structure and array

```
node* insert(char c[],int n)
{ node*tree=NULL;
if(c[n]!='\0')
{
    tree=(node*)malloc(sizeof(node));
    tree->left=insert(c,2*n+1);
    tree->data=c[n];
    tree->right=insert(c,2*n+2);
}
return tree;
}
```

```
//traverse the tree in inorder
void inorder(node*tree)
{
    if(tree!=NULL)
    {
        inorder(tree->left);
        printf("%c\t",tree->data);
        inorder(tree->right);
    }
}
```

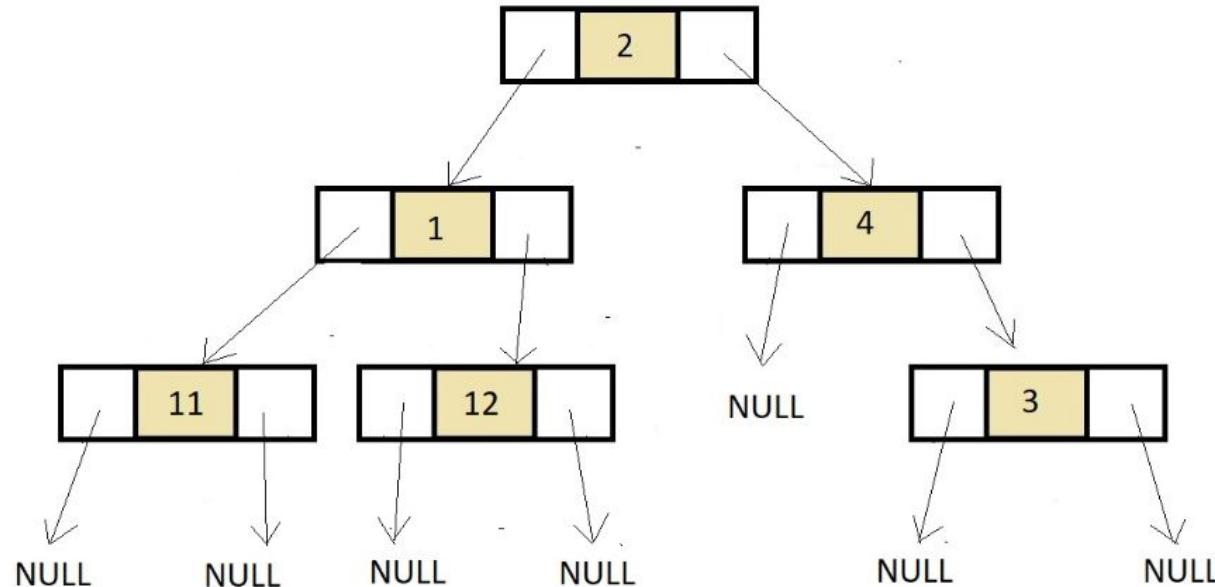
Binary Tree implementation: using linked list

- **Binary tree using doubly linked list:**
 - The below representation of a node here in the linked representation of a binary tree.



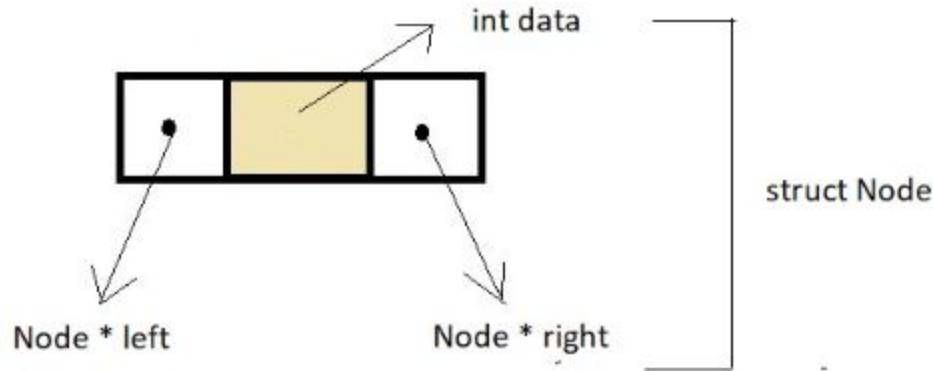
Binary Tree implementation: using linked list

- **Binary tree using doubly linked list:**
 - Other nodes will be added in a similar fashion



Binary Tree implementation: using linked list

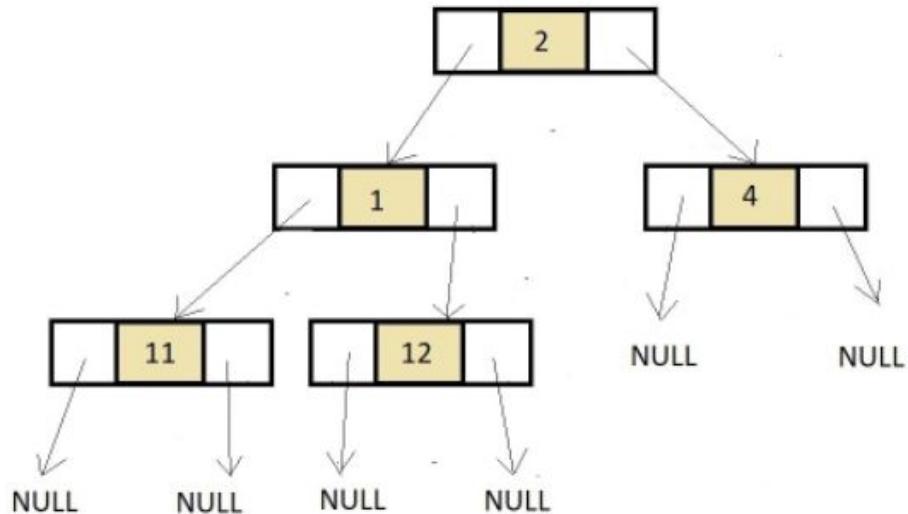
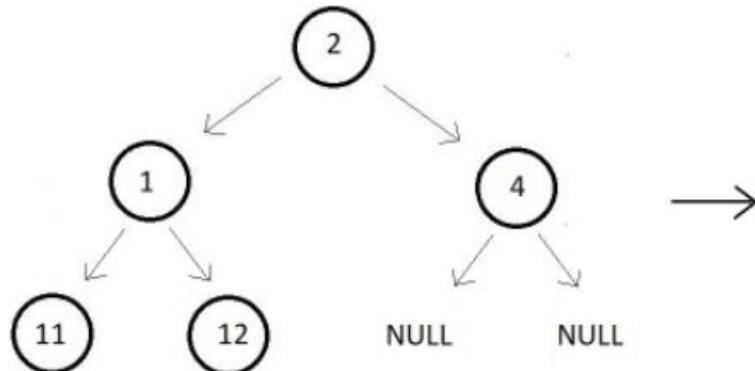
- Every tree node is represented as a doubly linked list and can be implemented using a structure as follows:
- The left pointer of this node points to the left child and the right pointer points to the right child, and if there is no left or right child, we represent that using a NULL.



```
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
};
```

Binary Tree implementation: using linked list

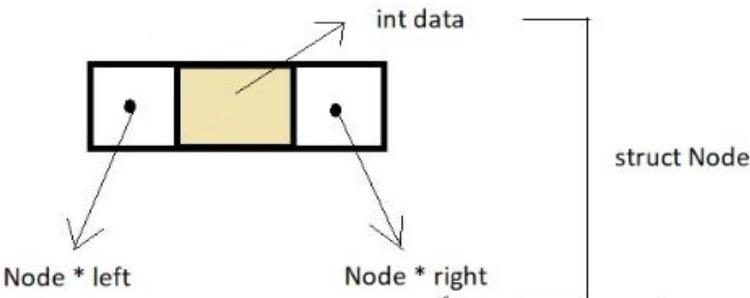
See how a tree looks in real life, and how efficiently the linked representation of a tree helps us visualize the same.



using linked list

Create root node:

```
// Constructing the root node
struct node *p;
p = (struct node *) malloc(sizeof(struct node));
p->data = 2;
p->left = NULL;
p->right = NULL;
```

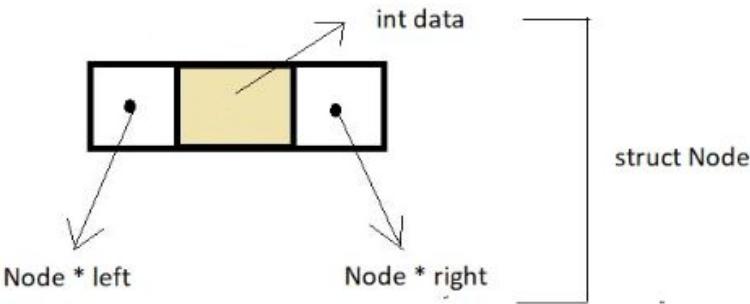


```
struct node{
    int data;
    struct node* left;
    struct node* right;
};
```

using linked list

Create the second node:

```
// Constructing the second node
struct node *p1;
p1 = (struct node *) malloc(sizeof(struct node));
p->data = 1;
p1->left = NULL;
p1->right = NULL;
```

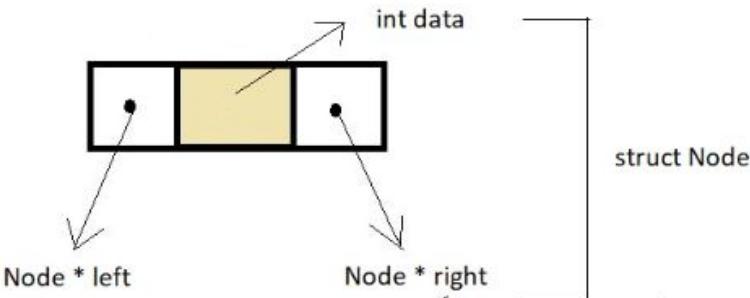


```
struct node{
    int data;
    struct node* left;
    struct node* right;
};
```

using linked list

Create the third node:

```
// Constructing the third node
struct node *p2;
p2 = (struct node *) malloc(sizeof(struct node));
p->data = 4;
p2->left = NULL;
p2->right = NULL;
*/
```



```
struct node{
    int data;
    struct node* left;
    struct node* right;
};
```

using linked list

Linking nodes

```
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
};
```

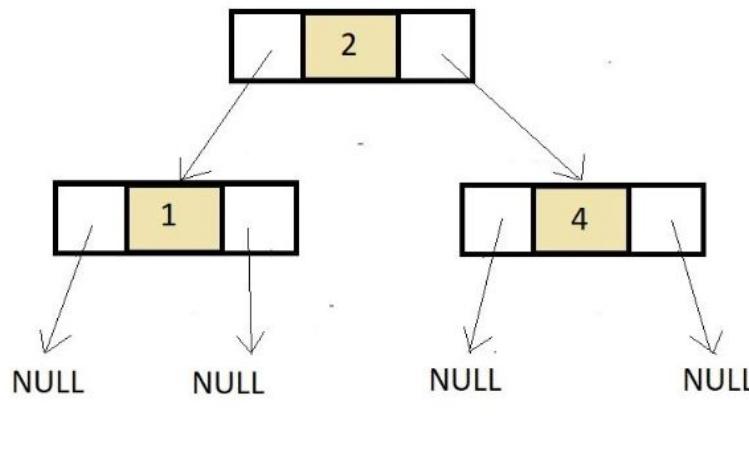
```
int main(){  
  
    struct node *p;  
    p = (struct node *) malloc(sizeof(struct node));  
    p->data = 2;  
    p->left = NULL;  
    p->right = NULL;  
  
    struct node *p1;  
    p1 = (struct node *) malloc(sizeof(struct node));  
    p1->data = 1;  
    p1->left = NULL;  
    p1->right = NULL;  
  
    // Constructing the third node  
    struct node *p2;  
    p2 = (struct node *) malloc(sizeof(struct node));  
    p2->data = 4;  
    p2->left = NULL;  
    p2->right = NULL;  
*/  
// Linking the root node with left and right children  
p->left = p1;  
p->right = p2;  
return 0;  
}
```

using linked list

Creating and Linking nodes
using functions:

```
#Create the root node
```

```
struct node* createNode(int data){  
    struct node *n;  
    n = (struct node *) malloc(sizeof(struct node));  
    n->data = data;  
    n->left = NULL;  
    n->right = NULL;  
    return n;  
}  
  
int main(){
```



```
struct node *p = createNode(2);  
struct node *p1 = createNode(1);  
struct node *p2 = createNode(4);  
  
p->left = p1;  
p->right = p2;  
return 0;
```

Create tree from Postorder and Inorder

Consider the postorder and inorder traversal of a tree is given as:

Inorder Traversal : { 4, 2, 1, 7, 5, 8, 3, 6 }

Postorder Traversal : { 4, 2, 7, 8, 5, 6, 3, 1 }

Construct a tree.

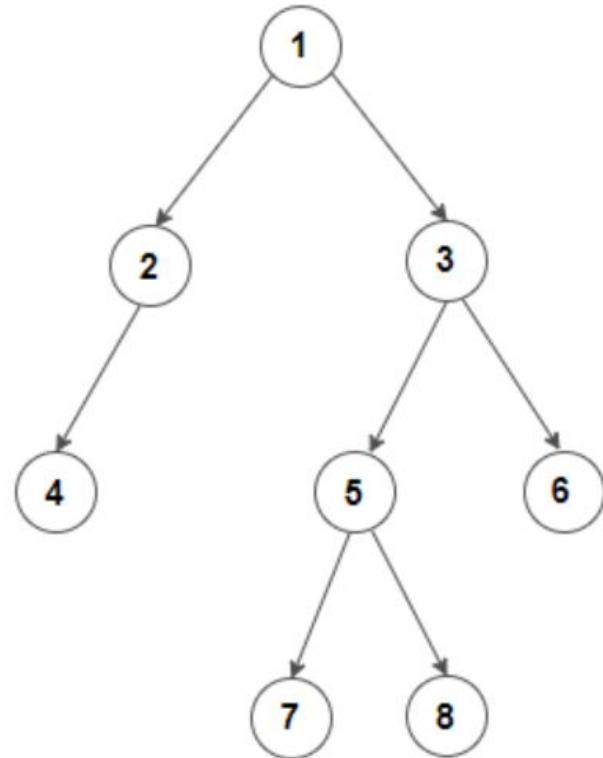
Create tree from Postorder and Inorder

Consider the postorder and inorder traversal of a tree is given as:

Inorder Traversal : { 4, 2, 1, 7, 5, 8, 3, 6 }

Postorder Traversal : { 4, 2, 7, 8, 5, 6, 3, 1 }

Construct a tree.



Create tree from Postorder and Inorder

Consider the postorder and inorder traversal of a tree is given as:

Inorder Traversal : { 9,5,1,7,2,12,8,4,3,11}

Postorder Traversal : { 9,1,2,12,7,5,3,11,4,8}

Construct a tree.

Create tree from Preorder and Inorder

Consider the preorder and inorder traversal of a tree is given as:

Inorder Traversal : { 4, 2, 1, 7, 5, 8, 3, 6 }

Preorder Traversal : { 1, 2, 4, 3, 5, 7, 8, 6 }

Construct a tree.

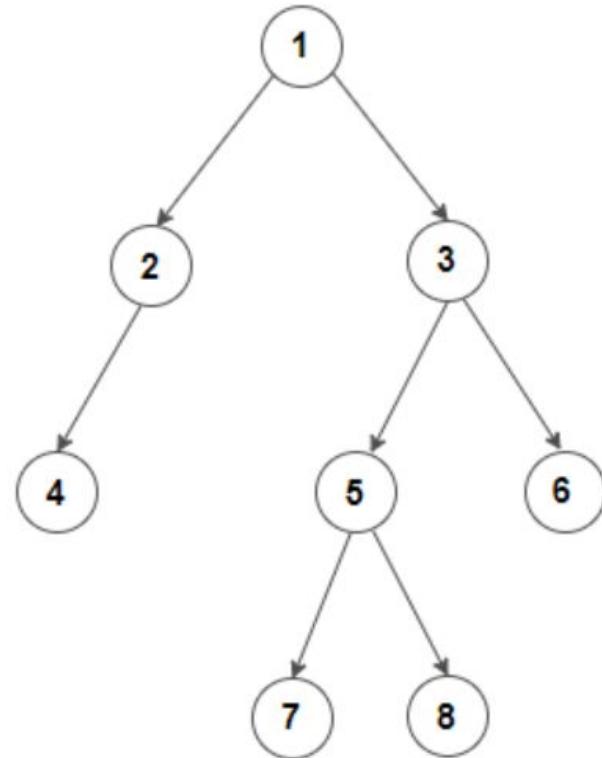
Create tree from Preorder and Inorder

Consider the preorder and inorder traversal of a tree is given as:

Inorder Traversal : { 4, 2, 1, 7, 5, 8, 3, 6 }

Preorder Traversal : { 1, 2, 4, 3, 5, 7, 8, 6 }

Construct a tree.



Create tree from Preorder and Inorder

Consider the preorder and inorder traversal of a tree is given as:

Inorder Traversal : { 8,4,10,9,11,2,5,1, 6,3,7}

Preorder Traversal : { 1, 2, 4, 8,9,10,11,5,3,6,7}

Construct a tree.

Inorder traversal using recursion: Revision

```
#include<stdio.h>

typedef struct node
{
    struct node*left;
    struct node*right;
    char data;
}node;

void main()
{
    node*tree=NULL;
    char c[]={ 'A','B','C','D','E','F','\0',
               'G','\0','\0','\0','\0','\0','\0','\0','\0',
               '\0','\0','\0','\0','\0','\0','\0' };
    tree=insert(c,0);
    inorder(tree);
}
```

```
node* insert(char c[],int n)
{ node*tree=NULL;
if(c[n]!='\0')
{
    tree=(node*)malloc(sizeof(node));
    tree->left=insert(c,2*n+1);
    tree->data=c[n];
    tree->right=insert(c,2*n+2);
}
return tree;
}

//traverse the tree in inorder
void inorder(node*tree)
{
    if(tree!=NULL)
    {
        inorder(tree->left);
        printf("%c\t",tree->data);
        inorder(tree->right);
    }
}
```

Inorder traversal without recursion

- Using **Stack** is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack.
 - **Step 1:** Create an empty stack S. Initialize **current** node as root.
 - **Step 2:** Push the current node to S and set **current = current->left** until **current** is **NULL**
 - **Step 3:** If **current** is **NULL** and stack is not empty then
 - Pop the top item from stack.
 - Print the popped item, set **current = popped_item->right**
 - Go to step 2.
 - **Step 4:** If **current** is **NULL** and stack is empty then we are done.

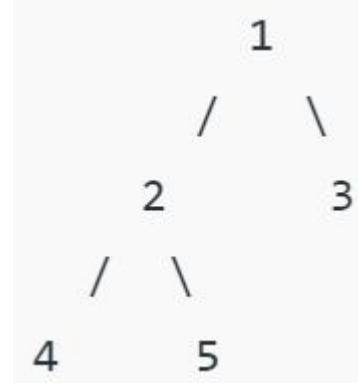
Inorder traversal without recursion

- Let's see the below tree for example of inorder traversal of a binary tree using stack:

Step 1: Creates an empty stack: **S = NULL**

Step 2: Sets current as address of root: **current -> 1**

Step 3: Pushes the current node and set **current = current->left** until **current** is **NULL**



current -> 1: push 1: Stack S -> 1

current -> 2: push 2: Stack S -> 2, 1

current -> 4: push 4: Stack S -> 4, 2, 1

current = NULL

Inorder traversal without recursion

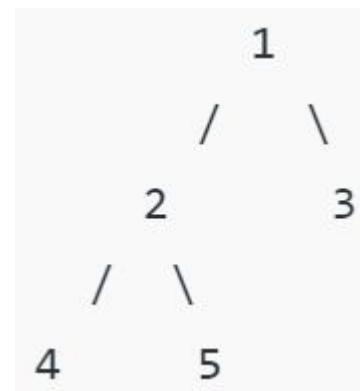
- Let's see the below tree for example of inorder traversal of a binary tree using stack:

Step 4: pops from S

a) **Pop 4:** Stack S → 2, 1

b) **print "4"**

c) **current = NULL /*right of 4 */** and go to step 3 Since current is NULL step 3 doesn't do anything.



Inorder traversal without recursion

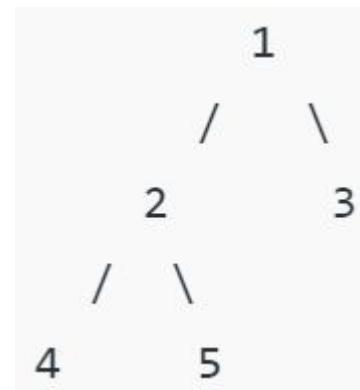
- Let's see the below tree for example of inorder traversal of a binary tree using stack:

Step 4: pops from S

- Pop 2: Stack S → 1
- print "2"
- current = 5 /*right of 2 */ and go to step 3

Step 3: Pushes the current node 5 and set current = current->left until current is NULL

Stack S -> 5, 1 and current = NULL



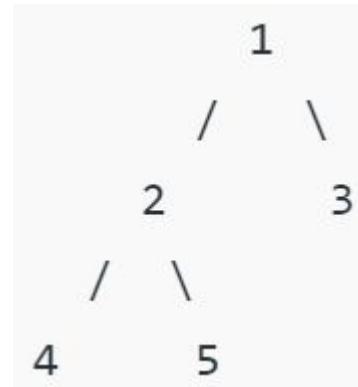
Inorder traversal without recursion

Step 4: pops from S

- a) **Pop 5:** Stack S → 1
- b) **print "5"**
- c) **current = NULL /*right of 5 */** and go to step 3

Step 4: pops from S

- a) **Pop 1:** Stack S → NULL
- b) **print "1"**
- c) **current = 3 /*right of 1 */** and go to step 3



Inorder traversal without recursion

Step 3: Pushes the **current** node 3 and set **current = current->left** until **current** is **NULL**

Stack S -> 3 and **current = NULL**

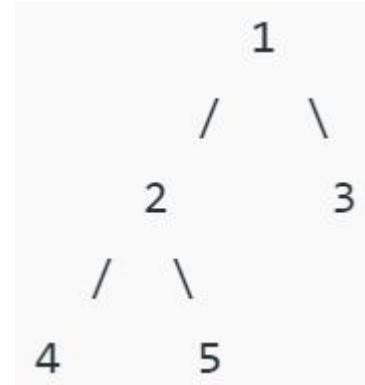
Step 4: pops from S

a) Pop 3: Stack S → NULL

b) print "3"

c) **current = NULL /*right of 3 */** and go to step 3

Step 5: Now, **Current** is **NULL** and **Stack** is **empty**, so **traversal is done**



Inorder traversal using stack

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree tNode has data,
pointer to left child
and a pointer to right child */
struct tNode
{
int data;
struct tNode* left;
struct tNode* right;
};

/* Structure of a stack node.
Linked List implementation is used
for stack. A stack node contains a
pointer to tree node and a pointer to
next stack node */
struct sNode
{
struct tNode *t;
struct sNode *next;
};
```

```
/* Stack related functions */
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);

struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
        malloc(sizeof(struct tNode));
    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;
    return(tNode);
}

int main()
{
    struct tNode *root = newtNode(1);
    root->left = newtNode(2);
    root->right = newtNode(3);
    root->left->left = newtNode(4);
    root->left->right = newtNode(5);

    inOrder(root);

    getchar();
    return 0;
}
```

```
void inOrder(struct tNode *root)
{
    struct tNode *current = root;
    struct sNode *s = NULL; /* Initialize stack s */
    bool done = 0;

    while (!done)
    {
        if(current != NULL)
        {
            push(&s, current);
            current = current->left;
        }
        else
        {
            if (!isEmpty(s))
            {
                current = pop(&s);
                printf("%d ", current->data);
                current = current->right;
            }
            else
                done = 1;
        }
    } /* end of while */
}
```

```
void push(struct sNode** top_ref, struct tNode *t)
{
    struct sNode* new_tNode =
        (struct sNode*) malloc(sizeof(struct sNode));
    if(new_tNode == NULL)
    {
        printf("Stack Overflow \n");
        getchar();
        exit(0);
    }
    new_tNode->t = t;
    new_tNode->next = (*top_ref);
    (*top_ref) = new_tNode; /* move the head to point to the new tNode */
}

bool isEmpty(struct sNode *top)
{
    return (top == NULL)? 1 : 0;
}
```

```
void inOrder(struct tNode *root)
{
    struct tNode *current = root;
    struct sNode *s = NULL; /* Initialize stack s */
    bool done = 0;

    while (!done)
    {
        if(current != NULL)
        {
            push(&s, current);
            current = current->left;
        }
        else
        {
            if (!isEmpty(s))
            {
                current = pop(&s);
                printf("%d ", current->data);
                current = current->right;
            }
            else
                done = 1;
        }
    } /* end of while */
}
```

```
struct tNode *pop(struct sNode** top_ref)
{
    struct tNode *res;
    struct sNode *top;

    /*If sNode is empty then error */
    if(isEmpty(*top_ref))
    {
        printf("Stack Underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->t;
        *top_ref = top->next;
        free(top);
        return res;
    }
}
```

Inorder traversal using Stack: Exercise

Step 1: Create an empty stack S. Initialize **current** node as root.

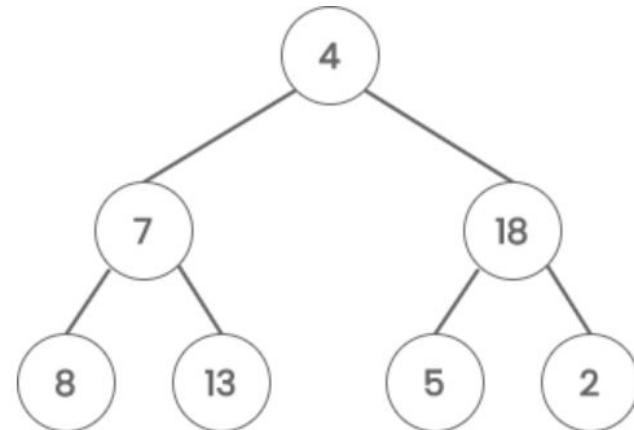
Step 2: Push the current node to S and set

current = current->left until **current** is **NULL**

Step 3: If **current** is **NULL** and stack is not empty then

- Pop the top item from stack.
- Print the popped item, set **current = popped_item->right**
- Go to step 2.

Step 4: If **current** is **NULL** and stack is empty then we are done.

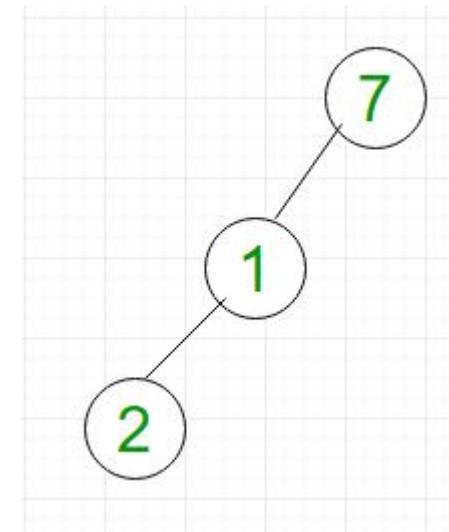


Binary Tree: Time Complexity

- Binary Tree supports various operations such as
 - Insertion ,
 - Deletion ,
 - Traversals ,
 - Searching.

Binary Tree: Time Complexity

- In a binary tree, a node can have maximum two children. Consider the left skewed binary tree shown in Figure:
- **Searching:**
 - For searching element 2, we have to traverse all elements (assuming we do breadth first traversal).
 - Therefore, searching in binary tree has worst case complexity of $O(n)$.



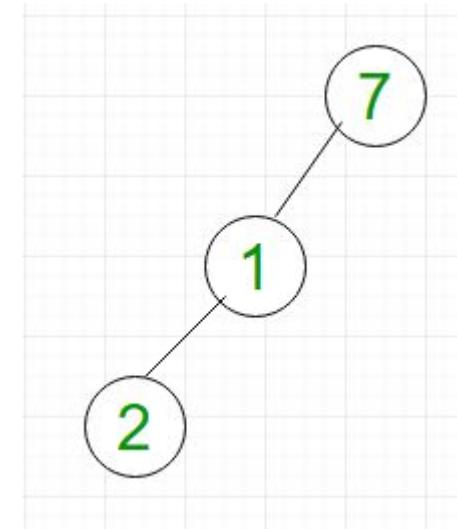
Binary Tree: Time Complexity

- **Insertion:**

- For inserting element as left child of 2, we have to traverse all elements.
- Therefore, insertion in binary tree has worst case complexity of $O(n)$.

- **Deletion:**

- For deletion of element 2, we have to traverse all elements to find 2 (assuming we do breadth first traversal).
- Therefore, deletion in binary tree has worst case complexity of $O(n)$.



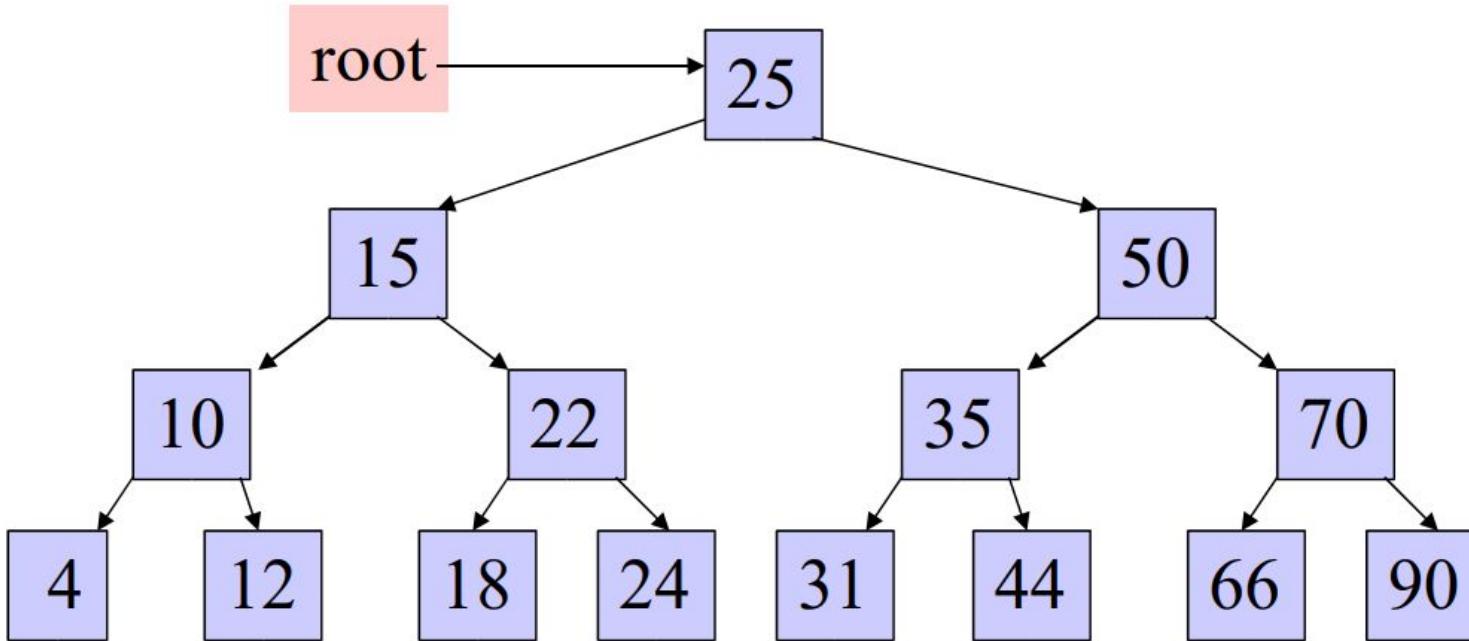
Binary Search Tree (BST)

- A **binary search tree** is a binary tree that is either empty or in which the data entry of every node has a key and satisfies the conditions:
 - The key of the left child of a node (if it exists) is less than the key of its parent node.
 - The key of the right child of a node (if it exists) is greater than the key of its parent node.
 - The left and right subtrees of the root are again binary search trees.

Note: No two entries in a binary search tree may have equal keys.

Binary Search Tree (BST)

- A **binary search tree** example:



Binary Search Tree (BST)

- A **binary search tree** advantages:
 - Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
 - As compared to array and linked lists, insertion and deletion operations are faster in BST.
- The BST arranges its node in a sorted manner.
- As the name suggest, the most important application of BST is in searching.
- The average running time of searching an element in a BST is $O(\log n)$, which is better than other data structures like array and linked list.

Binary Search Tree (BST): creation

- Binary search Tree: **Operations**
 - **Insert**
 - **Search/Traverse**
 - **Delete**

Binary Search Tree (BST): Insertion

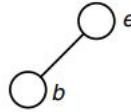
- The insert operation involves adding an element into the binary tree. The location of the new element is determined in such a manner that insertion does not disturb the sort order of the tree.
- If a Record with a key matching that of new data already belongs to the Search tree a code of duplicate error is returned. Otherwise, the Record new data is inserted into the tree in such a way that the properties of a binary search tree are preserved, and a code of success is returned.
- Let's start creating a BST by inserting the nodes in the following order:

e, b, d, f, a, g, c

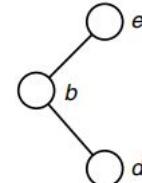
Binary Search Tree (BST): Insertion



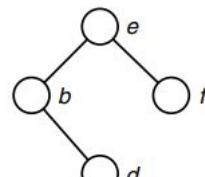
(a) Insert *e*



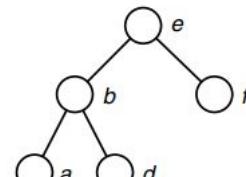
(b) Insert *b*



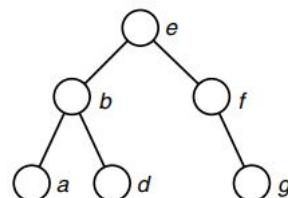
(c) Insert *d*



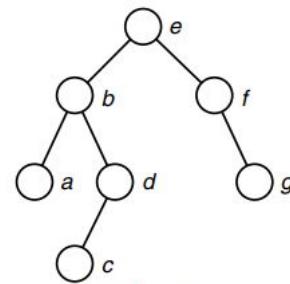
(d) Insert *f*



(e) Insert *a*



(f) Insert *g*



(g) Insert *c*

Binary Search Tree (BST): Insertion implementation

```
node *insert(node *r, int n)
{
    if (r==NULL)
    {
        r=(node*) malloc (sizeof(node));
        r->LEFT = r->RIGHT = NULL;
        r->INFO = n;
    }
    else if(n<r->INFO)
        r->LEFT = insert(r->LEFT, n);
    else if(n>r->INFO)
        r->RIGHT = insert(r->RIGHT, n);
    else if(n==r->INFO)
        printf("\nInsert Operation failed: Duplicate Entry!!");
    return(r);
}
```

A series of recursive function calls are required to identify the precise location where the new node will be inserted.

Binary Search Tree (BST): Insertion exercise

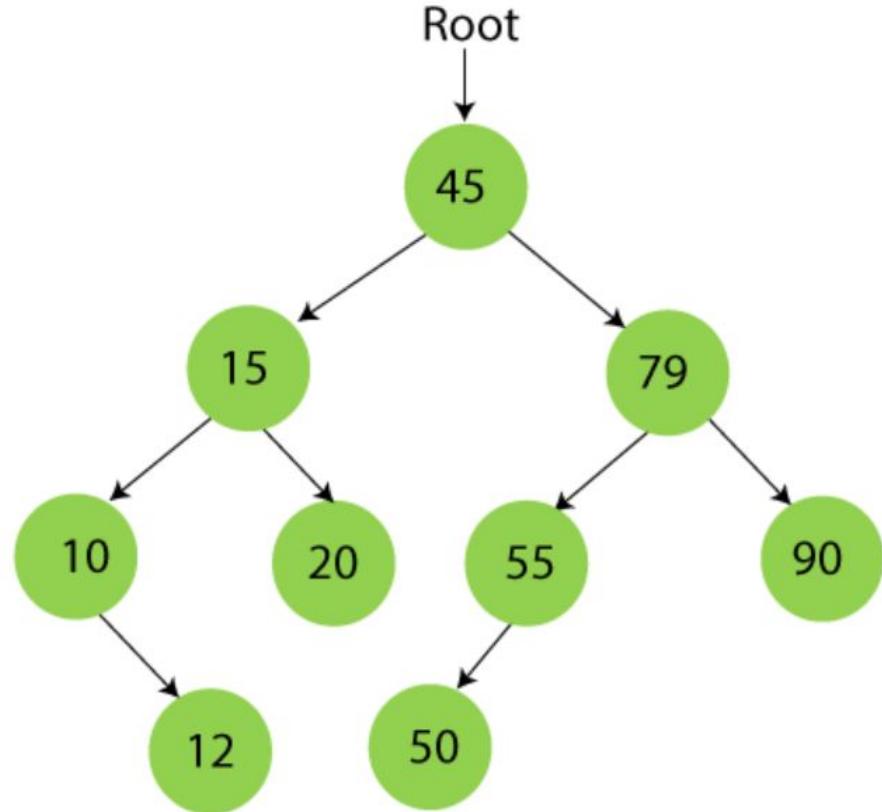
- Let's start creating a BST by inserting the nodes in the following order:

45, 15, 79, 90, 10, 55, 12, 20, 50

Binary Search Tree (BST): Insertion exercise

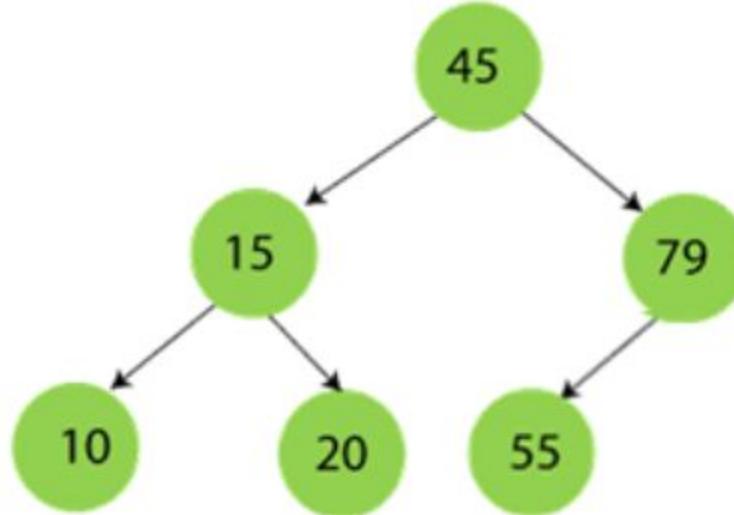
- Let's start creating a BST by inserting the nodes in the following order:

45, 15, 79, 90, 10, 55, 12, 20, 50



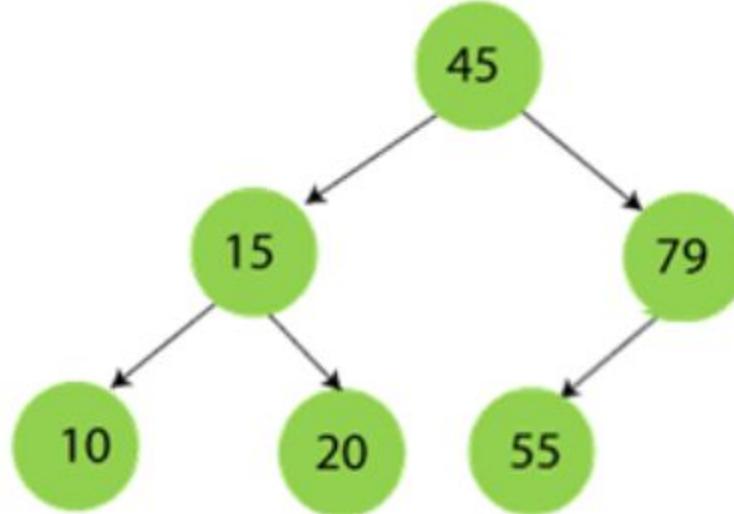
Binary Search Tree (BST): Insertion exercise

- Consider the tree below and insert item 65 into tree.
 - Points to be remembered: A new key in BST is always inserted at the leaf.



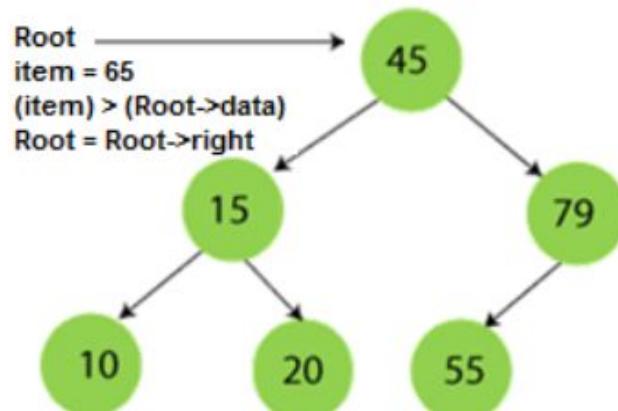
Binary Search Tree (BST): Insertion exercise

- Consider the tree below and insert item 65 into tree.
 - Points to be remembered: A new key in BST is always inserted at the leaf.

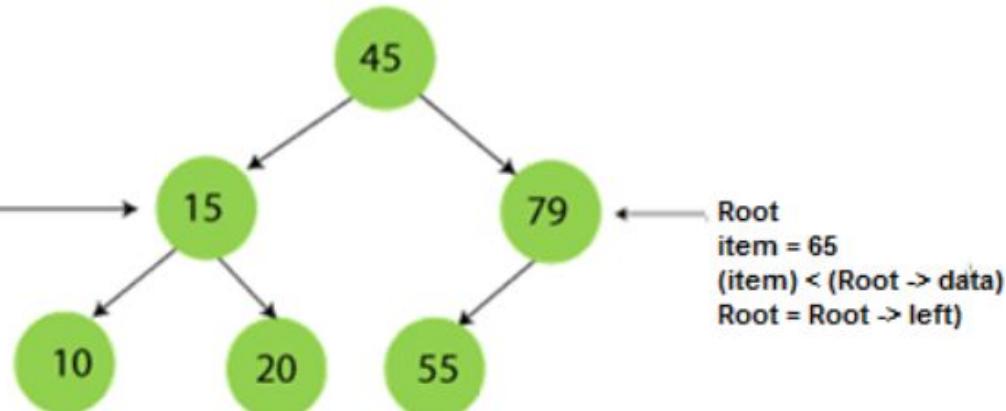


Binary Search Tree (BST): Insertion exercise

- Consider the tree below and insert item 65 into tree.
 - Points to be remembered: A new key in BST is always inserted at the leaf.



Insert node 65
Step1

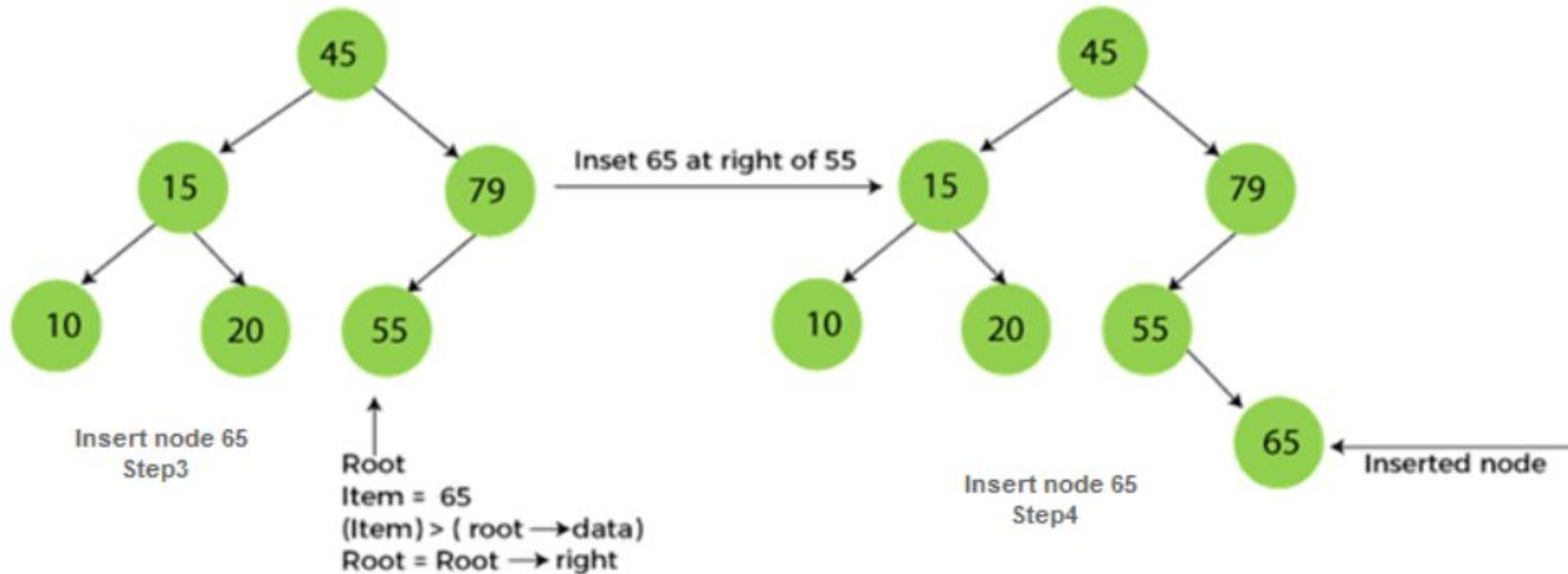


Insert node 65
Step2

Root
item = 65
(item) < (Root->data)
Root = Root -> left

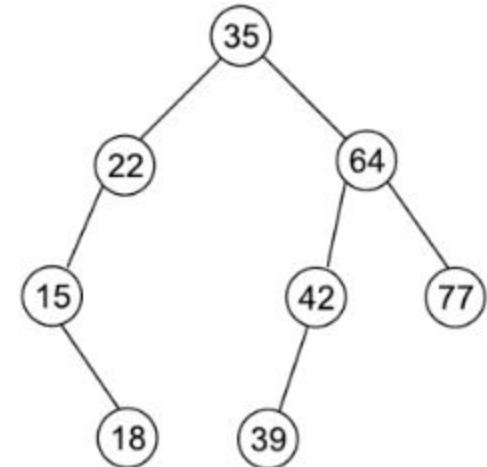
Binary Search Tree (BST): Insertion exercise

- Consider the tree below and insert item 65 into tree.
 - Points to be remembered: A new key in BST is always inserted at the leaf.



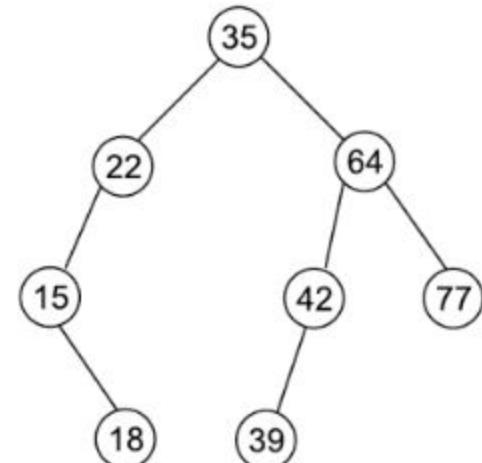
Binary Search Tree (BST): Searching

- The search operation involves traversing the various nodes of a binary tree to search the desired elements.
- The sorted nature of the tree greatly benefits the search operation as with each iteration, the nodes to be searched gets reduce.
- For example, if a value to be search is less than the root value then remainder of the search operation will only be performed in the left subtree while the right subtree will be completely ignored.



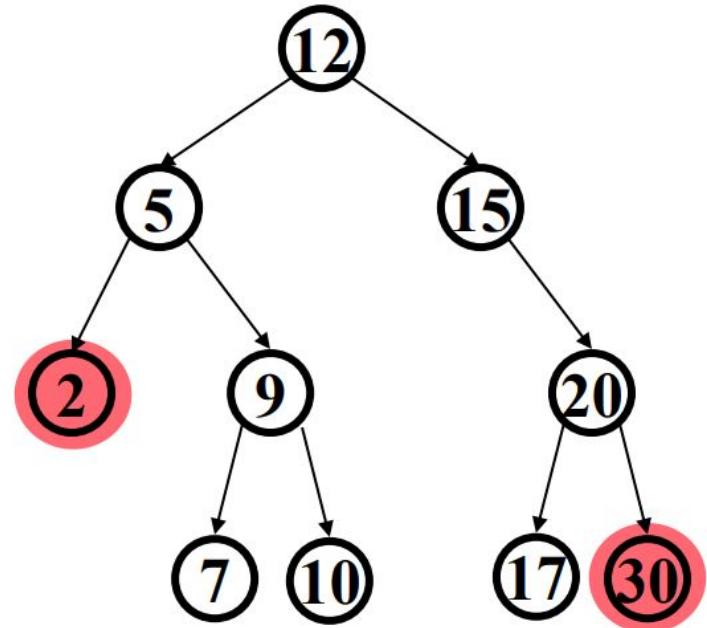
Binary Search Tree (BST): Searching implementation

```
void search(node *r, int n)
{
if(r==NULL)
{
printf("\n%d not present in the tree!!",n);
return;
}
else if(n==r->INFO)
printf("\nElement %d is present in the tree!!",n);
else if(n<r->INFO)
search(r->LEFT,n);
else
search(r->RIGHT,n);
}
```



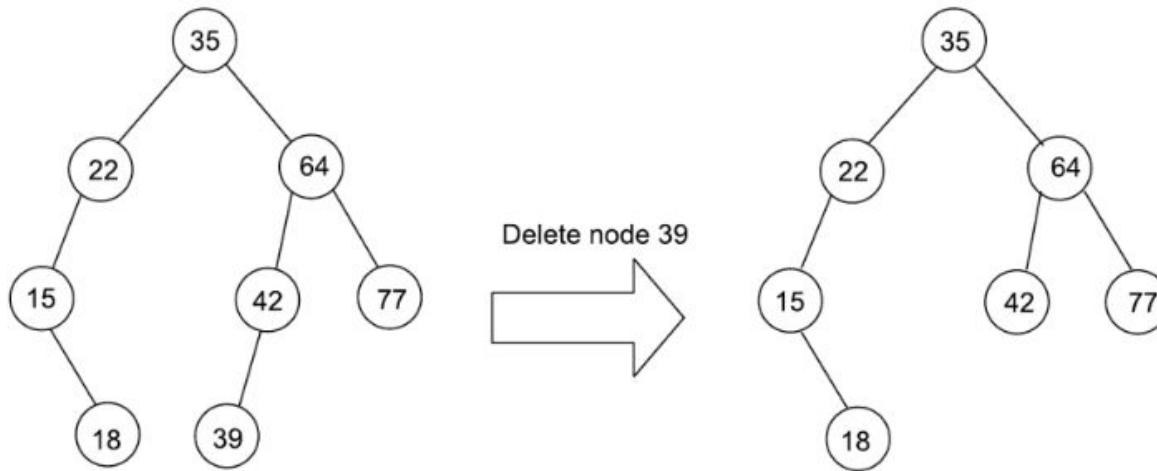
Binary Search Tree (BST): Searching

- Other BST searching operations are:
 - Finding **minimum** node value in BST
 - Left-Most node
 - Finding **maximum** node value in BST
 - Right-Most node



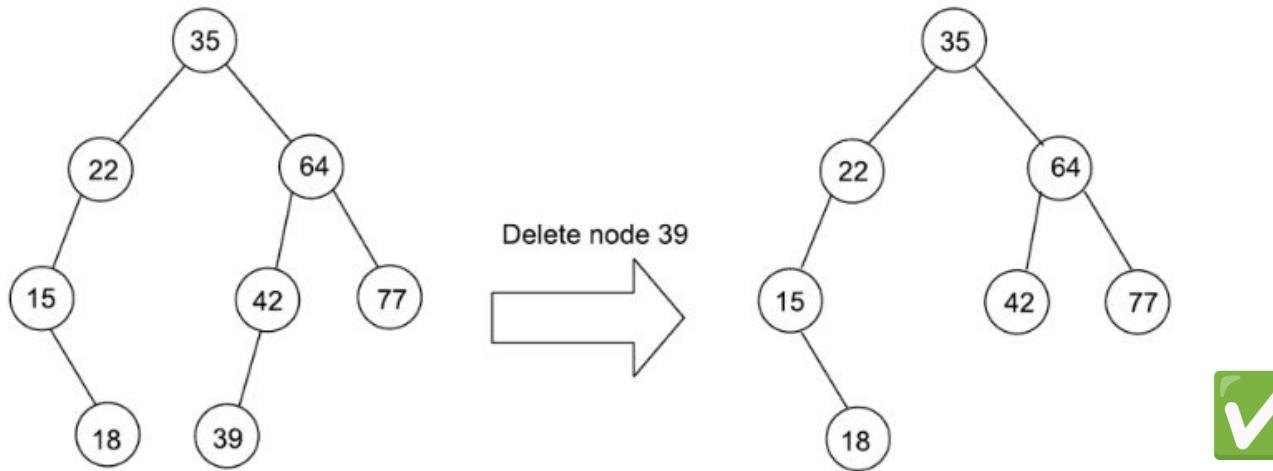
Binary Search Tree (BST): Deletion

- The delete operation involves removing an element from the BST.
- It is important to ensure that after the element is removed from the tree, the other elements are shuffled in such a manner that sort order of the tree is regained.



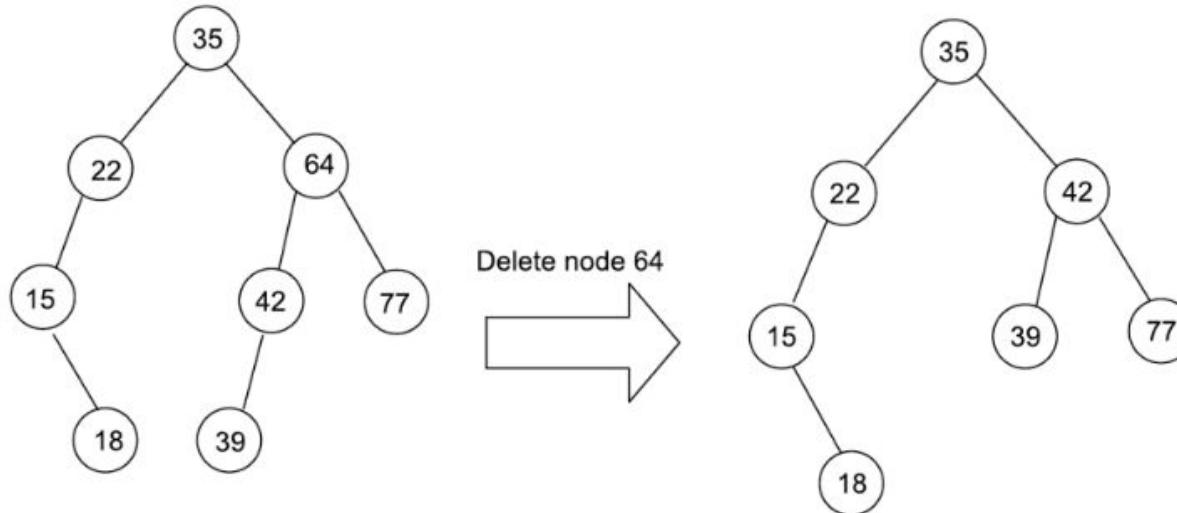
Binary Search Tree (BST): Deletion

- The delete operation involves removing an element from the BST.
- It is important to ensure that after the element is removed from the tree, the other elements are shuffled in such a manner that sort order of the tree is regained.



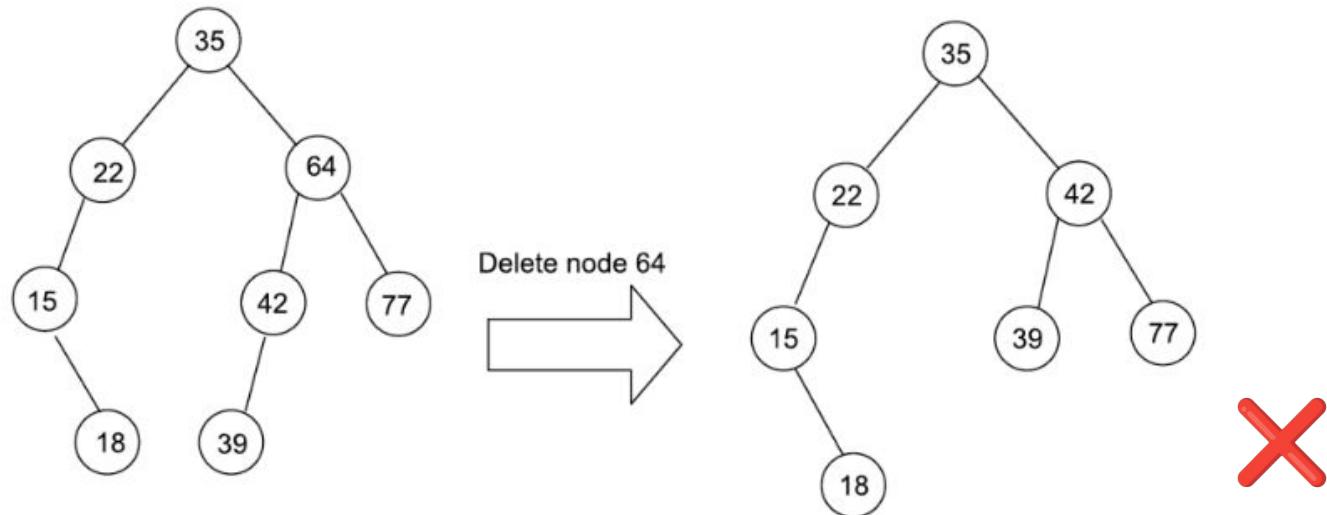
Binary Search Tree (BST): Deletion

- The delete operation involves removing an element from the BST.
- It is important to ensure that after the element is removed from the tree, the other elements are shuffled in such a manner that sort order of the tree is regained.



Binary Search Tree (BST): Deletion

- The delete operation involves removing an element from the BST.
- It is important to ensure that after the element is removed from the tree, the other elements are shuffled in such a manner that sort order of the tree is regained.

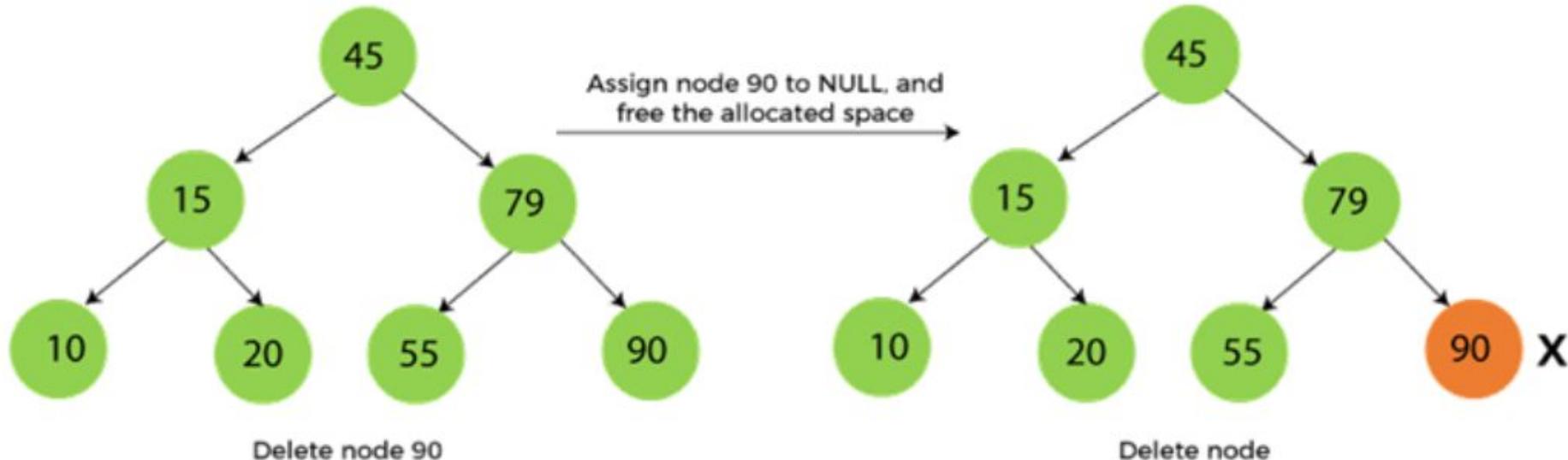


Binary Search Tree (BST): Deletion

- To delete a node from BST, there are three possible situations occur -
 - **Case 1:** The node to be deleted is the leaf node, or,
 - **Case II:** The node to be deleted has only one child, and,
 - **Case III:** The node to be deleted has two children

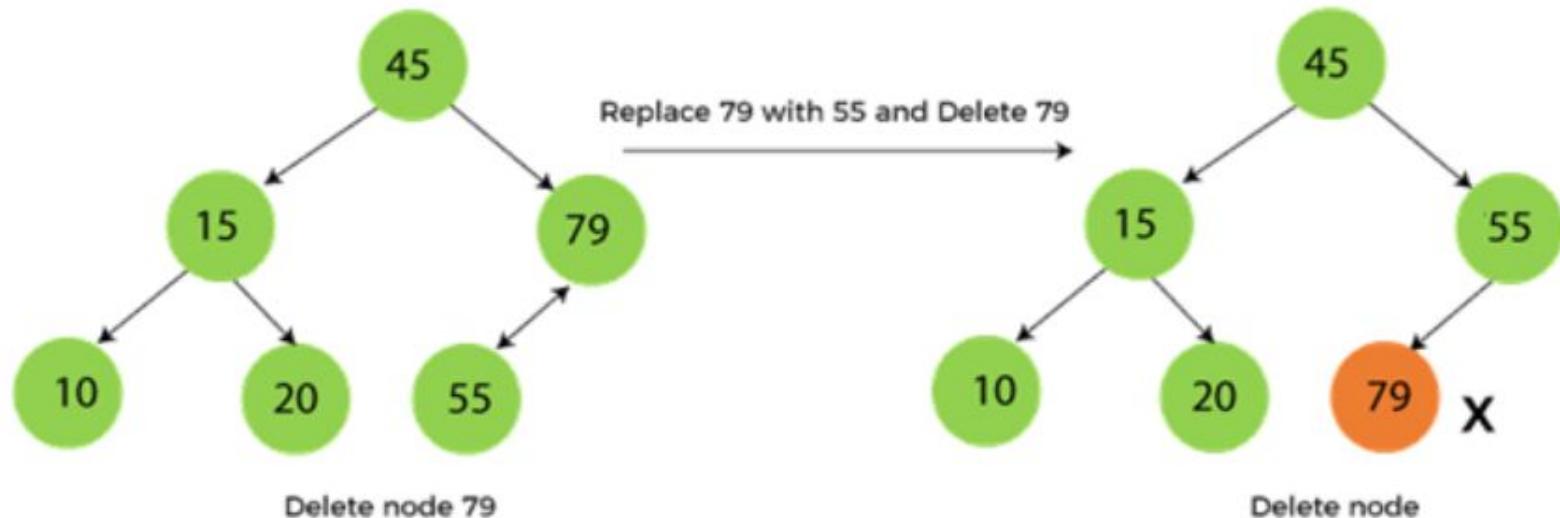
Binary Search Tree (BST): Deletion

- **Case I:** When the node to be deleted is the leaf node:
 - It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.



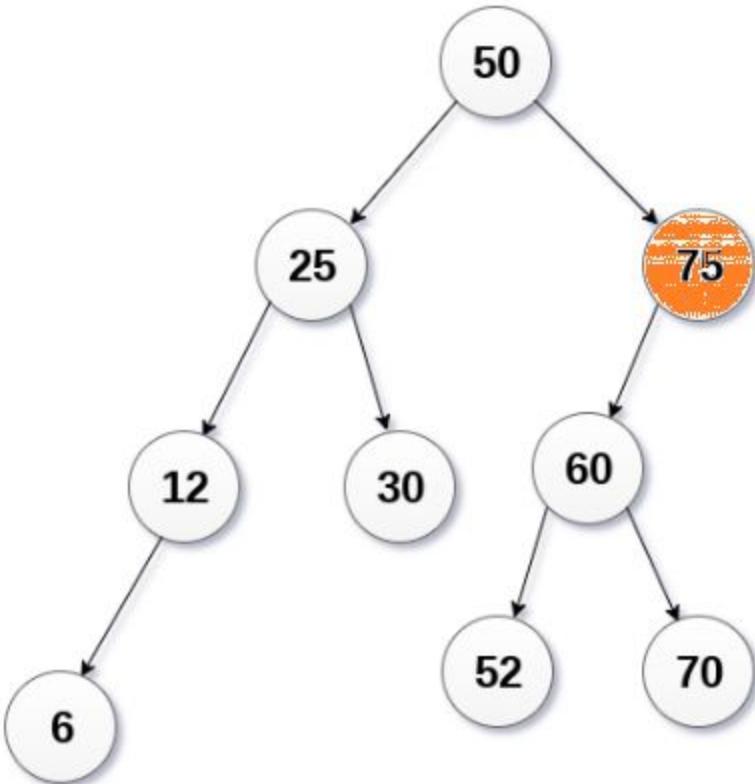
Binary Search Tree (BST): Deletion

- **Case II:** When the node to be deleted has only one child:
 - In this case, we have to swap the target node with its child, and then delete the child node.



Binary Search Tree (BST): Deletion

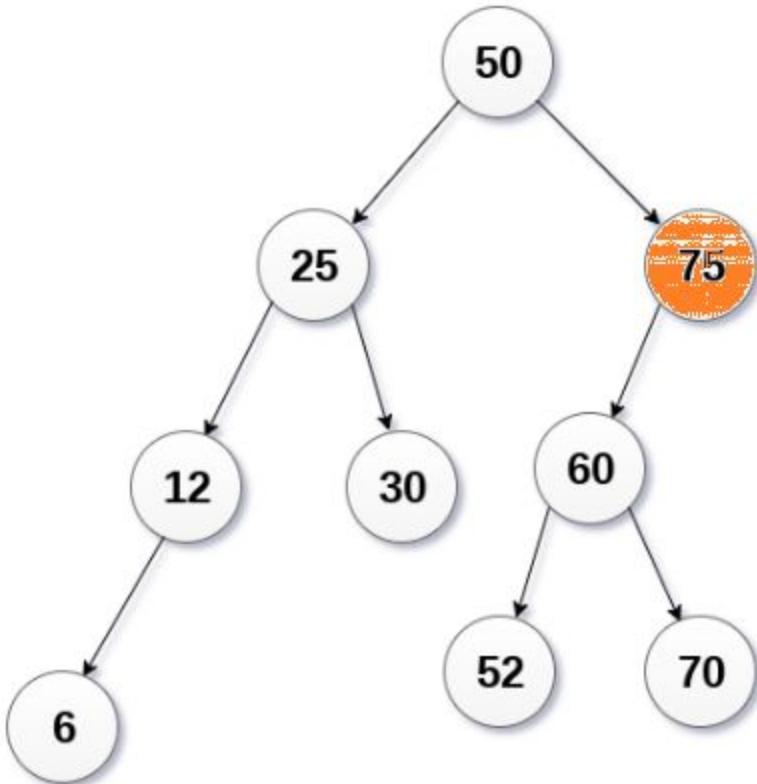
- **Case II:** When the node to be deleted has only one child:
 - Delete 75 from the tree



Binary Search Tree (BST): Deletion

- **Case II:** When the node to be deleted has only one child:
 - Delete 75 from the tree

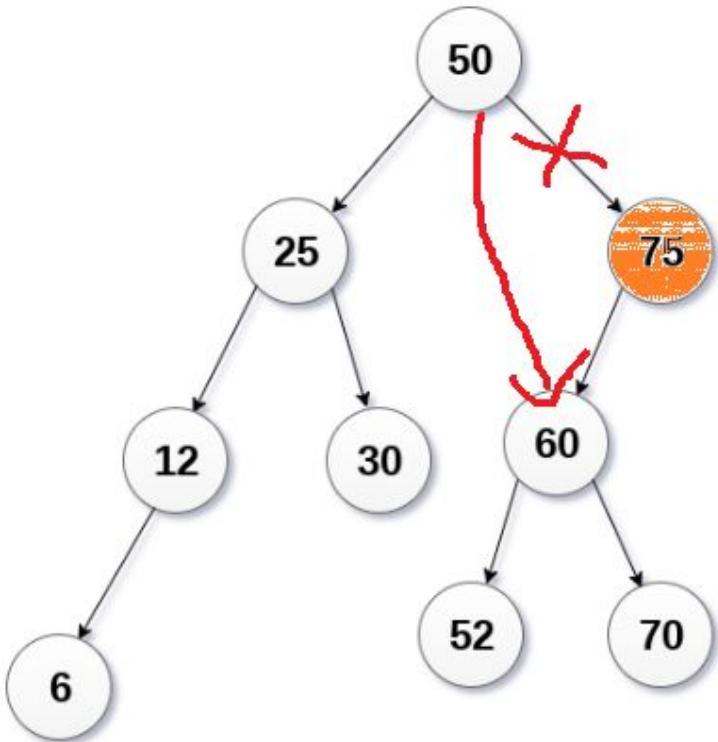
In this case, we have to link target node's parent to target node's only child .



Binary Search Tree (BST): Deletion

- **Case II:** When the node to be deleted has only one child:
 - Delete 75 from the tree

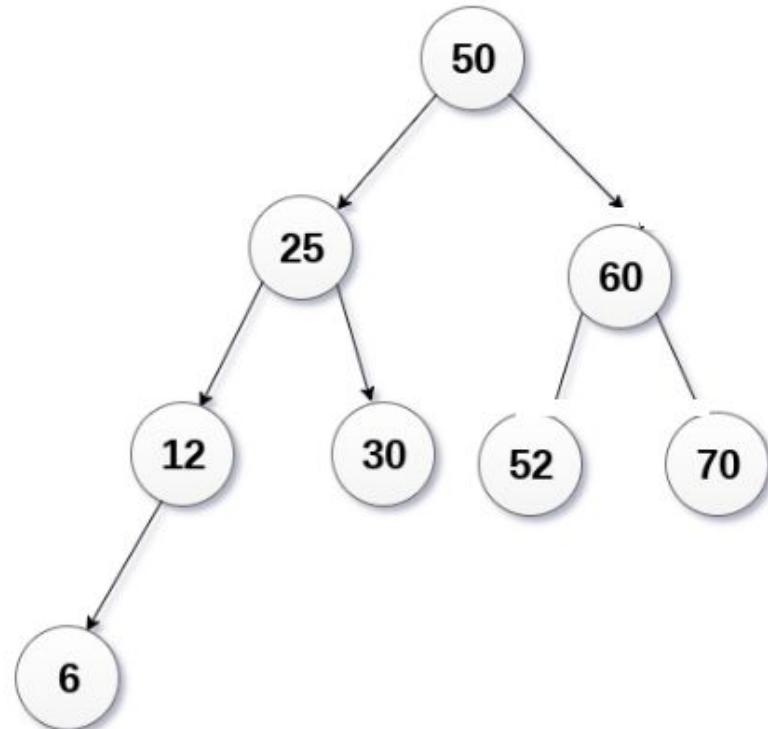
In this case, we have to link target node's parent to target node's only child .



Binary Search Tree (BST): Deletion

- **Case II:** When the node to be deleted has only one child:
 - Delete 75 from the tree

In this case, we have to link target node's parent to target node's only child .

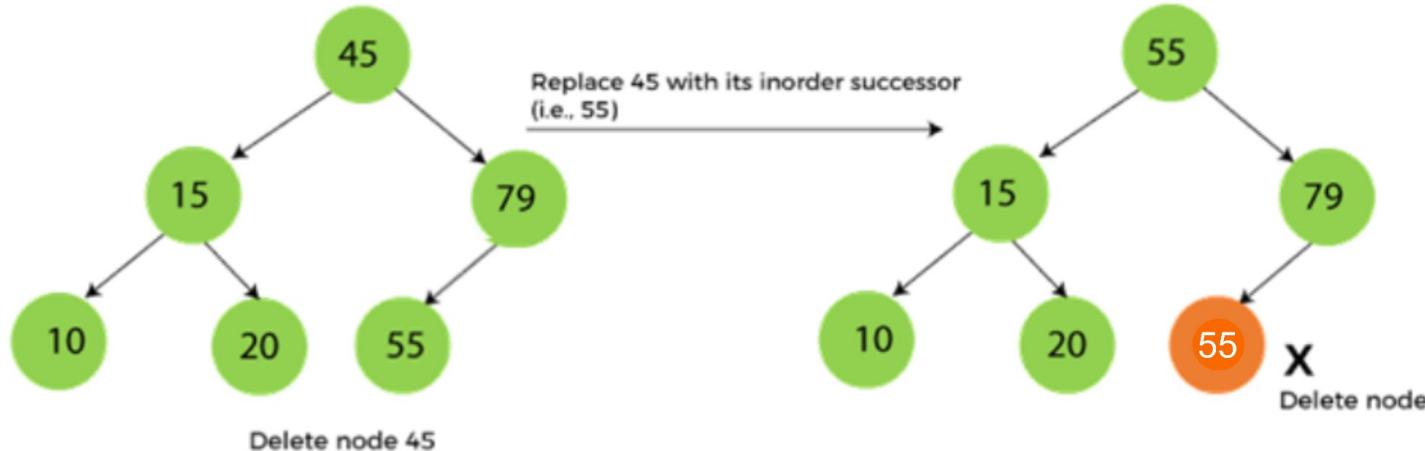


Binary Search Tree (BST): Deletion

- **Case III:** When the node to be deleted has only two children:
 - This case of deleting a node in BST is a bit complex among other two cases.
 - In such a case, the steps to be followed are listed as follows -
 - First, find the inorder successor or inorder predecessor of the node to be deleted.
 - After that, replace that node with the inorder successor or inorder predecessor node.
 - Delete the inorder successor or inorder predecessor as per Case I or Case II.

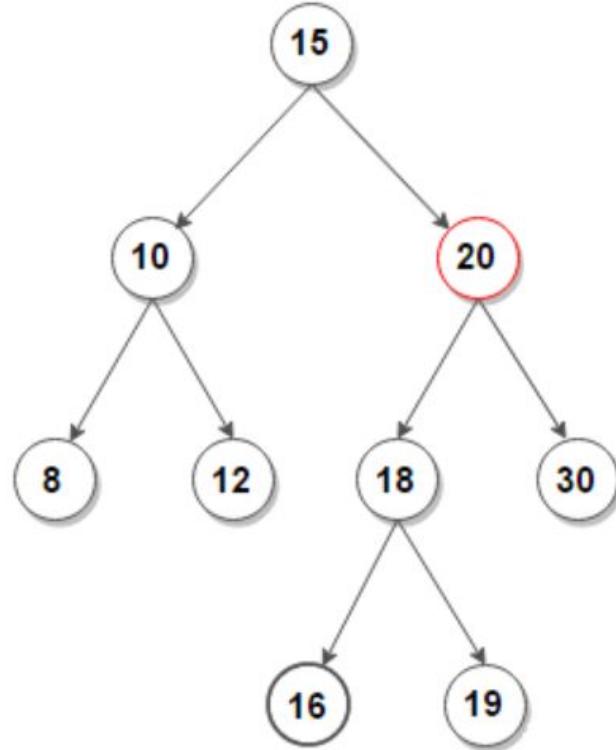
Binary Search Tree (BST): Deletion

- When the node to be deleted has only two children:
 - Suppose we have to delete 45 in the following tree. We find the inorder successor of the node 45 which is 55 and replace 45 with that. Then delete 45.



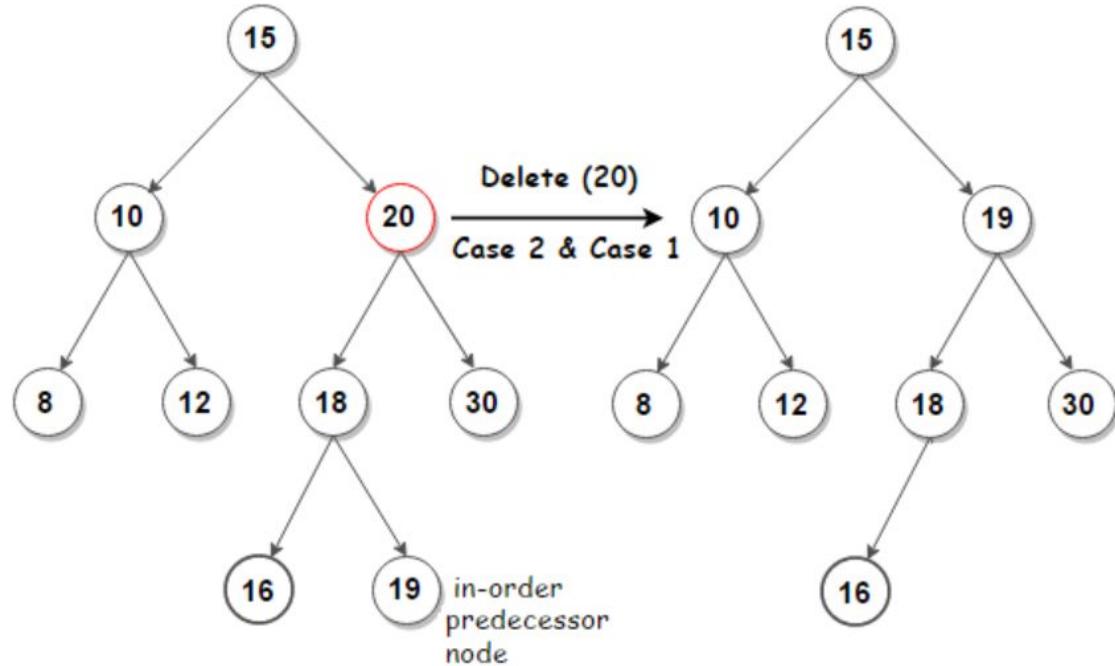
Binary Search Tree (BST): Deletion

- Delete node 20:
 - Using its successor
 - Using its predecessor



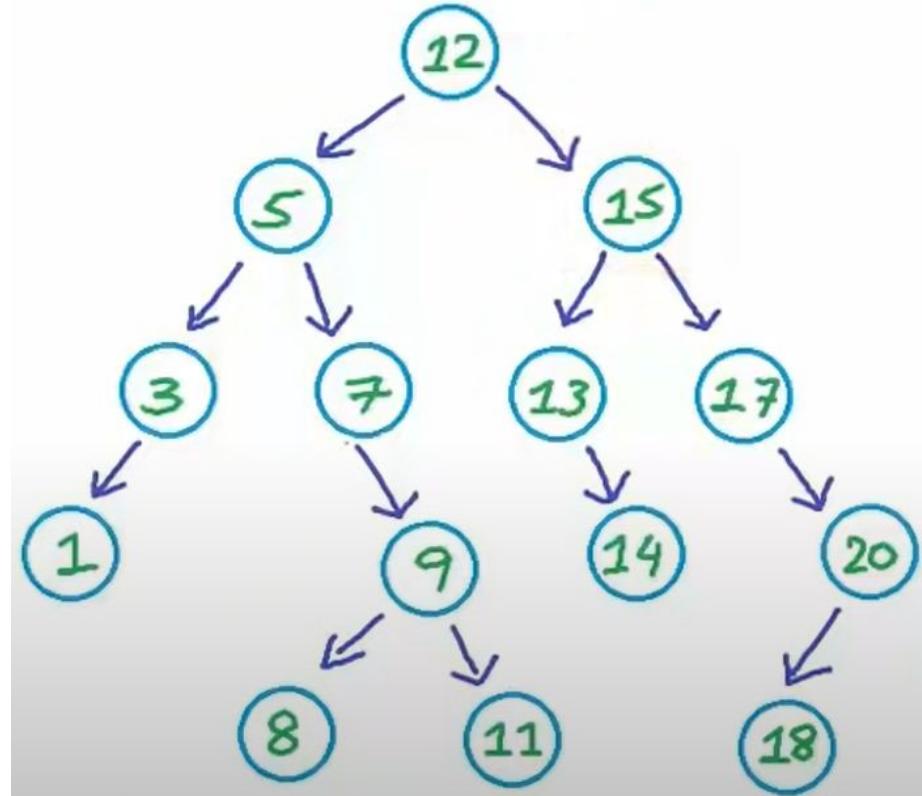
Binary Search Tree (BST): Deletion

- Delete node 20:
 - Using its successor
 - Using its predecessor



Binary Search Tree (BST): Deletion

- Delete node 15:
 - Using its successor
 - Using its predecessor



Binary Search Tree (BST): implementation

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int key;
    struct node *left, *right;
};

// Create a node
struct node *newNode(int item) {
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

Binary Search Tree (BST): implementation

```
// Driver code
int main() {
    struct node *root = NULL;
    root = insert(root, 8);
    root = insert(root, 3);
    root = insert(root, 1);
    root = insert(root, 6);
    root = insert(root, 7);
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 4);

    printf("Inorder traversal: ");
    inorder(root);

    printf("\nAfter deleting 10\n");
    root = deleteNode(root, 10);
    printf("Inorder traversal: ");
    inorder(root);
}

// Insert a node
struct node *insert(struct node *node, int key) {
    // Return a new node if the tree is empty
    if (node == NULL) return newNode(key);

    // Traverse to the right place and insert the node
    else if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    return node;
}
```

Binary Search Tree (BST): implementation

```
// Inorder Traversal
void inorder(struct node *root) { // Find the inorder successor
    if (root != NULL) {
        // Traverse left
        inorder(root->left);

        // Traverse root
        printf("%d -> ", root->key);

        // Traverse right
        inorder(root->right);
    }
}

// Find the leftmost leaf
struct node *minValueNode(struct node *node) {
    struct node *current = node;

    while (current && current->left != NULL)
        current = current->left;

    return current;
}
```

Binary Search Tree (BST): implementation

```
// Deleting a node
struct node *deleteNode(struct node *root, int key) {
    // Return if the tree is empty
    if (root == NULL) return root;

    // Find the node to be deleted
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    else {
        // If the node is with only one child or no child
        if (root->left == NULL) {
            struct node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
    }

    // If the node has two children
    struct node *temp = minValueNode(root->right);

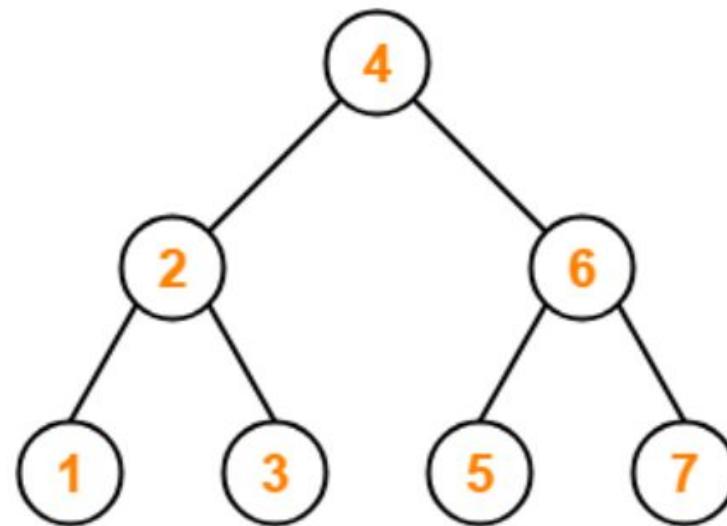
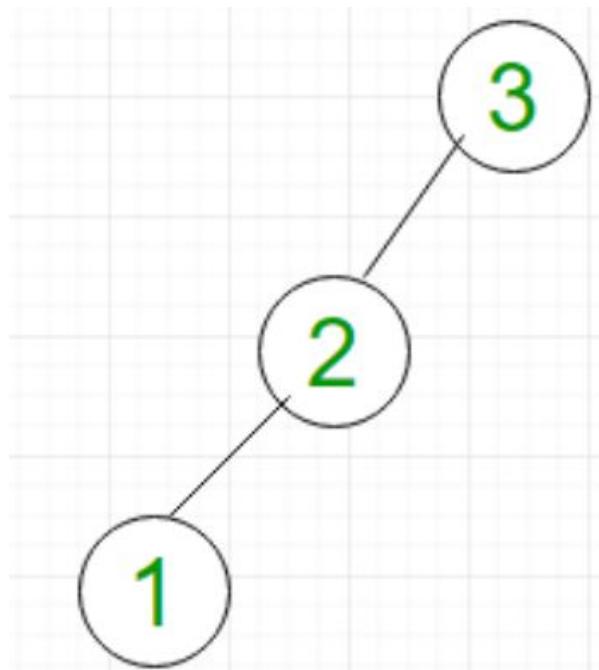
    // Place the inorder successor in
    // position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
}

return root;
}
```

BST complexity

- Worst case scenario: $O(n)$
- Best case scenario: $O(\log n)$



Heap tree

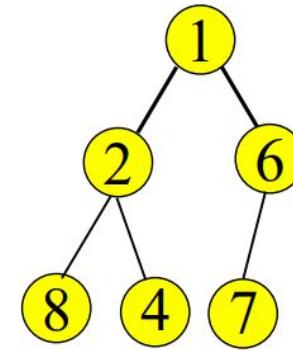
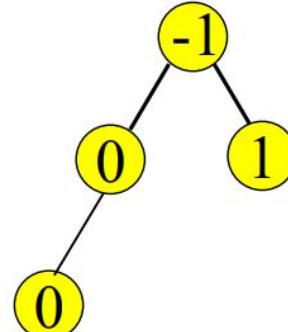
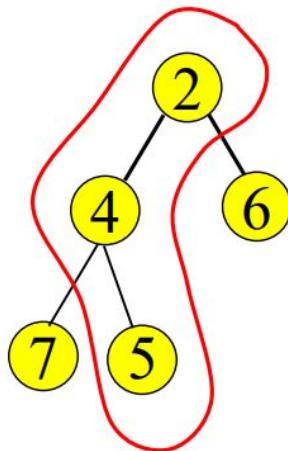
- Heap is a data structure that is used to implement the priority queue.
- Applications:
 - Find the smallest (or highest priority) item quickly –
 - Operating system needs to schedule jobs according to priority instead of FIFO
 - Find student with highest grade, employee with highest salary etc.
- In some operations, the heap are more efficient than BST.
- So, now the question is “What is Binary Heap?”

Defining Binary Heap

- A binary heap is a binary tree (NOT a BST) that has two property:
 - **Ordering property:**
 - every node is less than or equal to its children
 - or, every node is greater than or equal to its children
 - **Structural property:**
 - the tree is completely filled except possibly the bottom level, which is filled from left to right
- The root node is always the smallest node
 - or the largest, depending on the heap order

Binary Heap: Ordering property

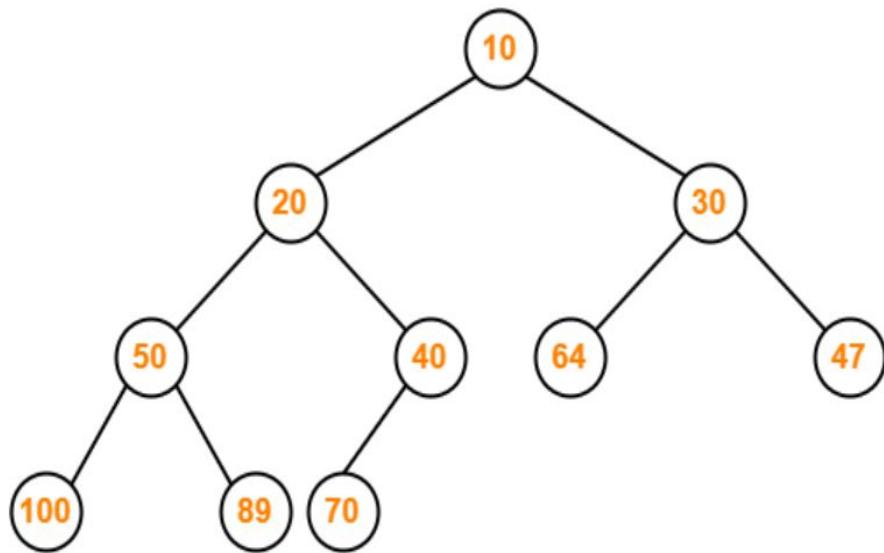
- A heap provides limited ordering information
- Each **path** is sorted, but the subtrees are not sorted relative to each other
 - A binary heap is NOT a binary search tree
- This gives rise to two types of heaps- **min heap** and **max heap**.



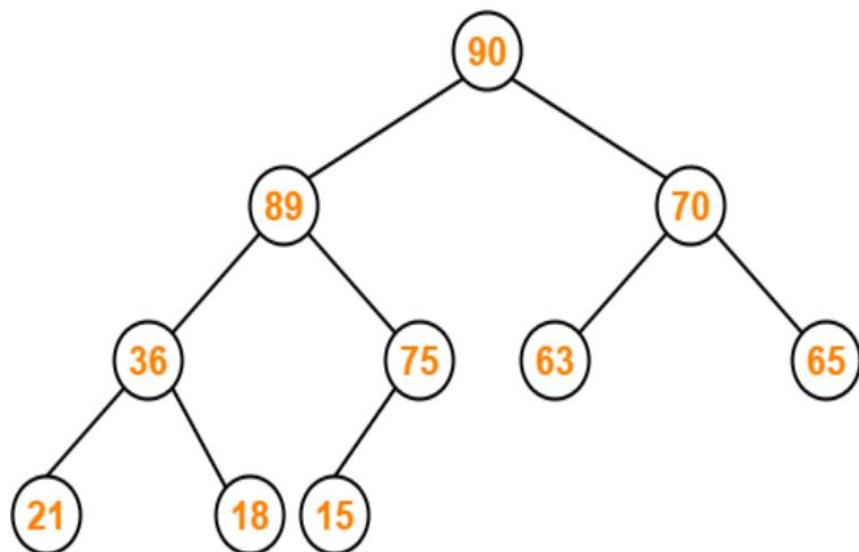
These are all valid binary heaps (minimum)

Binary Heap: Ordering property

- Min heap and Max heap.



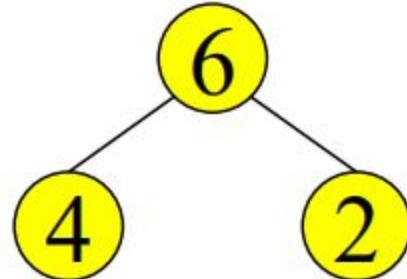
Min Heap



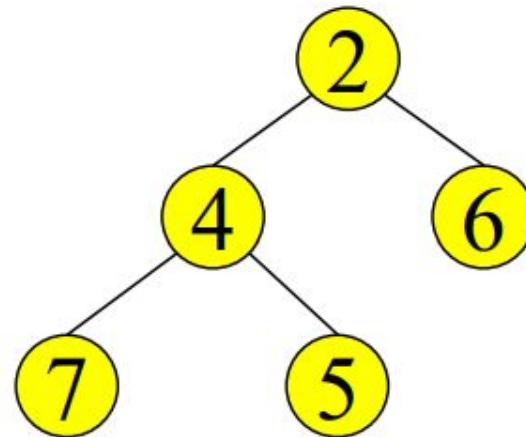
Max Heap

Binary Heap: structuring property

- A binary heap is a complete tree – except possibly the last level where the nodes are strictly filled from left to right.



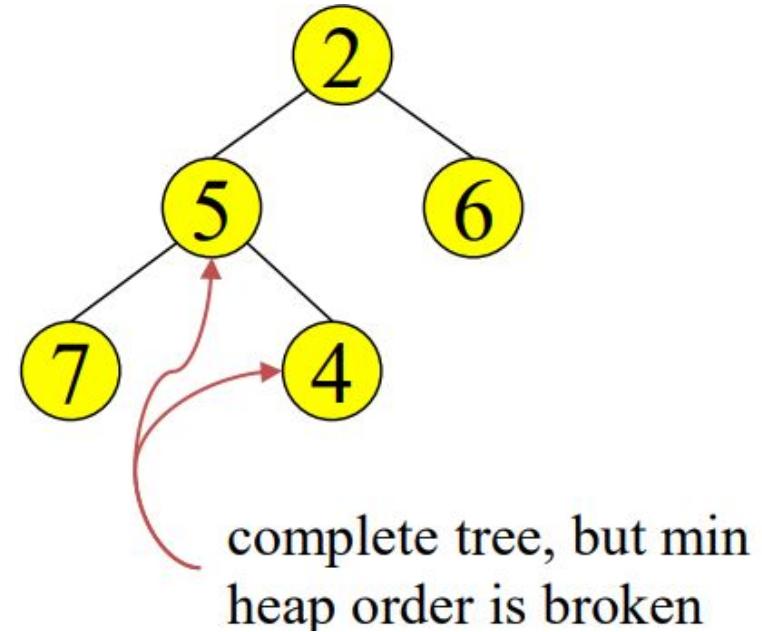
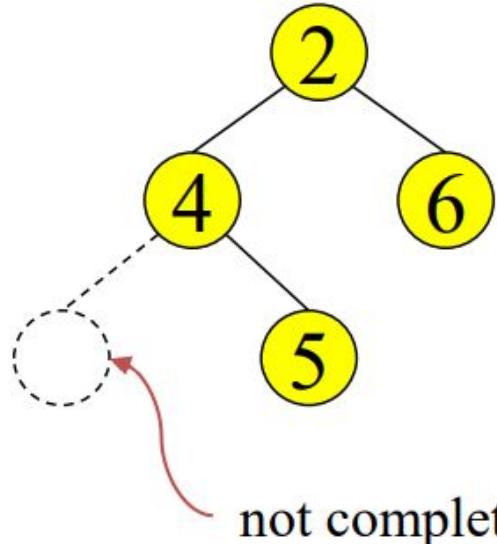
complete tree,
heap order is "max"



complete tree,
heap order is "min"

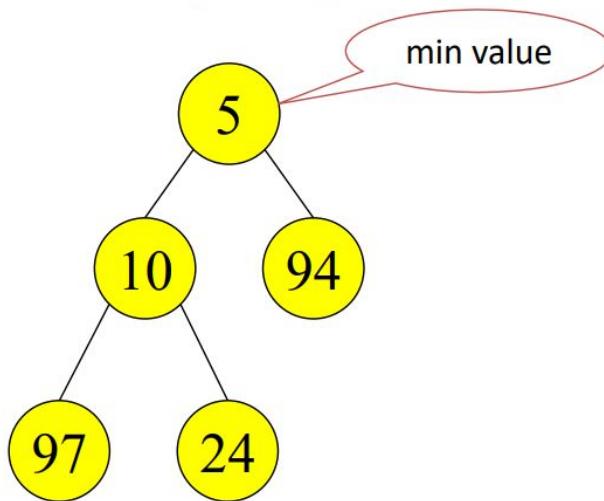
Binary Heap: structuring property

- A binary heap is a complete tree – except possibly the last level where the nodes are strictly filled from left to right.

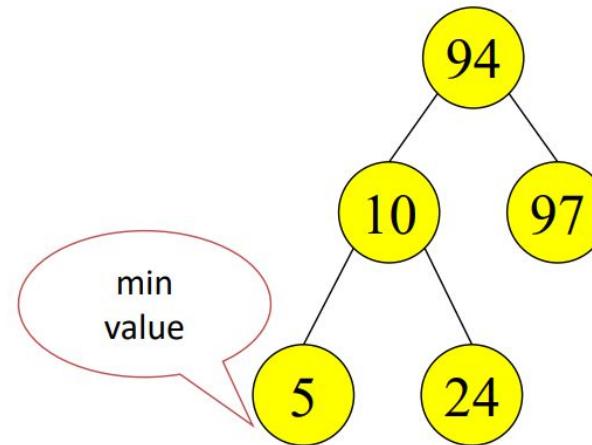


Binary Heap Vs Binary Search Tree

Binary Heap



Binary Search Tree

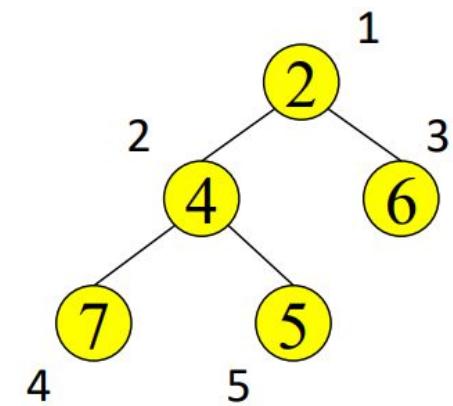
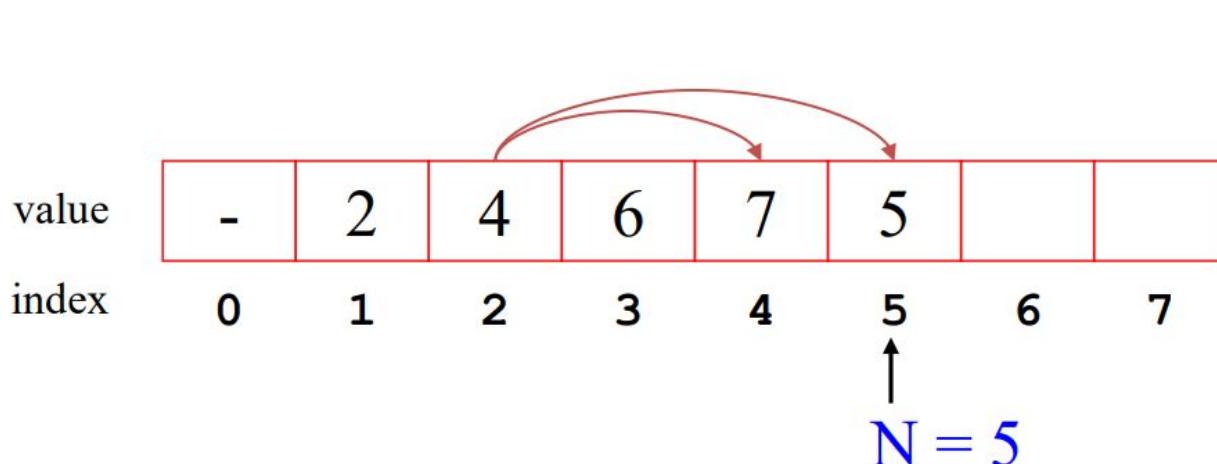


Parent is less than both
left and right children

Parent is greater than left
child, less than right child

Binary heap representation using Array

- Root node = $A[1]$
- Children of $A[i] = A[2i], A[2i + 1]$
- Parent of $A[j] = A[j/2]$
- Keep track of current size N (number of nodes)



Binary heap representation using Array

- Level order traversal technique may be used to achieve the array representation of a heap tree.
- Array representation of a heap never contains any empty indices in between.
- In max heap, every node contains greater or equal value element than all its descendants.
- In min heap, every node contains smaller value element than all its descendants.

Binary heap exercise

- Construct the binary heap from given array representation and find which one of the following array represents a binary max-heap?
 - 25, 14, 16, 13, 10, 8, 12
 - 25, 12, 16, 13, 10, 8, 14
 - 25, 14, 12, 13, 10, 8, 16
 - 25, 14, 13, 16, 10, 8, 12

Binary heap operations

- The common operation involved using heaps are:
 - **Heapify** → Process to rearrange the heap in order to maintain heap-property.
 - **Find-max (or Find-min)** → find a maximum item of a max-heap, or a minimum item of a min-heap, respectively.
 - **Insertion** → Add a new item in the heap.
 - **Deletion** → Delete an item from the heap.
 - **Extract Min-Max** → Returning and deleting the maximum or minimum element in max-heap and min-heap respectively.

Heapify

- It is a process to rearrange the elements of the heap in order to maintain the heap property. It is done when a certain node causes an imbalance in the heap due to some operation on that node.
- **The heapify can be done in two methodologies:**
 - **up_heapify()** → It follows the bottom-up approach. In this, we check if the nodes are following heap property by going in the direction of **rootNode** and if nodes are not following the heap property we do certain operations to let the tree follows the heap property.
 - **down_heapify()** → It follows the top-down approach. In this, we check if the nodes are following heap property by going in the direction of the leaf nodes and if nodes are not following the heap property we do certain operations to let the tree follows the heap property.

Down-Heapify Exercise

- Heapify is the process of creating a heap data structure from a binary tree.
It is used to create a Min-Heap or a Max-Heap.
- Let the input array be:

3	9	2	1	4	5
0	1	2	3	4	5

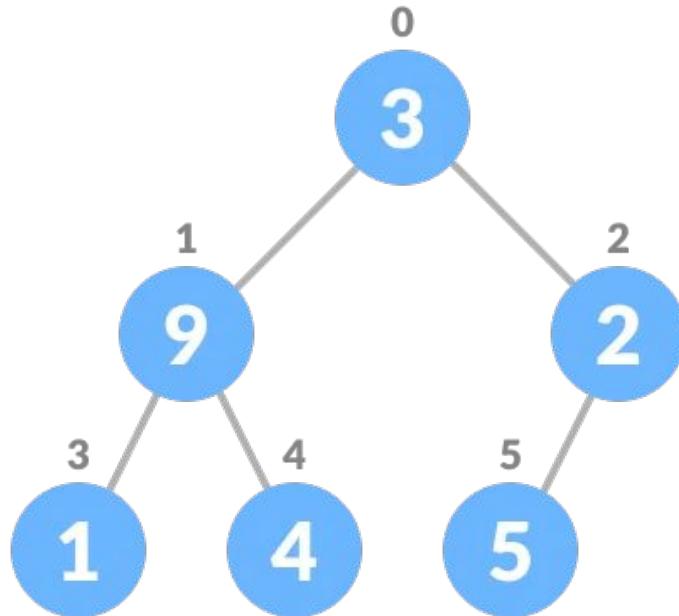
- Create a complete binary tree from the array

Down-Heapify Exercise

- Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.
- Let the input array be:

3	9	2	1	4	5
0	1	2	3	4	5

- Create a complete binary tree from the array.



Down-Heapify Exercise

- Steps to convert the binary tree into max-heap:

Step 1: Start from the first index of non-leaf node whose index is given by $n/2 - 1$.

Step 2: Set current element **i** as **largest**.

Step 3: The index of left child is given by $2i + 1$ and the right child is given by $2i + 2$. If **leftChild** is greater than **currentElement** (i.e. element at **i**th index), set **leftChildIndex** as **largest**. If **rightChild** is greater than element in **largest**, set **rightChildIndex** as **largest**.

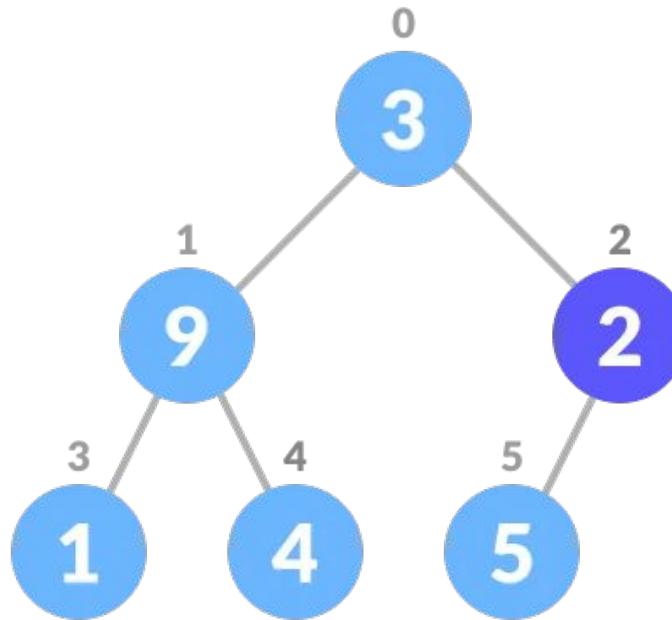
Step 4: Swap **largest** with **currentElement**

Step 5: Repeat step 1 to step 4 until subtrees are also heapified.

Down-Heapify Exercise

- Steps to convert the binary tree into max-heap:

Step 1: Start from the first index of non-leaf node whose index is given by $n/2 - 1$.

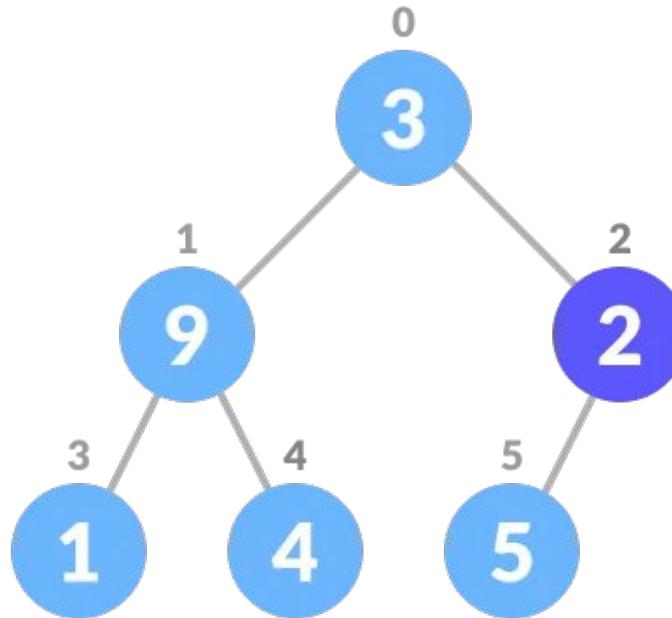


Down-Heapify Exercise

- Steps to convert the binary tree into max-heap:

Step 2: Set current element i as **largest**.

Here current element $i = 2 = \text{largest}$



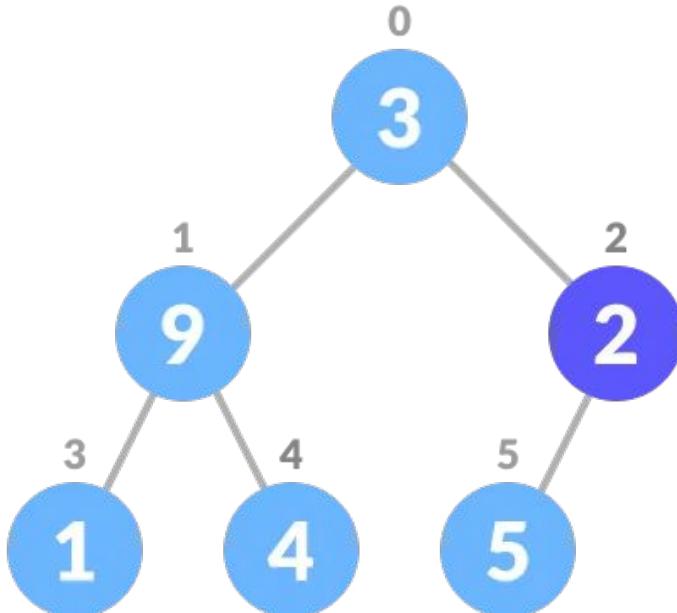
Down-Heapify Exercise

i = 2 = largest

Step 3:

- The index of left child is given by $2i + 1$ and the right child is given by $2i + 2$.
- If `leftChild` is greater than `currentElement` (i.e. element at `i`th index), set `leftChildIndex` as **largest**.
- If `rightChild` is greater than element in **largest**, set `rightChildIndex` as **largest**.

Here `leftChild 5 > 2`. Then, **largest** = 5



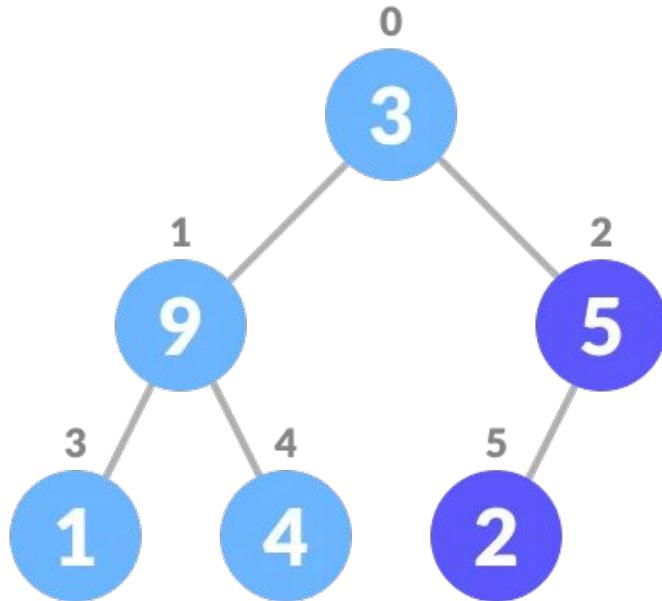
Down-Heapify Exercise

Largest = 5

CurrentElement = 2

Step 4: Swap largest with currentElement.

Repeat Steps for all subtrees while $i < 0$



Graphs

Graph Data Structure

- Graphs in data structures are non-linear data structures made up of a finite number of nodes or vertices and the edges that connect them.
- Graphs in data structures are used to address real-world problems in which it represents the problem area as a network like telephone networks, circuit networks, and social networks.
- For example, it can represent a single user as nodes or vertices in a telephone network, while the link between them via telephone represents edges.
- A pair (x,y) is referred to as an edge, which communicates that the x vertex connects to the y vertex.

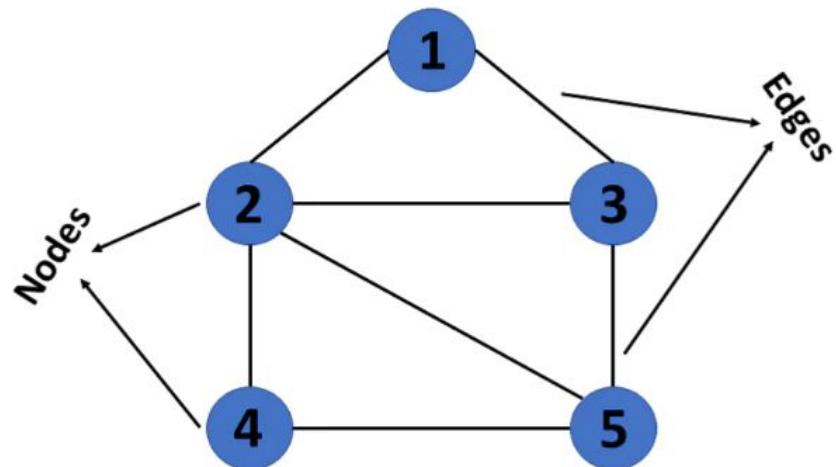
Graph Data Structure: visualization

- This graph has a set of vertices

$V = \{ 1, 2, 3, 4, 5 \}$ and

a set of edges

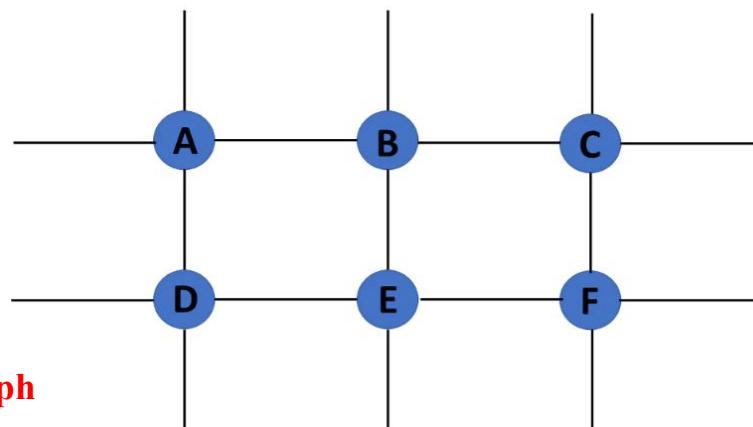
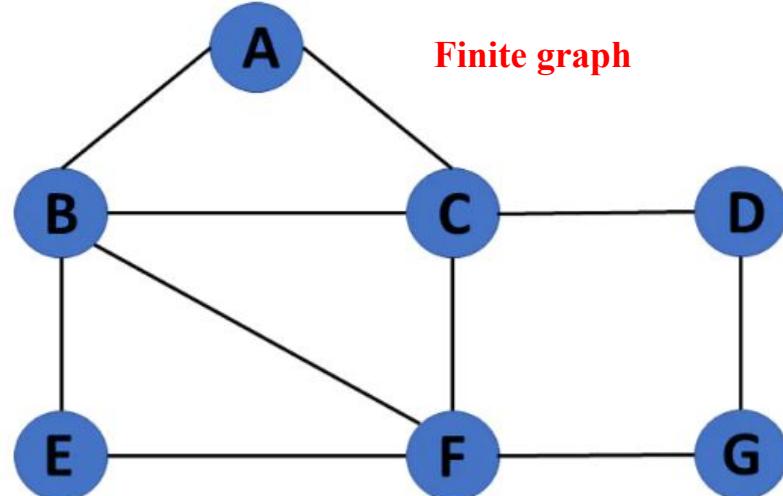
$E = \{ (1,2), (1,3), (2,3), (2,4), (2,5), (3,5), (4,5) \}$



Graph DS Types:

Finite and Infinite graph

- The graph $G=(V, E)$ is called a **finite** graph if the number of vertices and edges in the graph is limited in number.
- A graph $G=(V, E)$ is said to be **infinite** if it has infinite number of vertices as well as infinite number of edges.



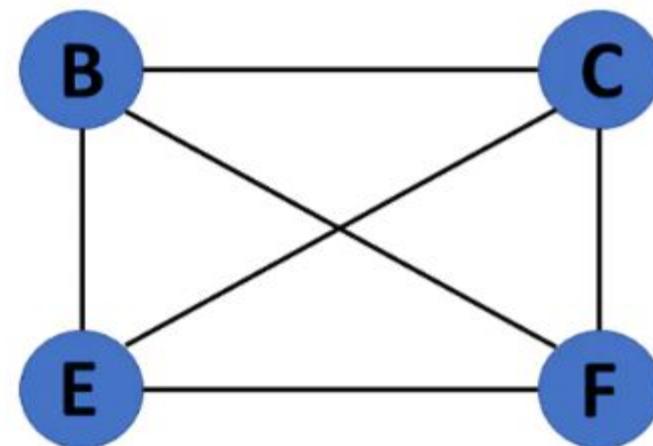
Graph DS Types:

Trivial and Simple graph

- A graph $G = (V, E)$ is trivial if it contains only a single vertex and no edges.
- If each pair of nodes or vertices in a graph $G = (V, E)$ has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices, depicting one-to-one interactions between two elements.



Trivial graph

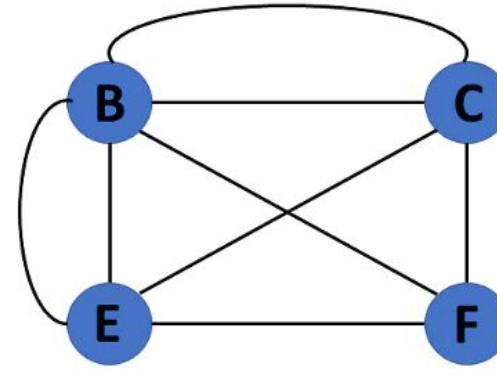


Simple graph

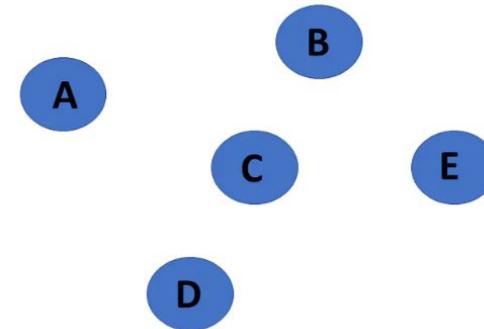
Graph DS Types:

Multi and Null graph

- If there are numerous edges between a pair of vertices in a graph $G = (V, E)$, the graph is referred to as a **multigraph**.
 - There are no self-loops in a **Multigraph**.
- It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph $G = (V, E)$ is a **null graph**.



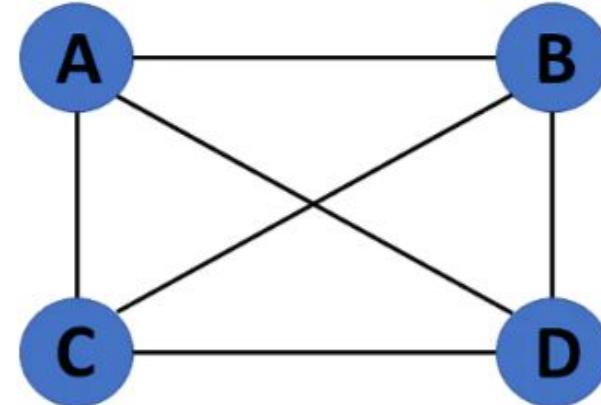
Multi graph



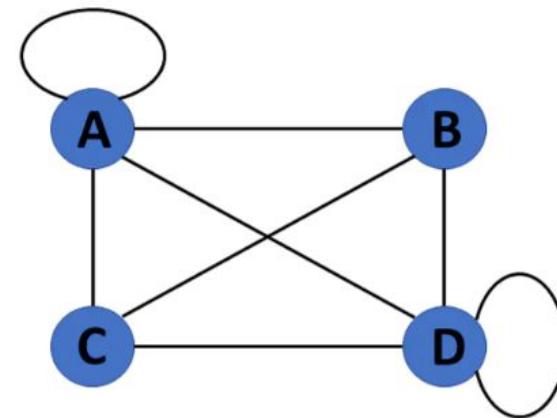
Null graph

Graph DS Types: Complete and Pseudo graph

- A graph $G = (V, E)$ is called a complete graph if there is a path from every vertex to every other vertex.
- If a graph $G = (V, E)$ contains a self-loop besides other edges, it is a pseudograph.



Complete graph

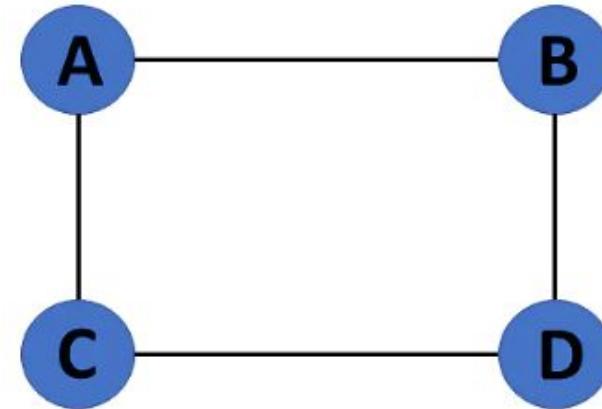


Null graph

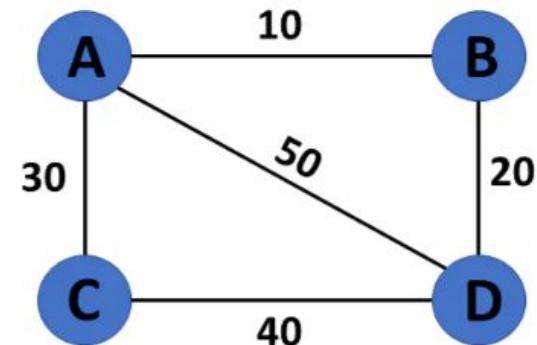
Graph DS Types:

Regular and Weighted graph

- If a graph $G = (V, E)$ is a simple graph with the same degree at each vertex, it is a regular graph. As a result, every whole graph is a **regular** graph.
- A graph $G = (V, E)$ is called a labeled or **weighted** graph because each edge has a value or weight representing the cost of traversing that edge.



Regular graph

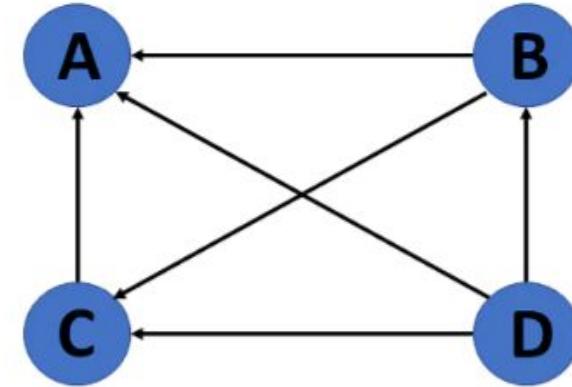


Weighted graph

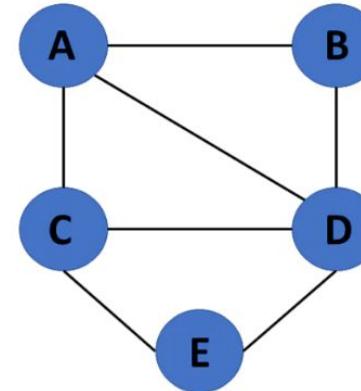
Graph DS Types:

Directed and Undirected graph

- A **directed** graph also referred to as a digraph, is a set of nodes connected by edges, each with a direction.
- An **undirected** graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction. You can form an undirected graph with a finite number of vertices and edges.



Directed graph

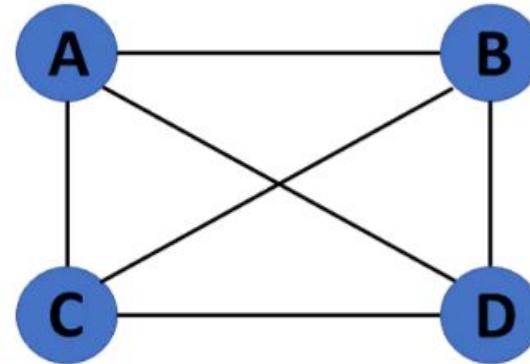


Undirected graph

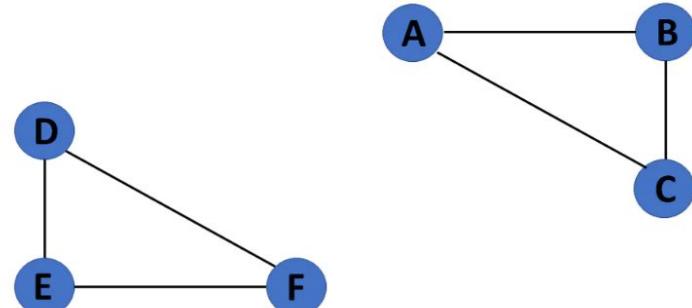
Graph DS Types:

Connected and Disconnected graph

- If there is a path between one vertex of a graph data structure and any other vertex, the graph is **connected**.
- When there is no edge linking the vertices, you refer to the null graph as a **disconnected** graph.



Connected graph

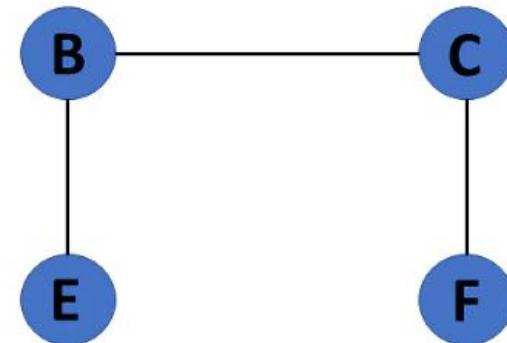
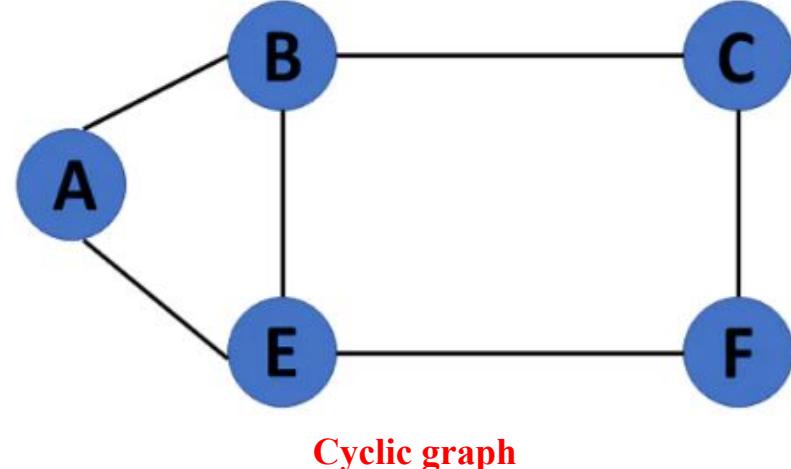


Disconnected graph

Graph DS Types:

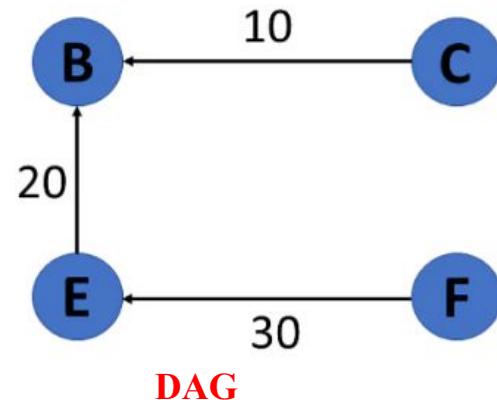
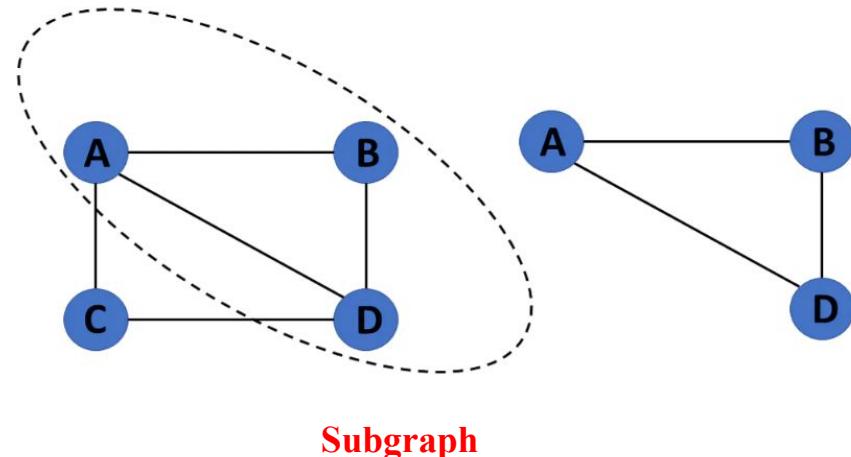
Cyclic and Acyclic graph

- If a graph contains at least one graph cycle, it is considered to be **cyclic**.
- When there are no cycles in a graph, it is called an **acyclic** graph.



Graph DS Types: Subgraph and Directed Acyclic graph

- The vertices and edges of a graph that are subsets of another graph are known as a **subgraph**.
- It is also known as a **directed acyclic graph (DAG)**, and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



Graph DS: Terminology-I

- An edge is (together with vertices) one of the two basic units out of which graphs are constructed.
- Each edge has two vertices to which it is attached, called its endpoints.
- Two vertices are called adjacent if they are endpoints of the same edge.
- Outgoing edges of a vertex are directed edges that the vertex is the origin.
- Incoming edges of a vertex are directed edges that the vertex is the destination.

Graph DS: Terminology-II

- The degree of a vertex in a graph is the total number of edges incident to it.
- In a directed graph, the out-degree of a vertex is the total number of outgoing edges, and the in-degree is the total number of incoming edges.
- A vertex with in-degree zero is called a source vertex, while a vertex with **out**-degree zero is called a sink vertex.
- An isolated vertex is a vertex with degree zero, which is not an endpoint of an edge.
- Path is a sequence of alternating vertices and edges such that the edge connects each successive vertex.

Graph DS: Terminology-III

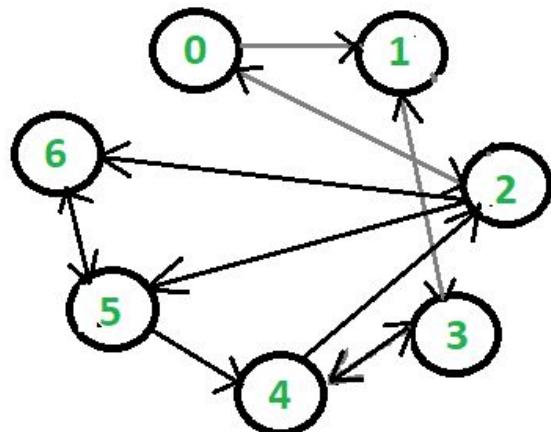
- A graph is Strongly Connected if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v .
- A directed graph is called Weakly Connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.
 - The vertices in a weakly connected graph have either out-degree or in-degree of at least 1.
- A bridge is an edge whose removal would disconnect the graph.
- Forest is a graph without cycles.
- Tree is a connected graph with no cycles.

Relationship between edges and vertices

- For a simple graph with m edges and n vertices, if the graph is
 - **directed**, then $m = n \times (n-1)$
 - **undirected**, then $m = n \times (n-1)/2$
 - **connected**, then $m = n-1$
 - a **tree**, then $m = n-1$
 - a **forest**, then $m = n-1$
 - **complete**, then $m = n \times (n-1)/2$
- Therefore, $O(m)$ may vary between $O(1)$ and $O(n^2)$, depending on how dense the graph is.

Graph: In-degree and Out-degree

- Given a directed graph, the task is to count the in and out degree of each vertex of the graph.



Vertex	In-degree	Out-degree
0	1	2
1	2	1
2	2	3
3	2	2
4	2	2
5	2	2
6	2	1

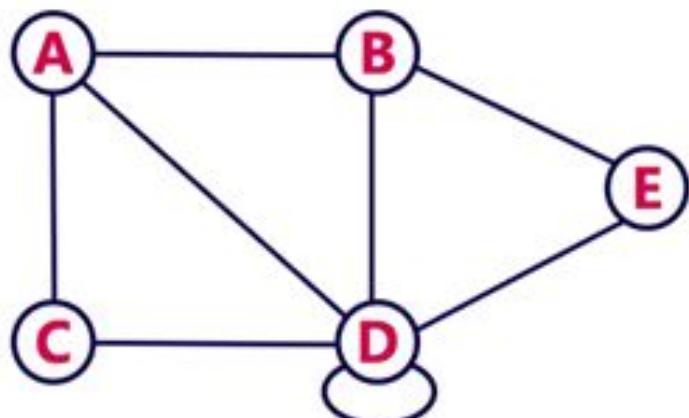
How do we represent graph in Computers?

- Graphs are commonly represented in two ways:
 - Adjacency matrix
 - Adjacency list

Adjacency Matrix

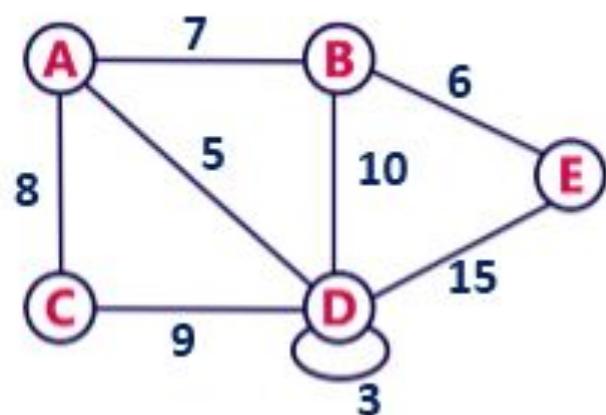
- Adjacency matrix
 - Adjacency matrix is a sequential representation.
 - An adjacency matrix is a 2D array of $V \times V$ vertices. Each row and column represent a vertex.
 - If the value of any element $a[i][j]$ is 1, it represents that there is an edge connecting vertex i and vertex j .
 - If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

Adjacency Matrix: undirected graph



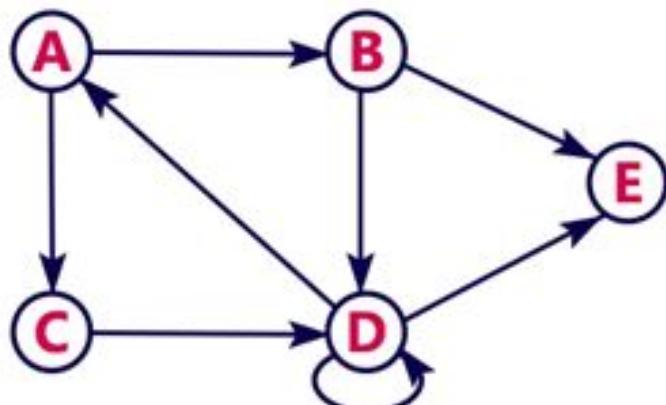
	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Adjacency Matrix: undirected weighted graph



	A	B	C	D	E
A	0	7	8	5	0
B	7	0	0	10	6
C	8	0	0	9	0
D	5	10	9	3	15
E	0	6	0	15	0

Adjacency Matrix: directed graph



	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

Adjacency Matrix: Implementation in C

```
#include <stdio.h>
#define V 4
void init(int arr[][V]) {
    int i, j;
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            arr[i][j] = 0;
}

// Add edges
void addEdge(int arr[][V], int i, int j) {
    arr[i][j] = 1;
    arr[j][i] = 1;
}

// Print the matrix
void printAdjMatrix(int arr[][V]) {
    int i, j;

    for (i = 0; i < V; i++) {
        printf("%d: ", i);
        for (j = 0; j < V; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}
```

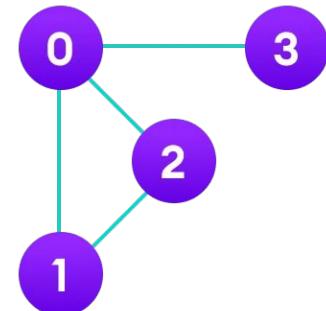
```
int main() {
    int adjMatrix[V][V];

    init(adjMatrix);
    addEdge(adjMatrix, 0, 1);
    addEdge(adjMatrix, 0, 2);
    addEdge(adjMatrix, 1, 2);
    addEdge(adjMatrix, 2, 0);
    addEdge(adjMatrix, 2, 3);

    printAdjMatrix(adjMatrix);

    return 0;
}
```

0:	0	1	1	0
1:	1	0	1	0
2:	1	1	0	1
3:	0	0	1	0



Adjacency Matrix: Pros

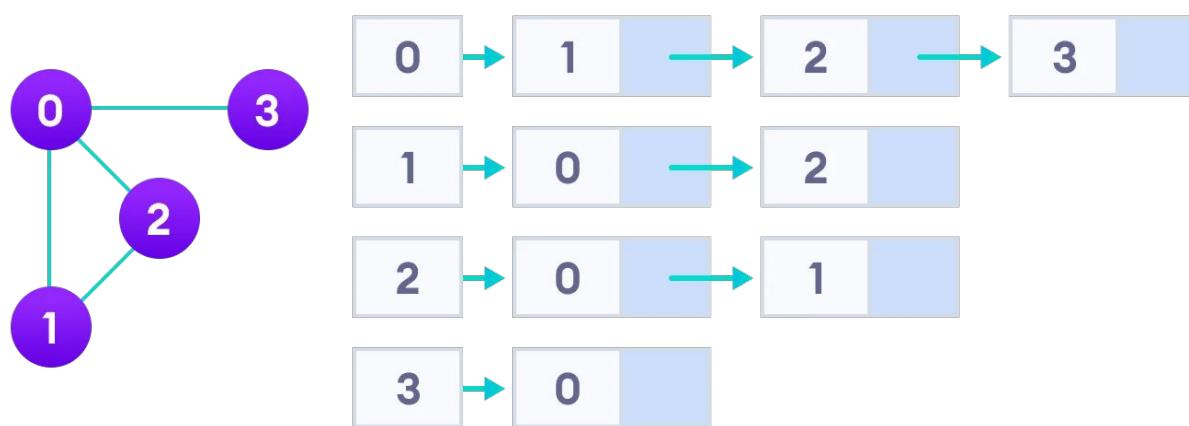
- The basic operations like adding an edge, removing an edge, and checking whether there is an edge from vertex i to vertex j are extremely time efficient, constant time operations.
- If the graph is dense and the number of edges is large, an adjacency matrix should be the first choice.
- The biggest advantage, however, comes from the use of matrices.
 - The recent advances in hardware enable us to perform even expensive matrix operations on the GPU.
 - By performing operations on the adjacent matrix, we can get important insights into the nature of the graph and the relationship between its vertices.

Adjacency Matrix: Cons

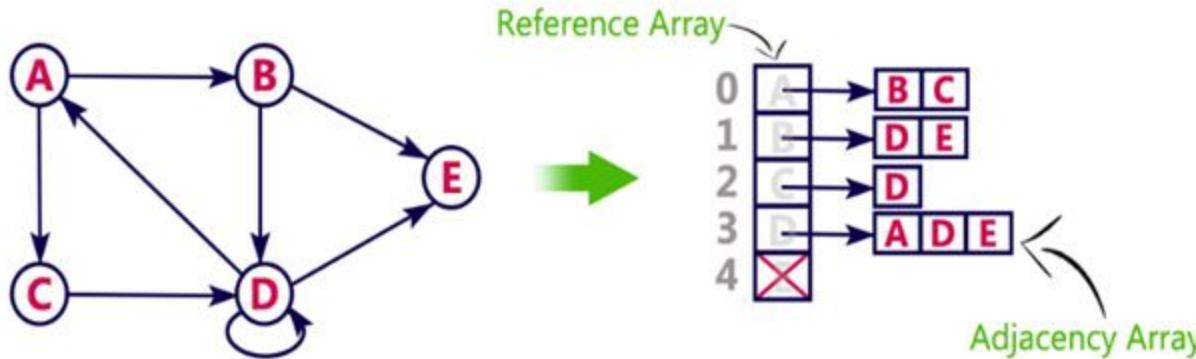
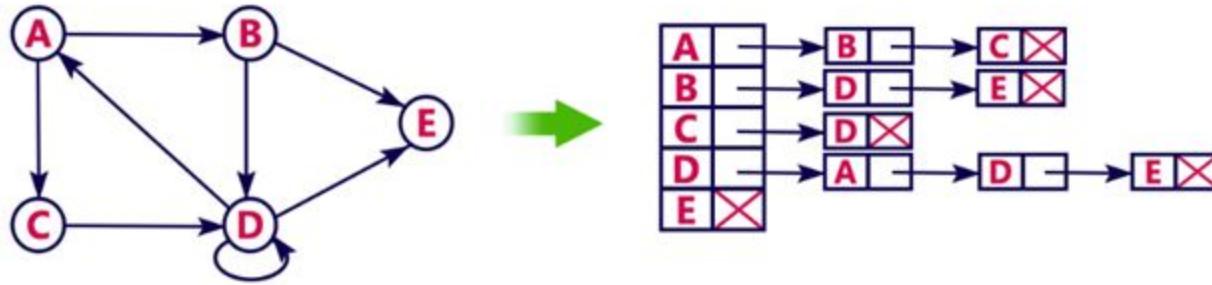
- The $V \times V$ space requirement of the adjacency matrix makes it a memory hog.
 - Graphs out in the wild usually don't have too many connections and this is the major reason why **adjacency lists** are the better choice for most tasks.
- While basic operations are easy, operations like **inEdges** and **outEdges** are expensive when using the adjacency matrix representation.

Adjacency List

- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.



Adjacency List: examples



Adjacency List: Pros

- Pros:
 - An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.
 - It also helps to find all the vertices adjacent to a vertex easily.
- Cons:
 - Finding the adjacent list is not quicker than the adjacency matrix because all the connected nodes must be first explored to find them.

Adjacency List: Representation in C

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int vertex;
    struct node* next;
};

struct node* createNode(int);

struct Graph {
    int numVertices;
    struct node** adjLists;
};
```

```
// Create a node
struct node* createNode(int v) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Create a graph
struct Graph* createAGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));

    int i;
    for (i = 0; i < vertices; i++)
        graph->adjLists[i] = NULL;

    return graph;
}
```

Adjacency List: Representation in C

```
// Add edge
void addEdge(struct Graph* graph, int s, int d) {
    // Add edge from s to d
    struct node* newNode = createNode(d);
    newNode->next = graph->adjLists[s];
    graph->adjLists[s] = newNode;

    // Add edge from d to s
    newNode = createNode(s);
    newNode->next = graph->adjLists[d];
    graph->adjLists[d] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
    int v;
    for (v = 0; v < graph->numVertices; v++) {
        struct node* temp = graph->adjLists[v];
        printf("\n Vertex %d\n: ", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}
```

```
int main() {
    struct Graph* graph = createAGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 0, 3);
    addEdge(graph, 1, 2);

    printGraph(graph);

    return 0;
}
```

Adjacency matrix: Time and Space Complexity

- Assuming the graph has n vertices, the time complexity to build such a matrix is $O(n^2)$.
 - To fill every value of the matrix we need to check if there is an edge between every pair (v_i, v_j) of vertices. The amount of such pairs of n given vertices is $(n*(n-1))/2$. That is why the time complexity of building the matrix is $O(n^2)$.
- The space complexity is also $O(n^2)$. Given a graph, to build the adjacency matrix, we need to create a square nxn matrix and fill its values with 0 and 1. It costs us $O(n^2)$ space.

Adjacency matrix: Pros and Cons

- The advantage of such representation is that we can check in **O(1)** time if there exists edge $e_{ij} = (v_i, v_j)$ by simply checking the value at i_{th} row and j_{th} column of our matrix.
- However, this approach has one big disadvantage. We need **O(n^2)** space all graphs.
 - In complete graph case, the matrix will be full of ones except the main diagonal, where all the values will be equal to zero. But, the complete graphs rarely happens in real-life problems.
 - So, if the target graph would contain many vertices and few edges, then representing it with the adjacency matrix is inefficient.

Adjacency list: Time and Space Complexity

- If m is the number of edges in a graph, then the time complexity of building such a list is $O(m)$.
- The space complexity is $O(n + m)$.
- But, in the worst case of a complete graph, which contains $(n*(n-1))/2$ edges, the time and space complexities reduce to $O(n^2)$.

Adjacency matrix: Pros and Cons

- As it was mentioned, complete graphs are rarely meet. Thus, this representation is more efficient if space matters. Moreover, we may notice, that the amount of edges doesn't play any role in the space complexity of the adjacency matrix, which is fixed. But, the fewer edges we have in our graph the less space it takes to build an adjacency list.
- However, there is a **major disadvantage** of representing the graph with the adjacency list. The access time to check whether edge $e_{ij} = (v_i, v_j)$ is present is constant in adjacency matrix, but is linear in adjacency list.
- In a complete graph with n vertices, for every vertex v_i the element of L_i would contain $n - 1$ elements, as every vertex is connected with every other vertex in such a graph.
- Therefore, the time complexity checking the presence of an edge in the adjacency list is **$O(n)$** .

Removing a vertex

- Let's suppose we want to remove v_i .
- If we use the adjacency matrix, we'll have to set all the entries in the i -th row and the i -th column to zero. Doing so will delete all the edges incident to v_i , effectively removing v_i from the graph. In total, we'll iterate over $2n$ cells, so the time complexity will be $O(n)$.
- On the other hand, removing a vertex from an adjacency list will cost more. To remove all the outgoing edges, we set to NULL the pointer to the v_i 's list, L_i . However, to delete all the occurrences of v_i from other nodes' lists, we have to iterate over all the other lists. In the worst case, each node will be connected to all the other vertices, so we'll traverse $(n-1)n$ list elements. Therefore, removing a vertex from the list representation of a graph is an $O(n^2)$ operation.

Removing an edge

- To remove an edge e_{ij} from an adjacency matrix M , we set $M[i,j]$ to zero. If the graph is symmetric, we do the same with $M[j, i]$. Accessing a cell in the matrix is an **$O(1)$** operation, so the complexity is **$O(1)$** in the best-case, average-case, and worst-case scenarios.
- If we store the graph as an adjacency list, the complexity of deleting an edge is **$O(n)$** . That's because, in the worst case, we traverse the whole list L_i to remove v_j from it. If the graph is symmetric, we do the same with L_j , removing v_i from it. In total, we iterate over no more than $2n$ elements, so the complexity is **$O(n)$** .

Graph Searching Techniques

- Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops.
- That means using graph traversal we visit all the vertices of the graph without getting into looping path.
- There are two graph traversal techniques and they are as follows...
 - DFS (Depth First Search)
 - BFS (Breadth First Search)

BFS Vs DFS

- For traversal we follow the following terms:
 - Visiting a vertex
 - Exploring a vertex
- BFS (Breadth First Search)

A

A, B, D, E

A, B, D, E, C

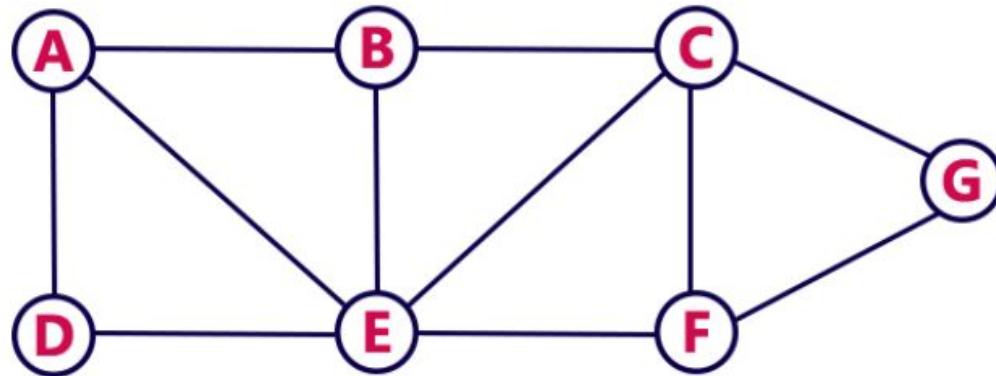
A, B, D, E, C,

A, B, D, E, C, F

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

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

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



Depth First Search

A

A, B

A, B, C

A, B, C, E,

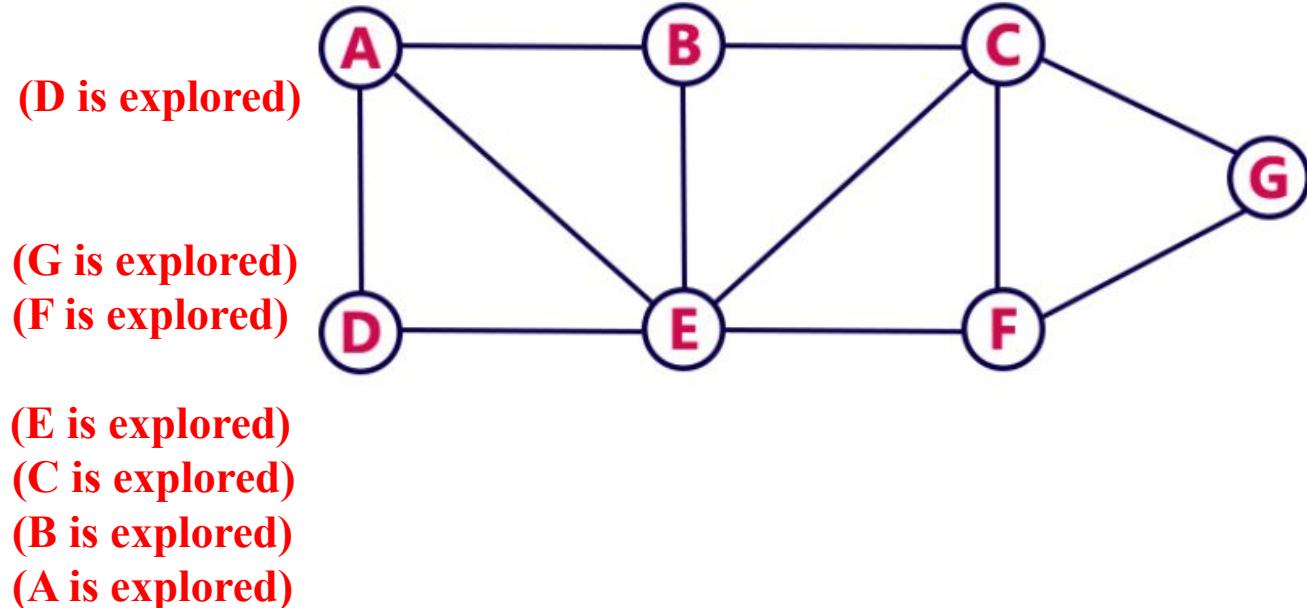
A, B, C, E, D

A, B, C, E, D

A, B, C, E, D, F

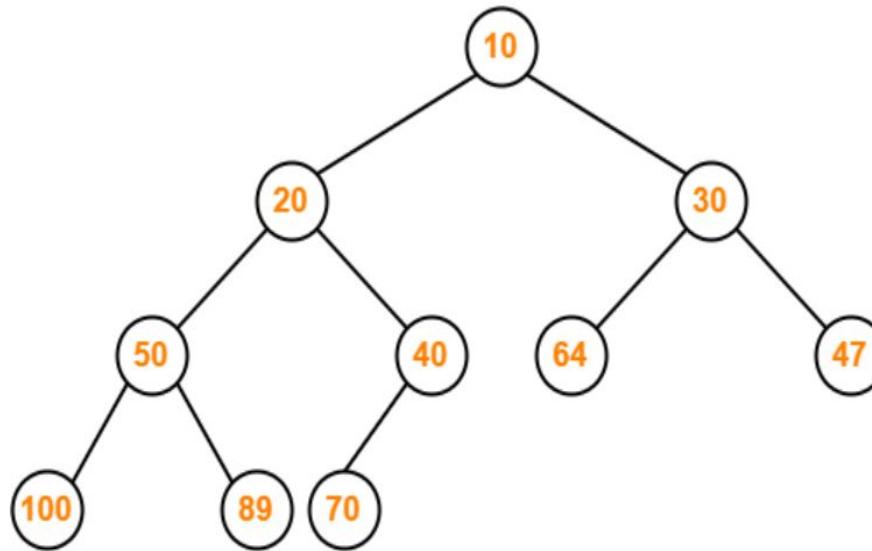
A, B, C, E, D, F, G

- For traversal we follow the following terms:
 - Visiting a vertex
 - Exploring a vertex



BFS Vs DFS

- Consider the following graph and perform BFS and DFS



BFS using Queue

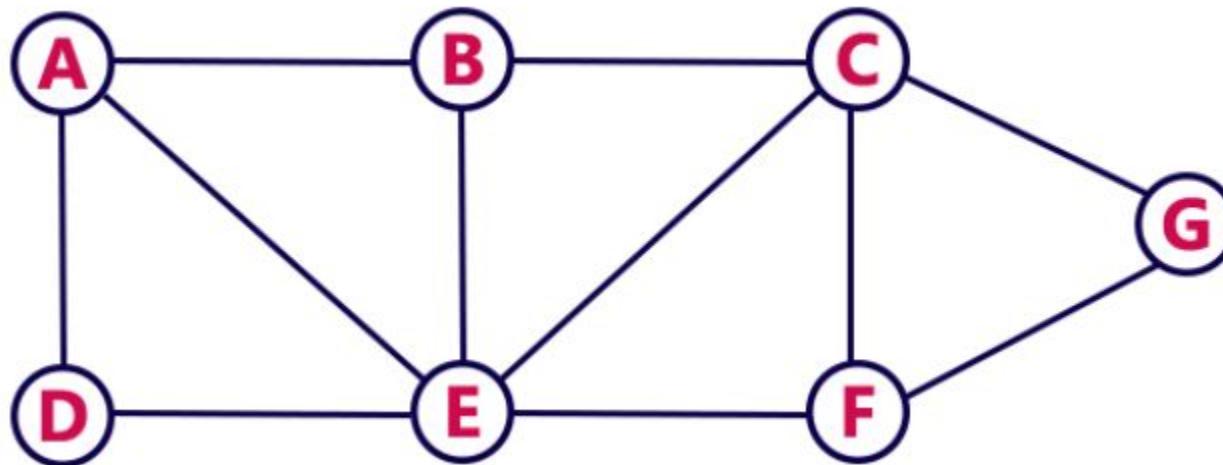
- The Queue data structure is used for the Breadth First Search traversal.
- When we use the BFS algorithm for the traversal in a graph, we can consider any node as a root node.
- BFS traversal of a graph produces a **spanning tree** as final result.
 - **Spanning Tree** is a graph without loops.
- We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

BFS using Queue: Algorithm

- **Step 1** - Define a Queue of size total number of vertices in the graph.
- **Step 2** - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
- **Step 3** - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4** - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5** - Repeat steps 3 and 4 until queue becomes empty.
- **Step 6** - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

BFS using Queue: Example

- Consider the following graph for BFS traversal



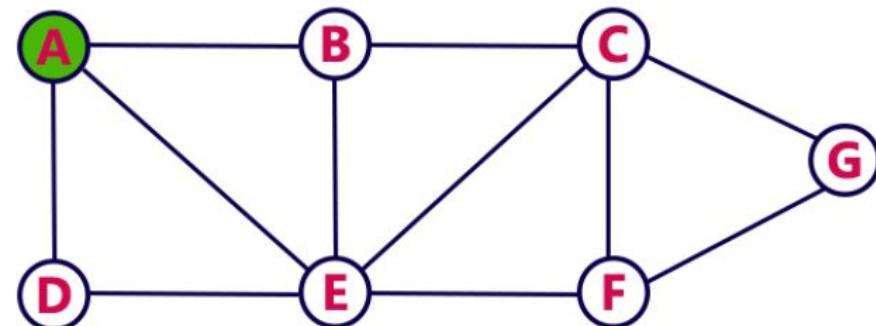
BFS using Queue: Example

- Step 1: Select vertex A as starting point (visit A). Insert A into the Queue.

BFS: A

Queue

A						
---	--	--	--	--	--	--

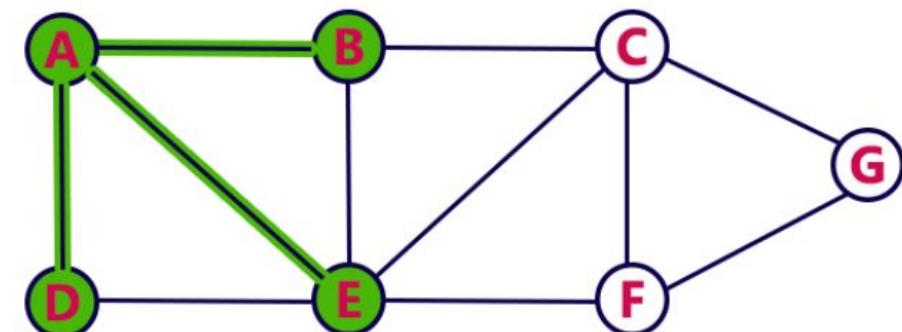


BFS using Queue: Example

- **Step 2:** Delete the vertex **A** from queue and start exploring it.
 - Visit all new vertices of **A** which are not visited that are **B**, **D**, **E**. Add them into queue.

BFS: A, D, E, B

Queue

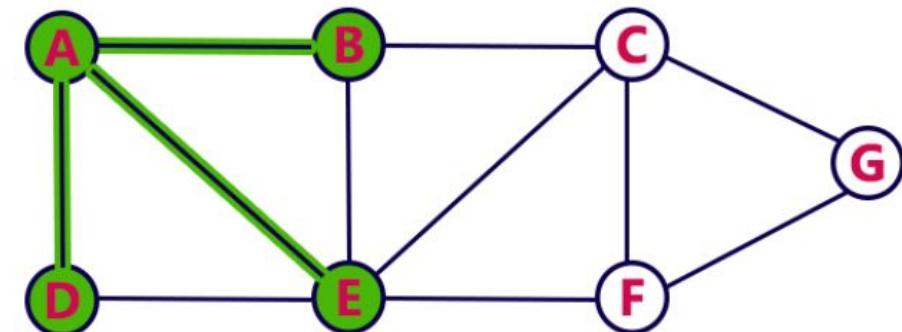


BFS using Queue: Example

- **Step 3:** Select next vertex (**D**) for exploration from queue and delete that vertex.
 - Visit all new vertices of **D** which are not visited that are NULL.

BFS: A, D, E, B

Queue

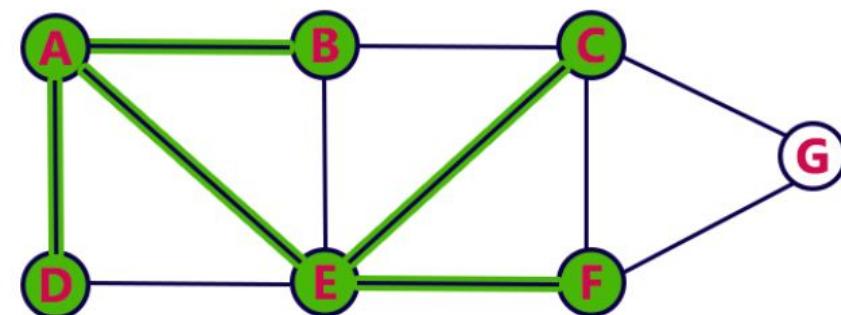


BFS using Queue: Example

- Step 4: Select next vertex (E) for exploration from queue and delete that vertex.
 - Visit all new vertices of E which are not visited that are C and F.

BFS: A, B, D, E, C, F

Queue

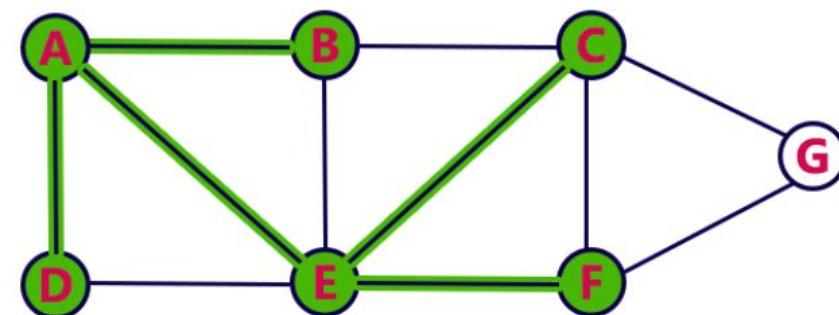


BFS using Queue: Example

- **Step 5:** Select next vertex (**B**) for exploration from queue and delete that vertex.
 - Visit all new vertices of **B** which are not visited that None.

BFS: A, B, D, E, C, F

Queue

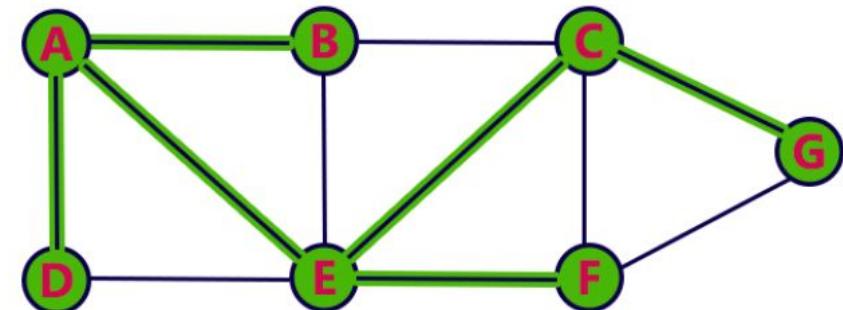
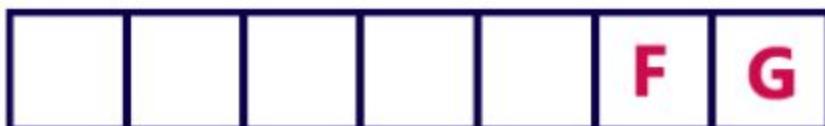


BFS using Queue: Example

- **Step 6:** Select next vertex (**C**) for exploration from queue and delete that vertex.
 - Visit all new vertices of **C** which are not visited that is **G**.

BFS: A, B, D, E, C, F, G

Queue

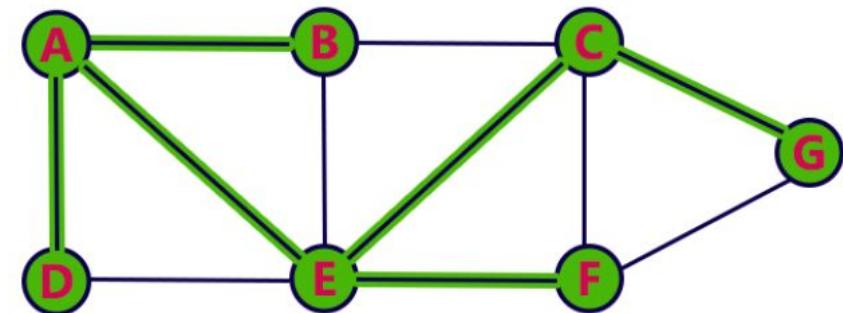


BFS using Queue: Example

- **Step 7:** Select next vertex (**F**) for exploration from queue and delete that vertex.
 - Visit all new vertices of **F** which are not visited that are None.

BFS: A, B, D, E, C, F, G

Queue

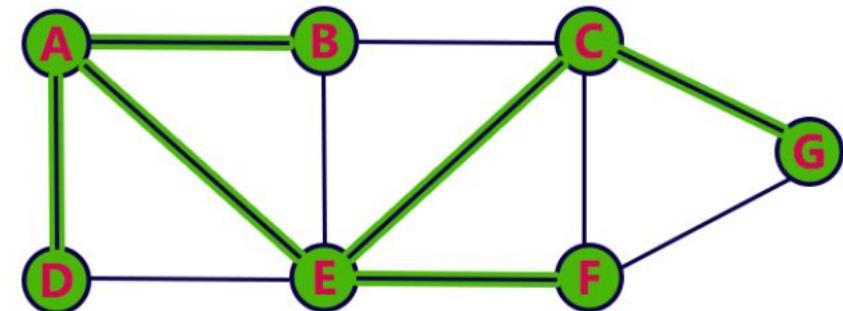


BFS using Queue: Example

- **Step 7:** Select next vertex (**G**) for exploration from queue and delete that vertex.
 - Visit all new vertices of **G** which are not visited that are None.

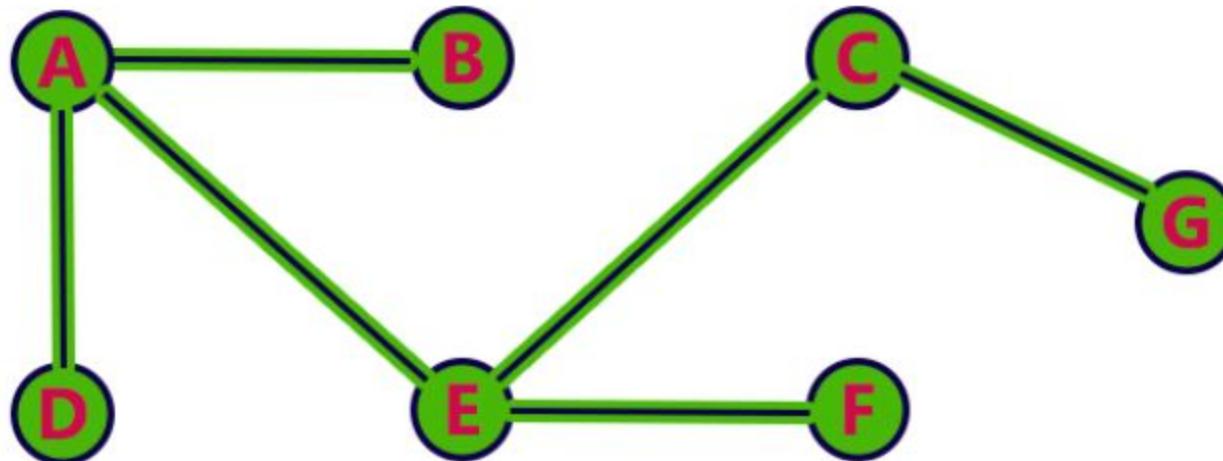
BFS: A, B, D, E, C, F, G

Queue



BFS using Queue: Example

- Queue became empty so stop BFS process. Final result of BFS is a spanning tree as shown below:
- **BFS:** A, B, D, E, C, F, G



DFS using Stack

- DFS traversal of a graph produces a spanning tree as final result.
- Spanning Tree is a graph without loops.
- We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

Note: We use backtracking in DFS

Backtracking is coming back to the vertex from which we reached the current vertex.

DFS using Stack: Algorithm

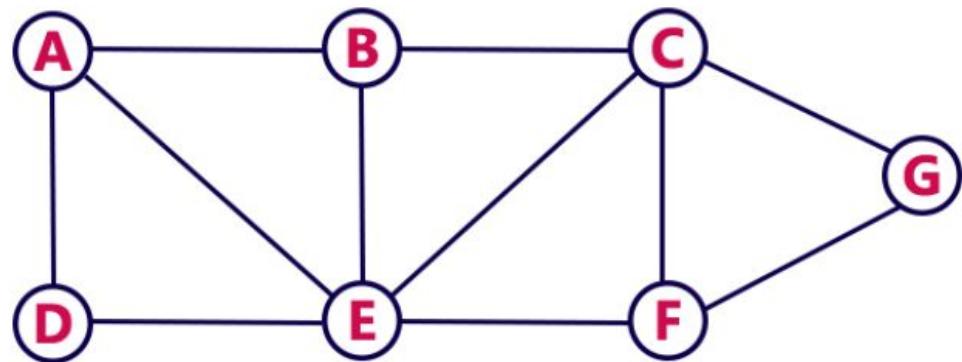
- **Step 1** - Define a Stack of size total number of vertices in the graph.
- **Step 2** - Select any vertex as starting point for traversal. Visit that vertex and push it onto the Stack.
- **Step 3** - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it onto the stack.
- **Step 4** - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5** - When there is no new vertex to visit then use backtracking and pop one vertex from the stack.
- **Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7** - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 1: Select a vertex A as starting point (Visit A).

Push A onto the stack.

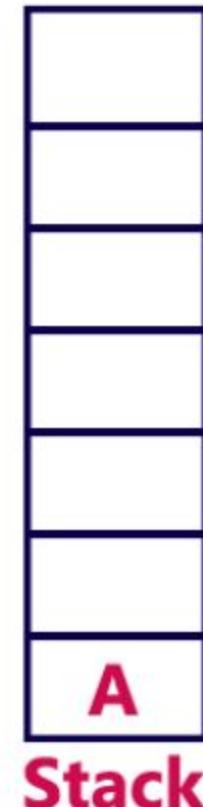
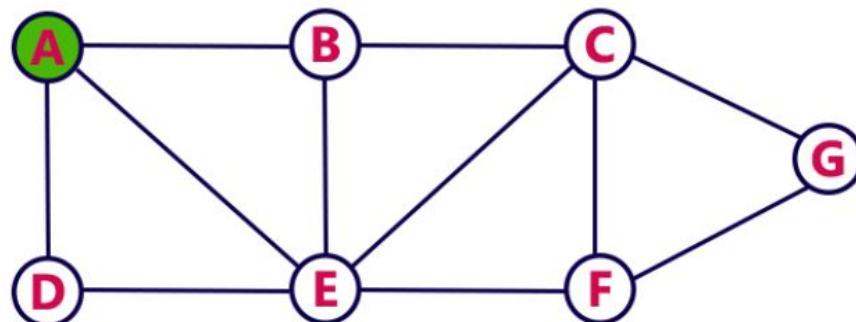


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 1: Select a vertex A as starting point (Visit A).

Push A onto the stack.

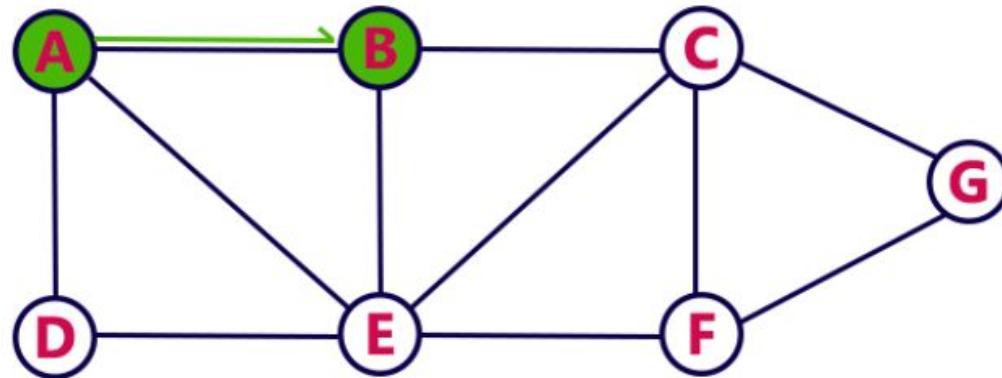


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 2: Visit any adjacent vertex of A which is not visited (Visit B).

Push newly visited vertex B onto the stack.

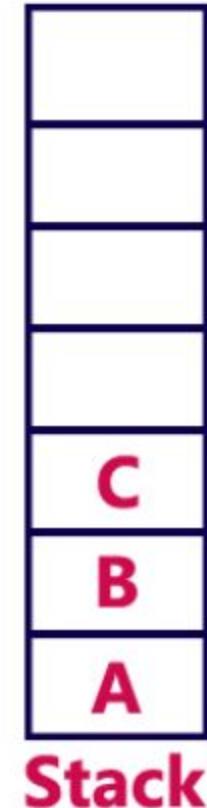
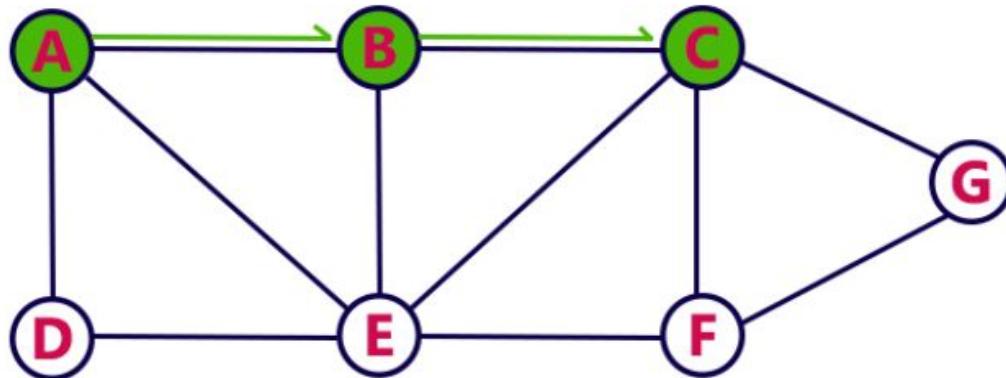


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 3: Visit any adjacent vertex of B which is not visited (Visit C).

Push newly visited vertex C onto the stack.

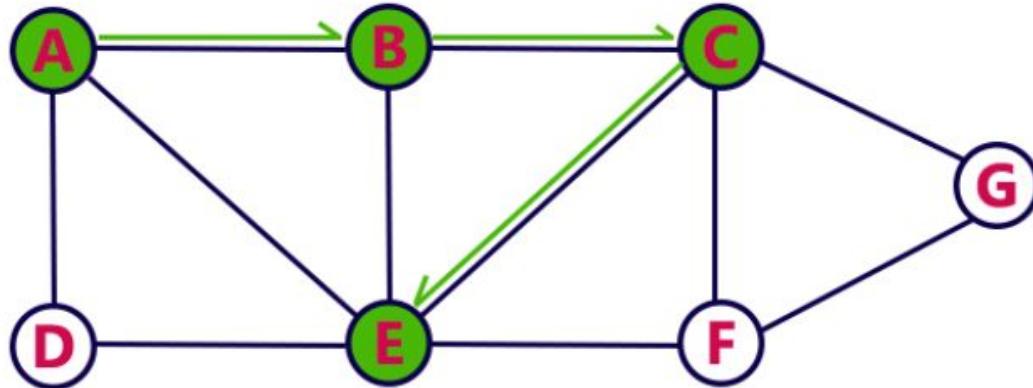


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 4: Visit any adjacent vertex of C which is not visited (Visit E).

Push newly visited vertex E onto the stack.

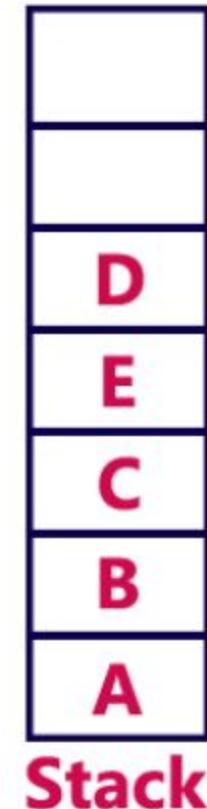
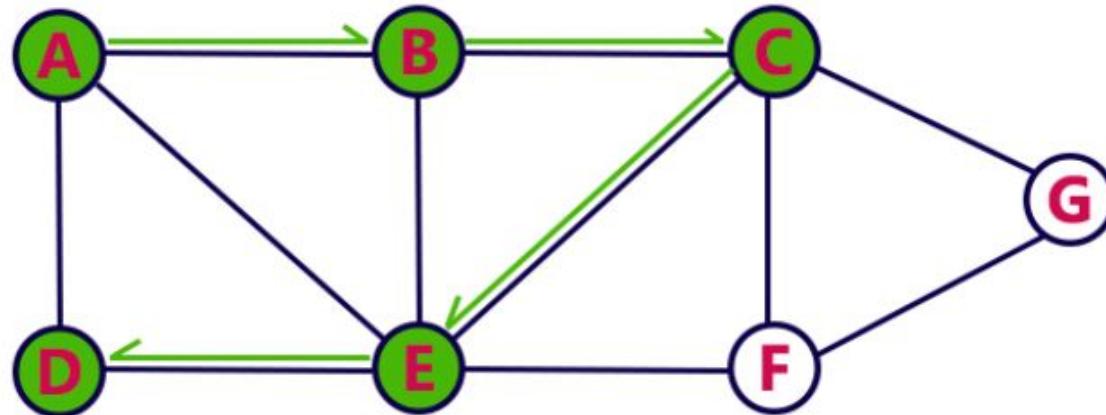


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 5: Visit any adjacent vertex of E which is not visited (Visit D).

Push newly visited vertex D onto the stack.

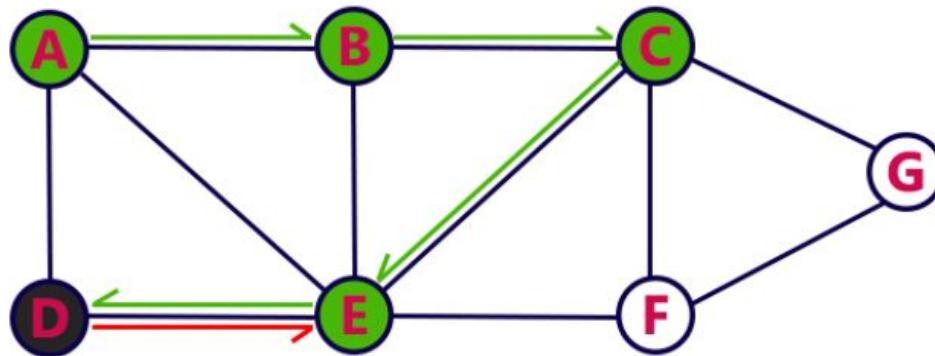


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 6: Visit any adjacent vertex of D which is not visited. All are visited.

Backtrack, and Pop D from the stack.

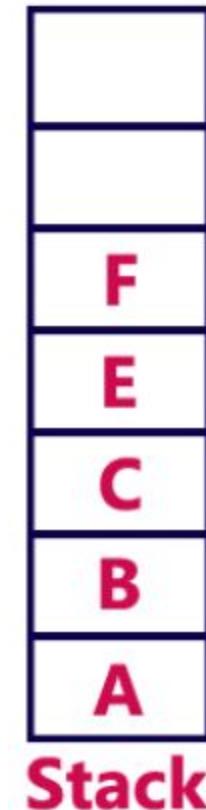
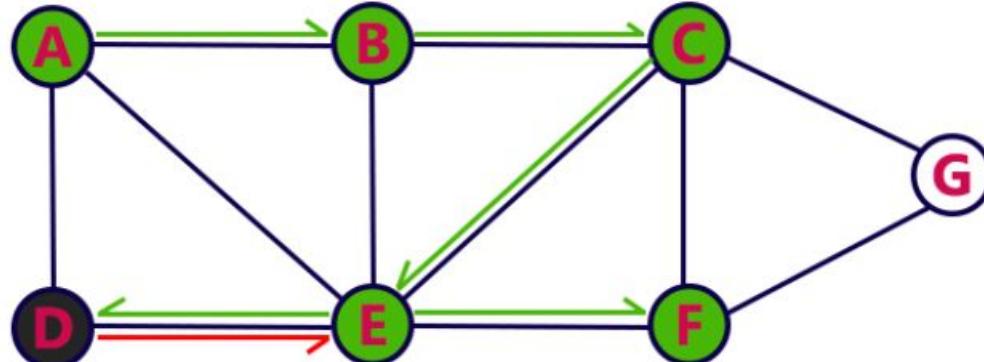


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 7: Visit any adjacent vertex of E which is not visited. Visit F.

Push newly visited vertex F onto the stack.

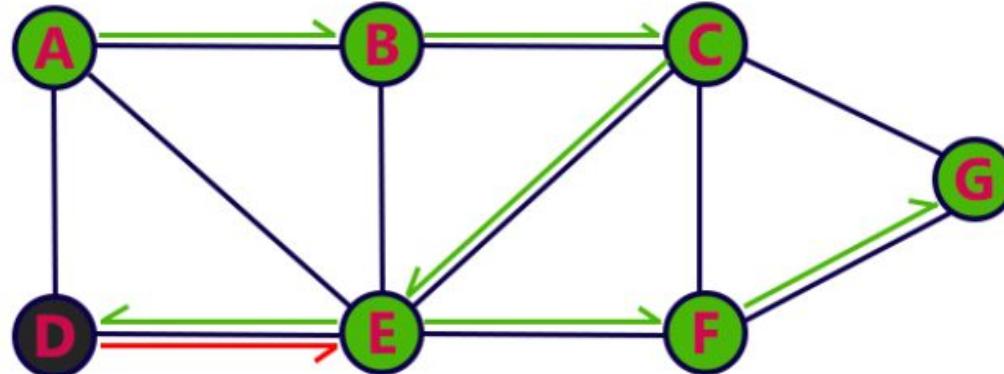


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 8: Visit any adjacent vertex of F which is not visited. Visit G.

Push newly visited vertex G onto the stack.

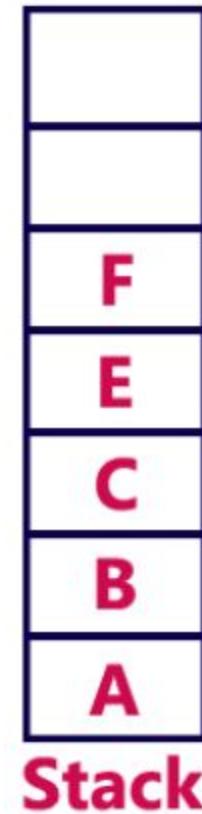
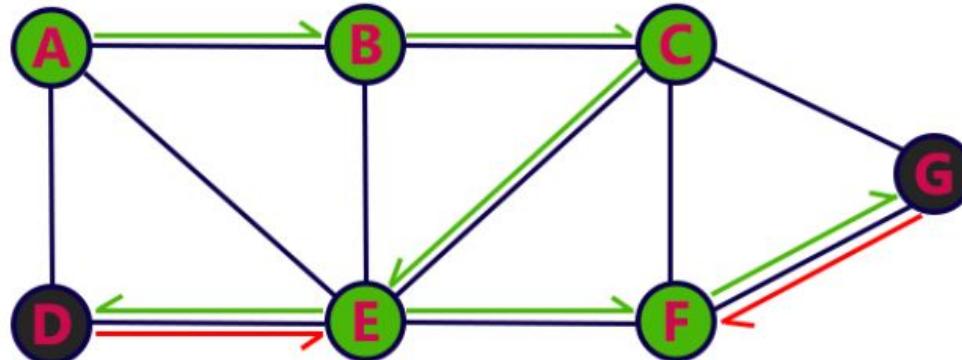


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 9: Visit any adjacent vertex of G which is not visited. All are visited.

Backtrack, and Pop G from the stack.

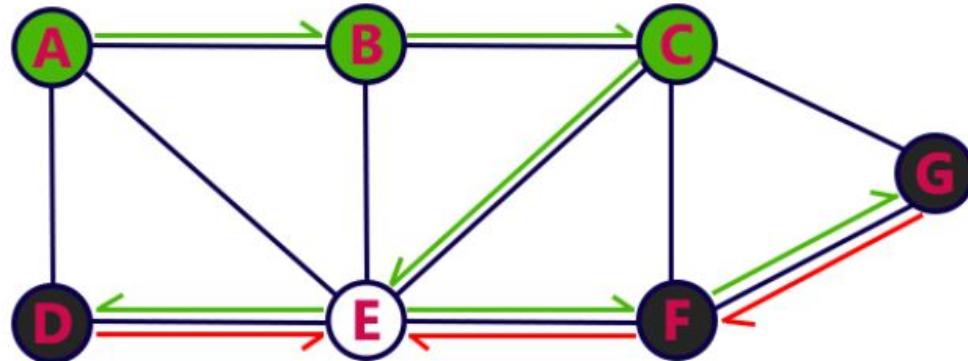


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 10: Visit any adjacent vertex of F which is not visited. All are visited.

Backtrack, and Pop F from the stack.

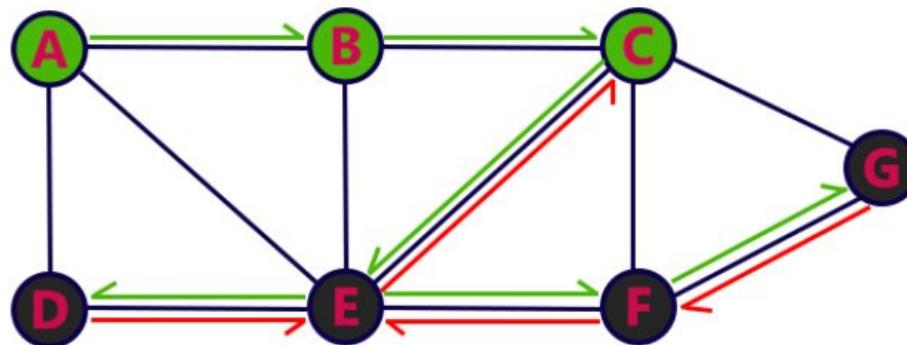


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 11: Visit any adjacent vertex of E which is not visited. All are visited.

Backtrack, and Pop E from the stack.

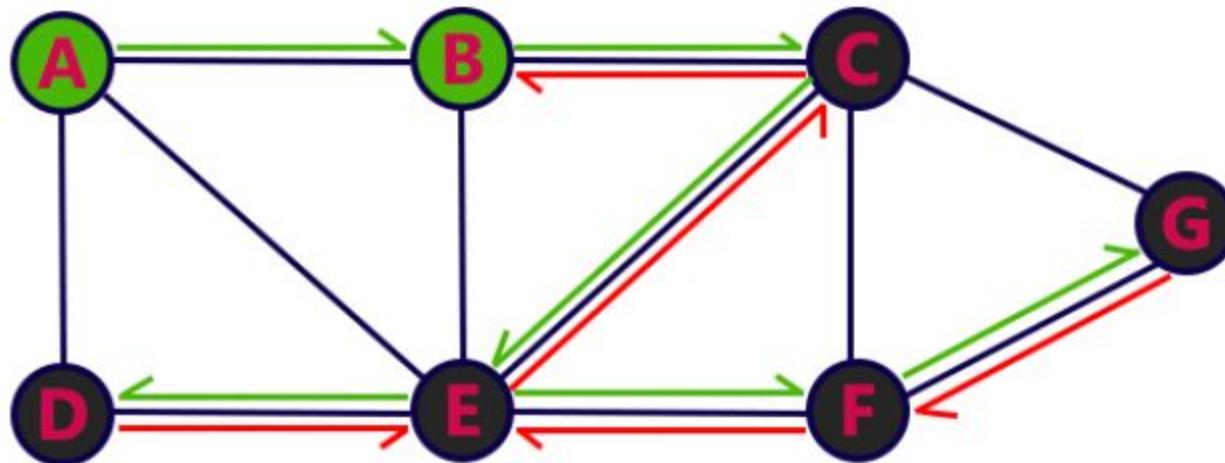


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 12: Visit any adjacent vertex of C which is not visited. All are visited.

Backtrack, and Pop C from the stack.

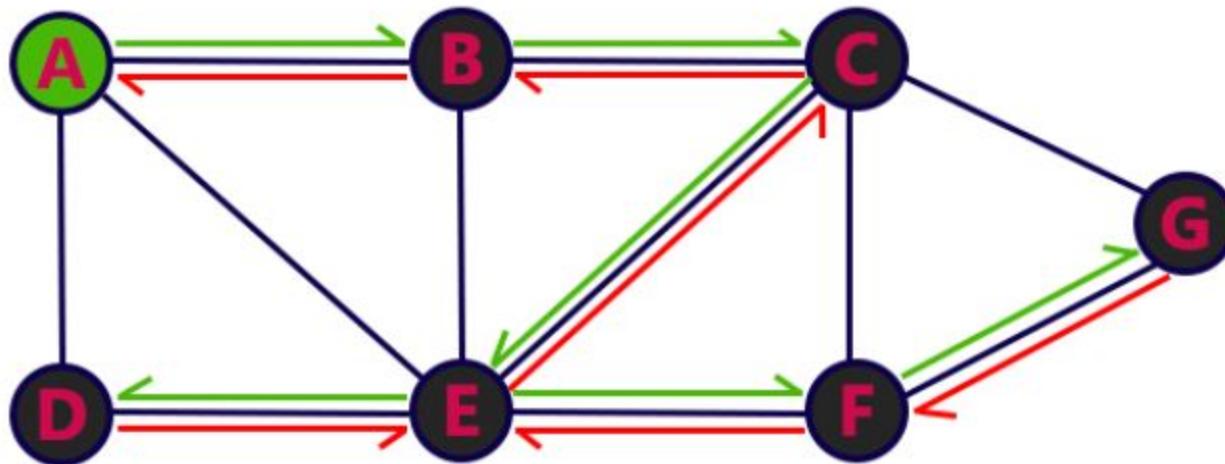


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 13: Visit any adjacent vertex of B which is not visited. All are visited.

Backtrack, and Pop B from the stack.

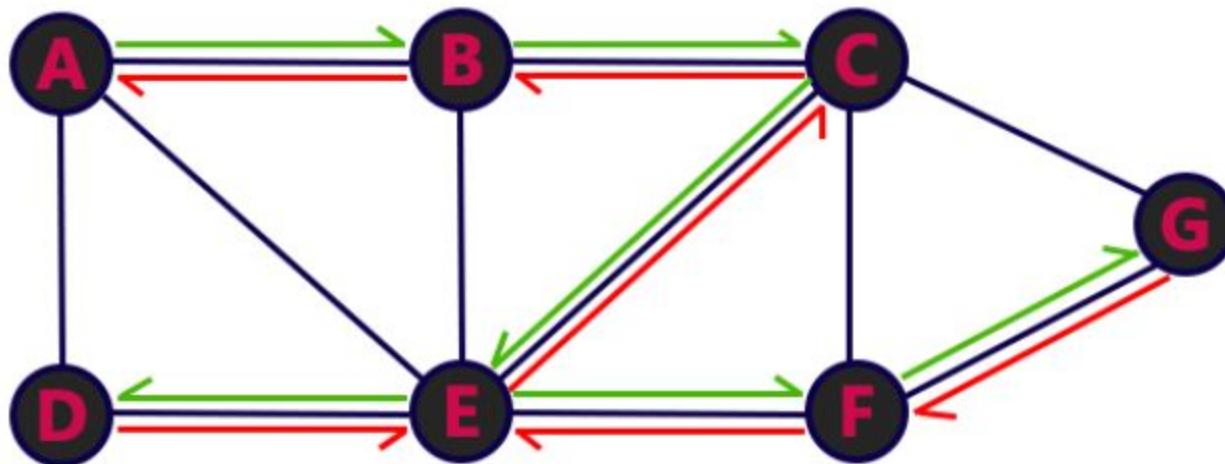


DFS using Stack: Example

- Consider the following graph for DFS traversal:

Step 14: Visit any adjacent vertex of A which is not visited. All are visited.

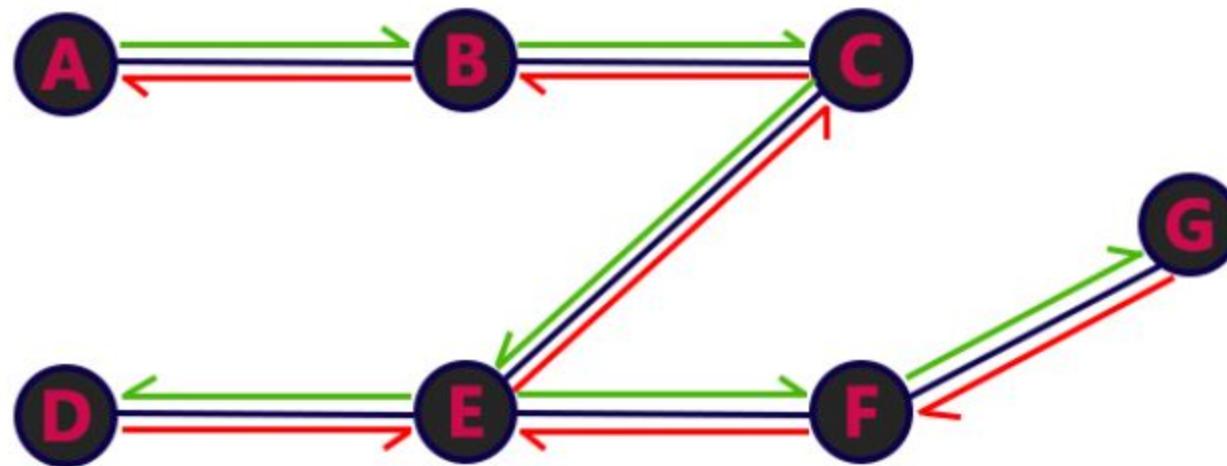
Backtrack, and Pop A from the stack.



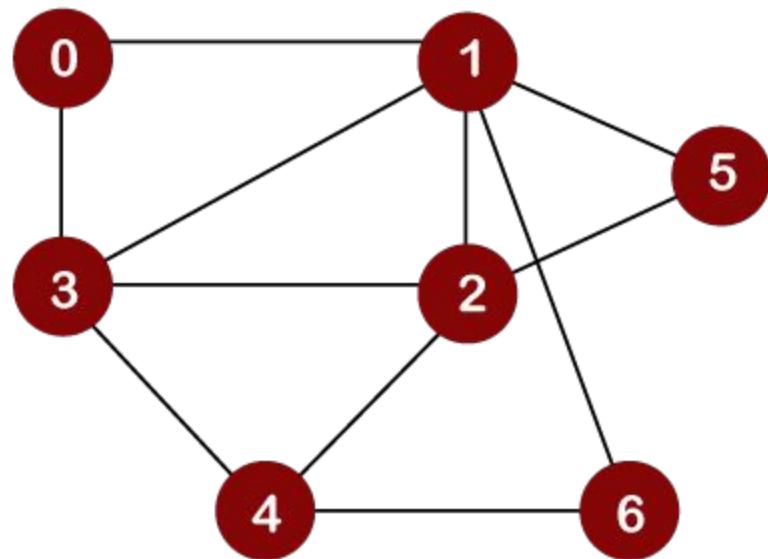
DFS using Stack: Example

- Stack became empty.

Final result of DFS traversal is following spanning tree.



BFS and DFS: exercise

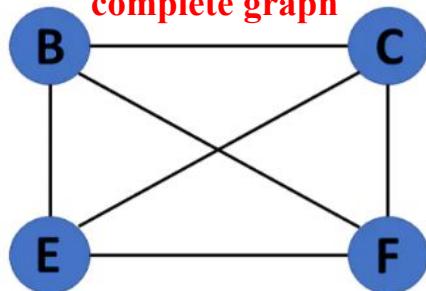


Spanning trees

Question: How many minimum number of edges are required to connect/traverse all vertices in a graph?

Answer: Spanning tree tells that how many minimum number of edges are required to connect all vertices in a graph.

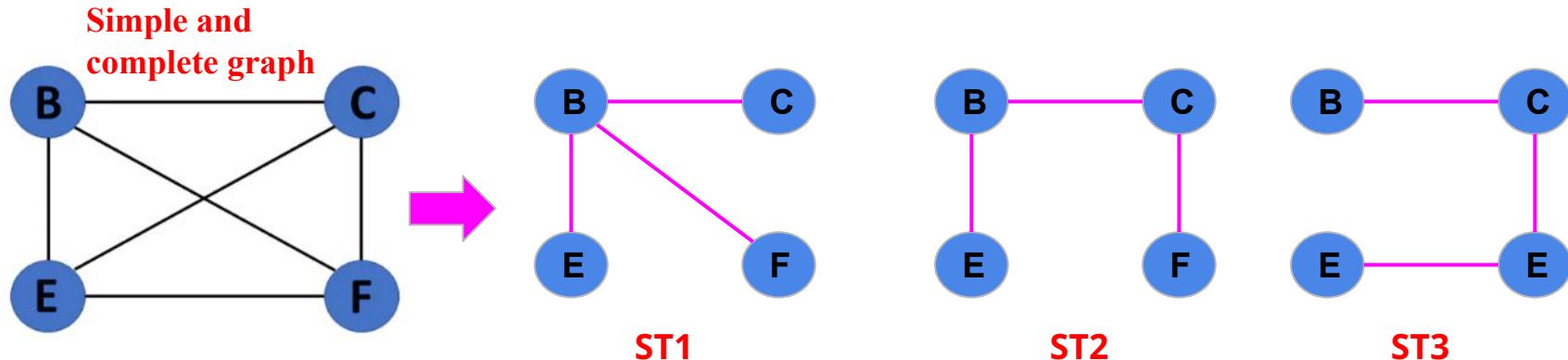
Simple and
complete graph



Spanning trees

Question: How many minimum number of edges are required to connect/traverse all vertices in a simple and complete graph having ‘n’ vertices?

Answer: It turns out that, if number of vertices is ‘n’ in a complete graph then its corresponding **spanning tree** will have ‘n-1’ edges only.



Minimum cost spanning tree

Problem: Given a weighted graph, find the spanning tree with least weight

Solution: We can apply greedy method for solving this problem.

- There are two methods which can solve this problem:
 - [Prims](#) algorithm
 - [Kruskal](#) algorithm

MST: Prim's algorithm

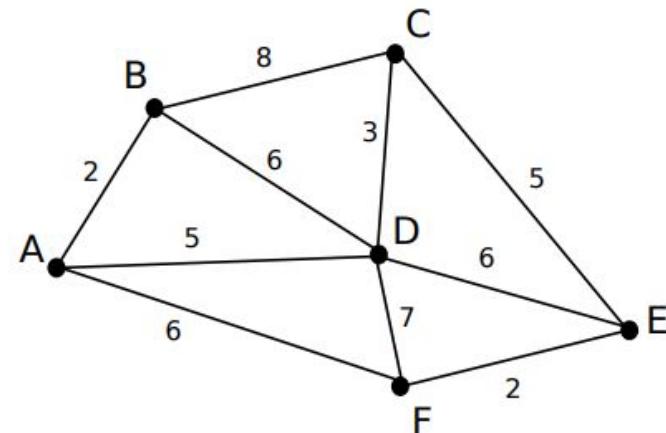
Prim's algorithm is an algorithm for determining the minimal spanning tree in a connected graph.

Algorithm:

Step 1: Choose any starting vertex. Look at all edges connecting to the vertex and choose the one with the lowest weight and add this to the tree.

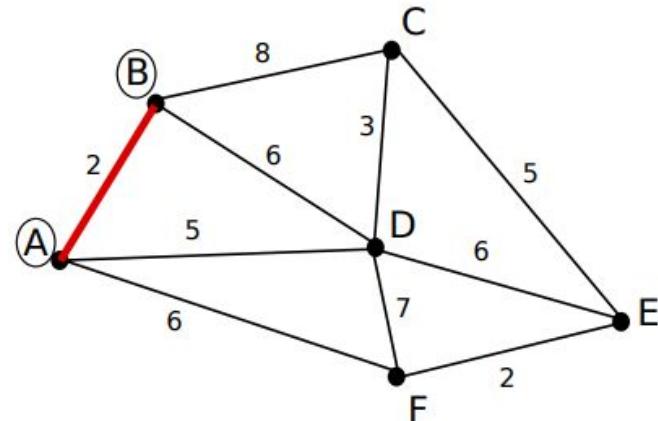
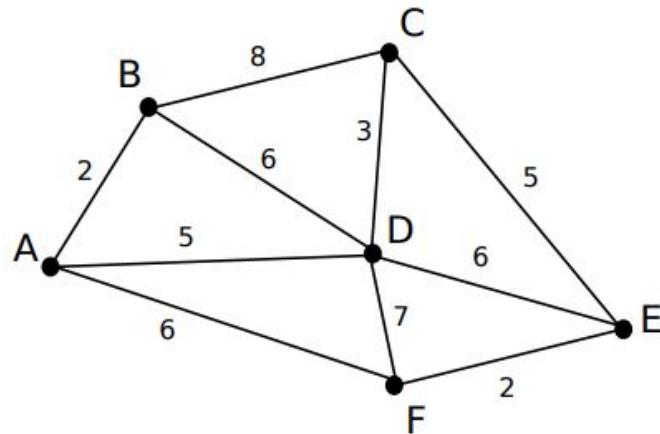
Step 2: Look at all edges connected to the tree that do not have both vertices in the tree. Choose the one with the lowest weight and add it to the tree.

Step 3: Repeat step 2 until all vertices are in the tree.



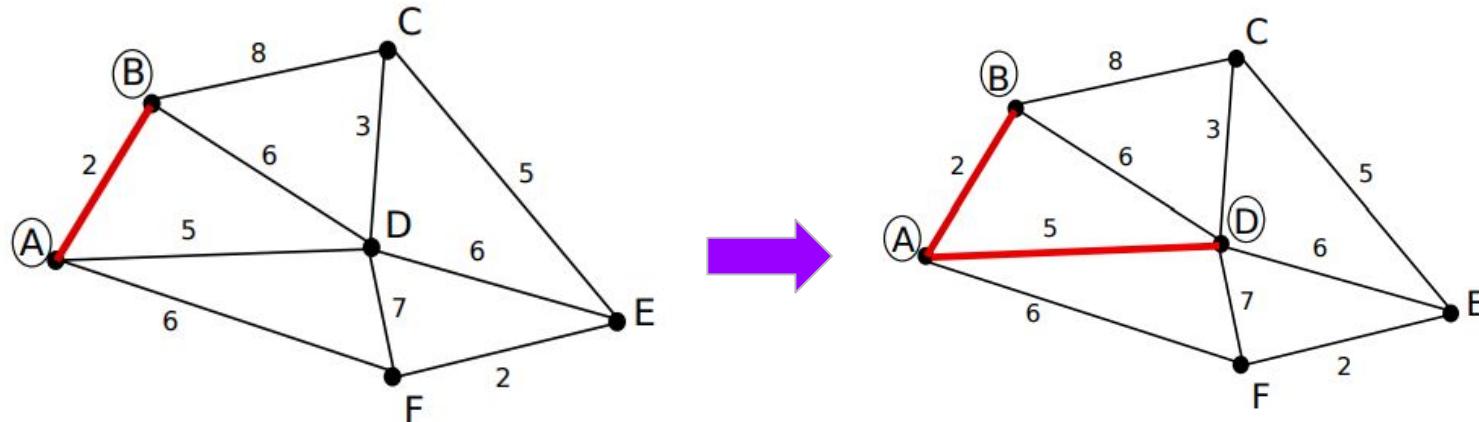
MST: Prim's algorithm

Choose vertex A. **Choose edge with lowest weight: (A,B)**



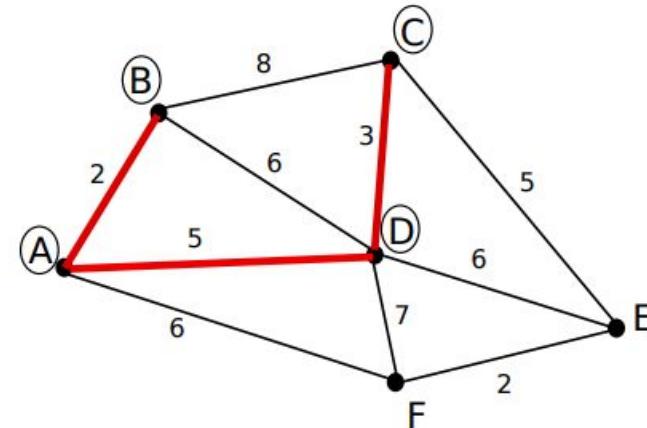
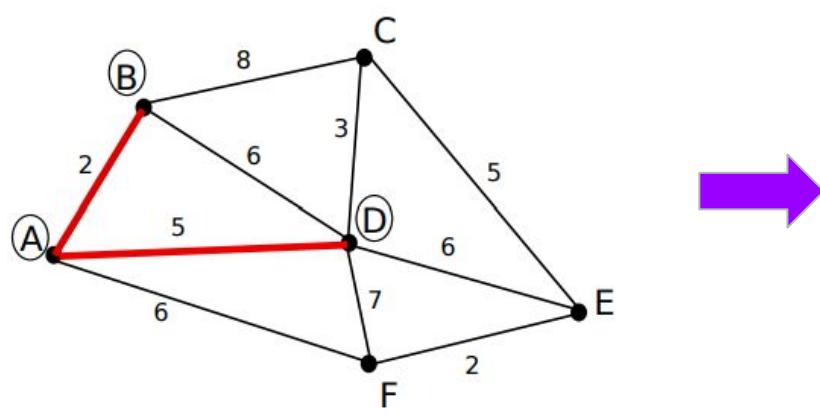
MST: Prim's algorithm

Look at all edges connected to A and B: (B,C), (B,D), (A,D), (A,F). Choose the one with minimum weight and add it to the tree: (A,D).



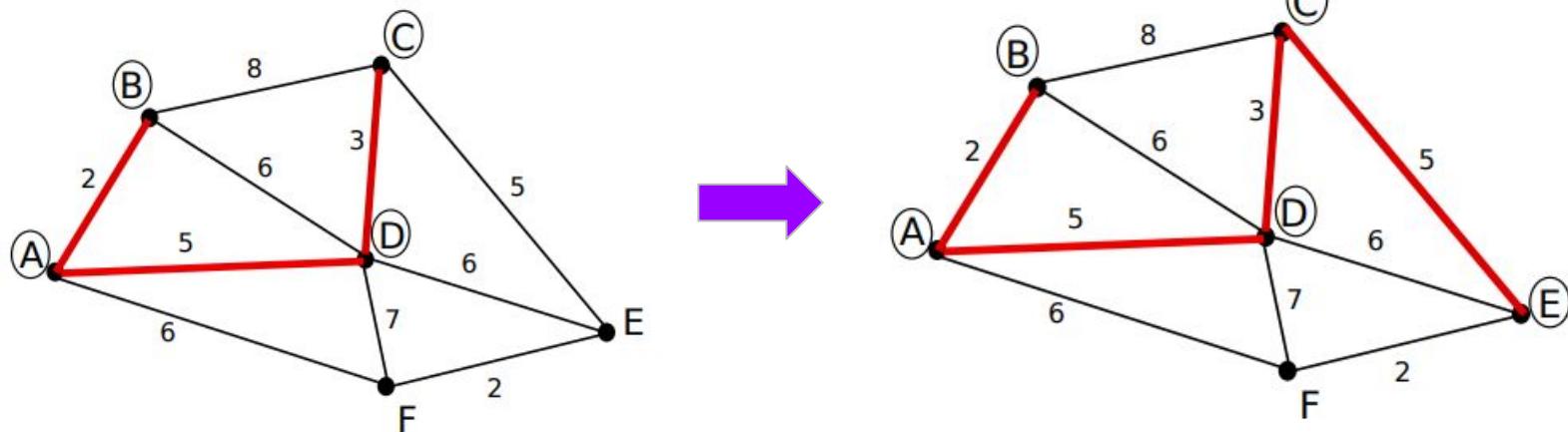
MST: Prim's algorithm

Look at all edges connected to the tree: (B,C), (D,C), (D,E), (D,F), (A,F). We don't need to consider edge (B,D) because both B and D are in the tree already.
We choose edge (D,C).



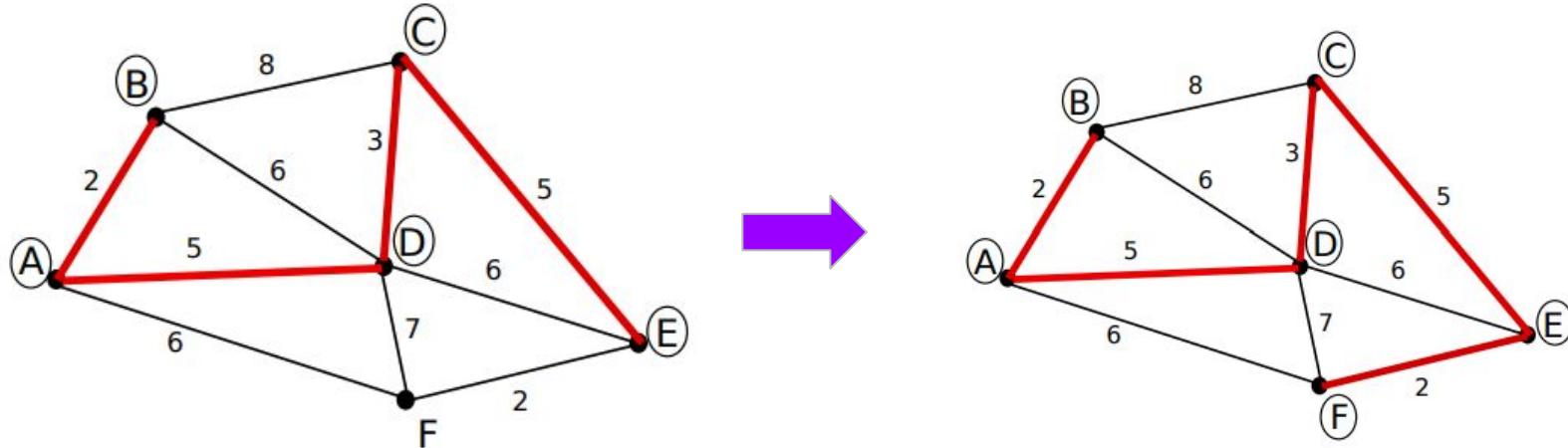
MST: Prim's algorithm

Look at all edges connected to A, B, C and D. We still have to connect E and F to the tree. So we look at the edges connected to those and choose the one with the **lowest weight: (C,E)**.



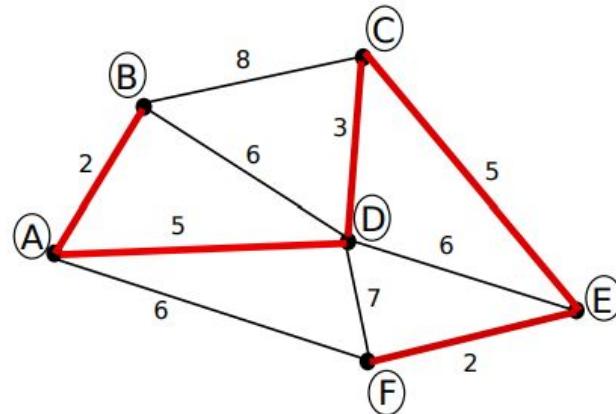
MST: Prim's algorithm

There is only one vertex F to add before we have a connected minimum spanning tree. **We choose edge (E,F)** and add that one to the tree.



MST: Prims algorithm

The tree is now connected and spans all vertices in the graph.

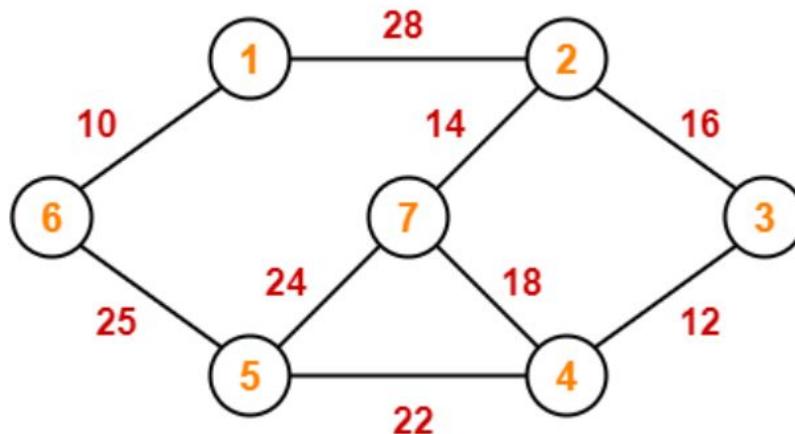


Kruskal Algorithm: main idea

- The **Kruskal's Algorithm** is based directly on the generic algorithm.
 - Unlike **Prim's algorithm**, we make a different choices of cuts.
 - Initially, trees of the forest are the vertices (no edges).
 - In each step add the cheapest edge that does not create a cycle.
- Observe that unlike **Prim's algorithm**, which only grows one tree, **Kruskal's algorithm** grows a collection of trees (a forest).
- Continue until the forest ‘merge to’ a single tree. This is a minimum spanning tree.

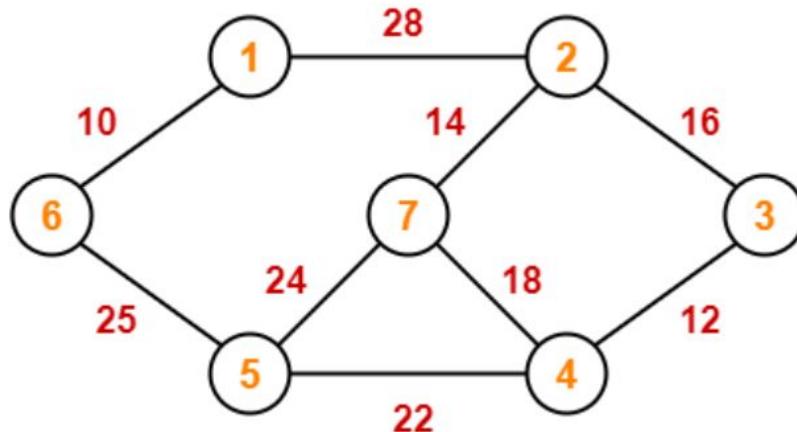
Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**



Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**



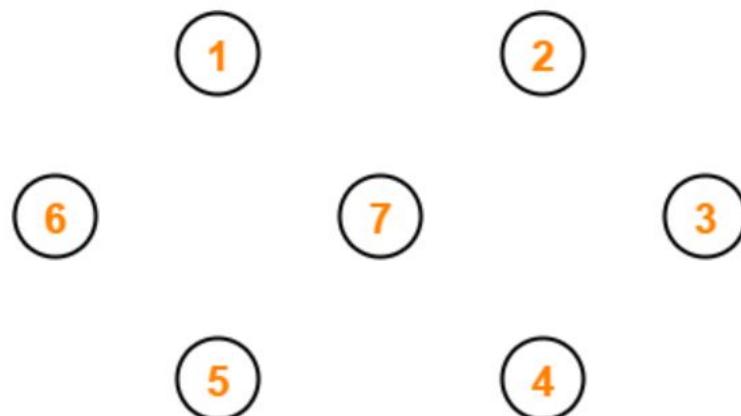
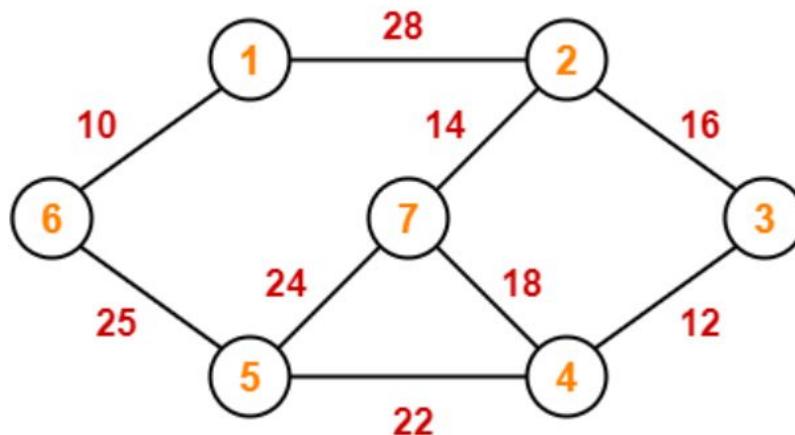
Solution:

- To construct MST using Kruskal's Algorithm,
 - Simply draw all the vertices on the paper.
 - Connect these vertices using edges with minimum weights such that no cycle gets formed.

Kruskal Algorithm: example

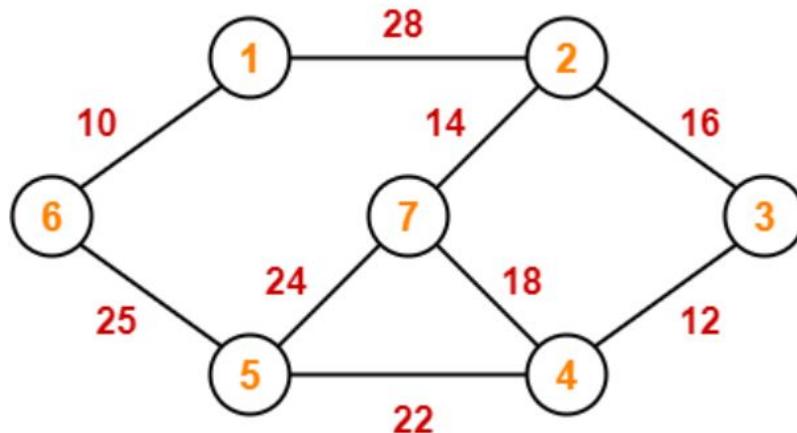
- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**

Step 1:



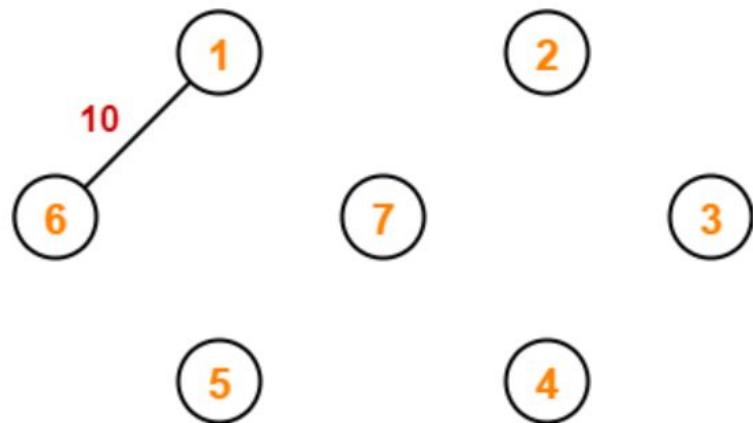
Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**



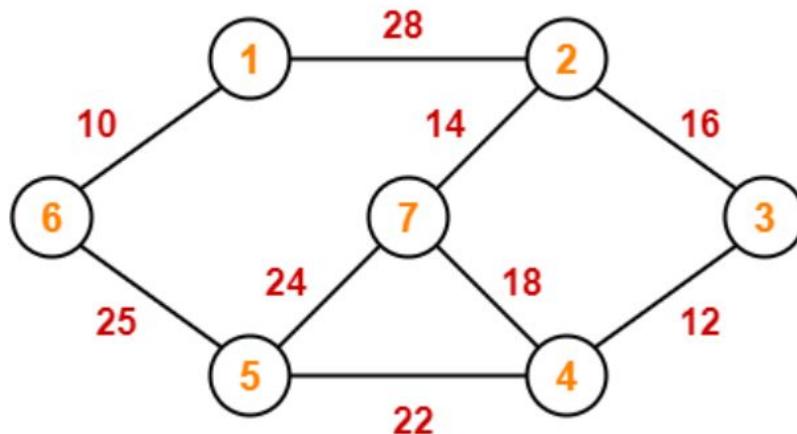
Step 2: Find the minimum cost edge in the graph and connect that in MST

Also, check for no cycle formed.



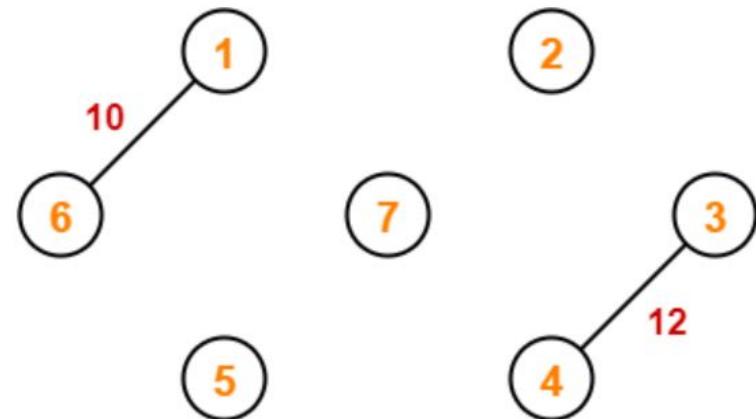
Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**



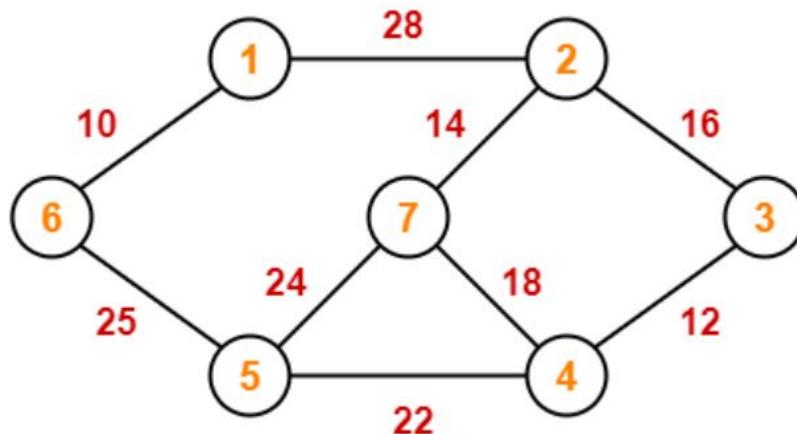
Step 2: Find the minimum cost edge in the graph and connect that in MST

Also, check for no cycle formed.



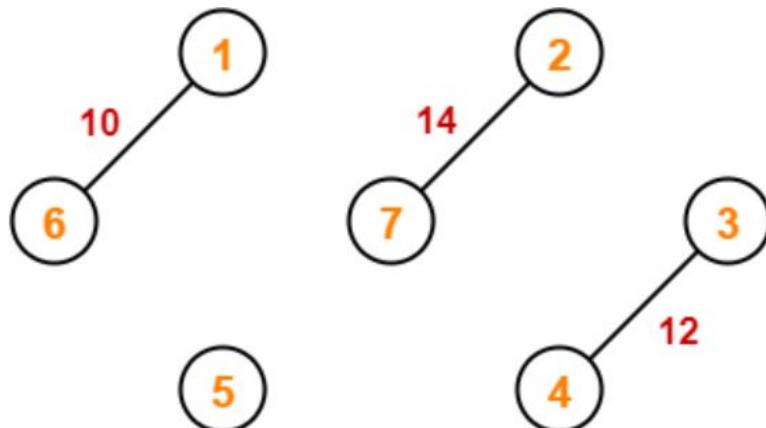
Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**



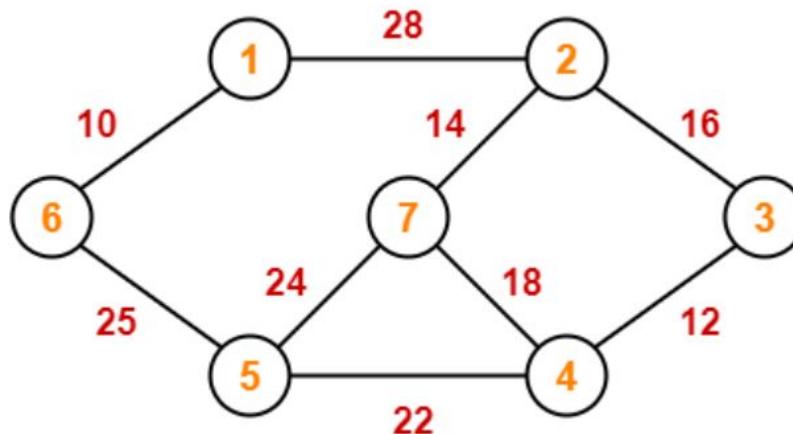
Step 2: Find the minimum cost edge in the graph and connect that in MST

Also, check for no cycle formed.



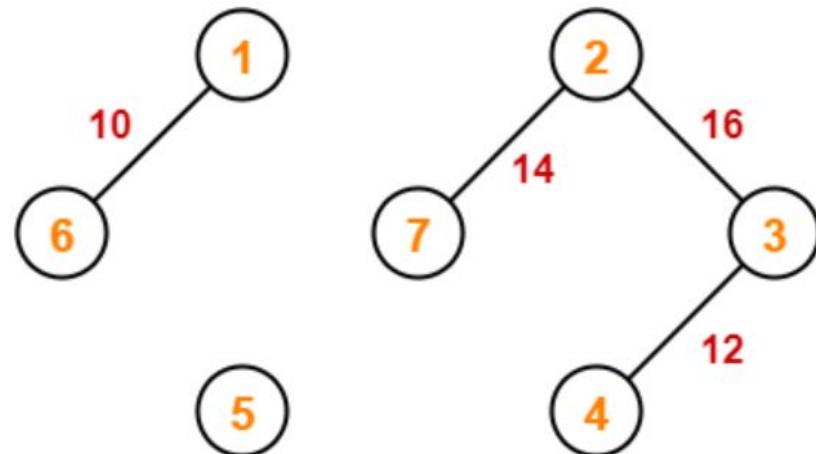
Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**



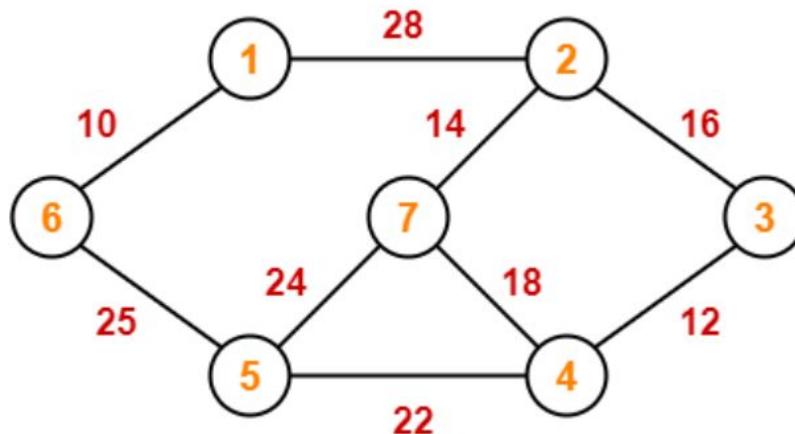
Step 2: Find the minimum cost edge in the graph and connect that in MST

Also, check for no cycle formed.



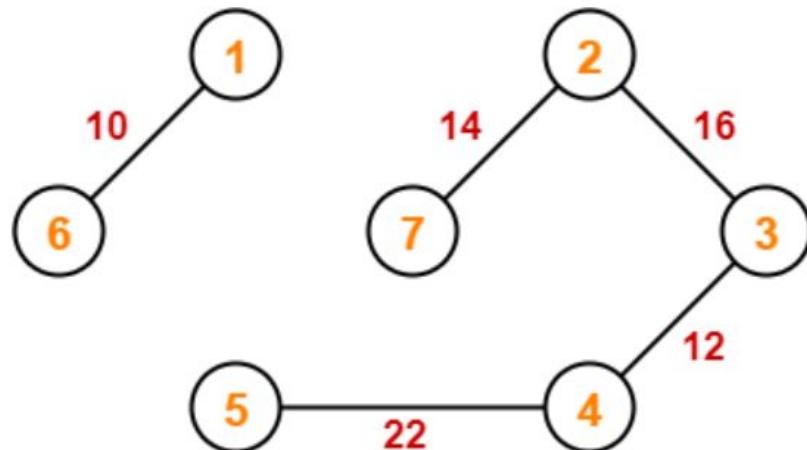
Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**



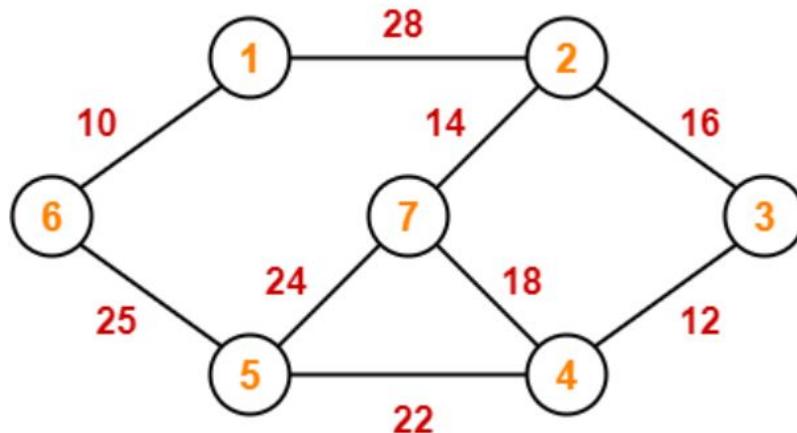
Step 2: Find the minimum cost edge in the graph and connect that in MST

Also, check for no cycle formed.



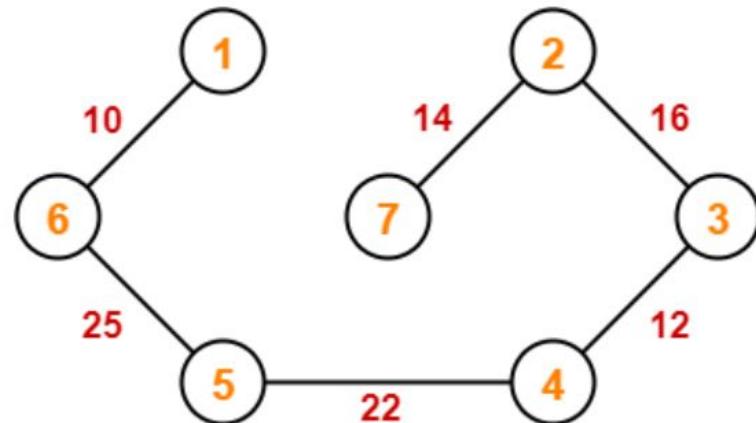
Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**



Step 2: Find the minimum cost edge in the graph and connect that in MST

Also, check for no cycle formed.

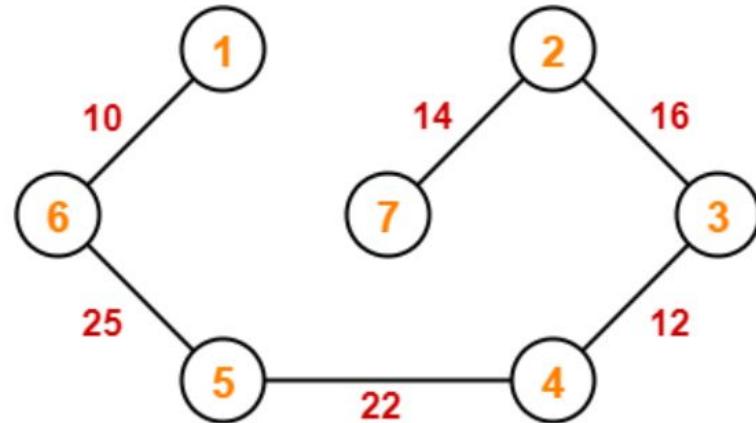
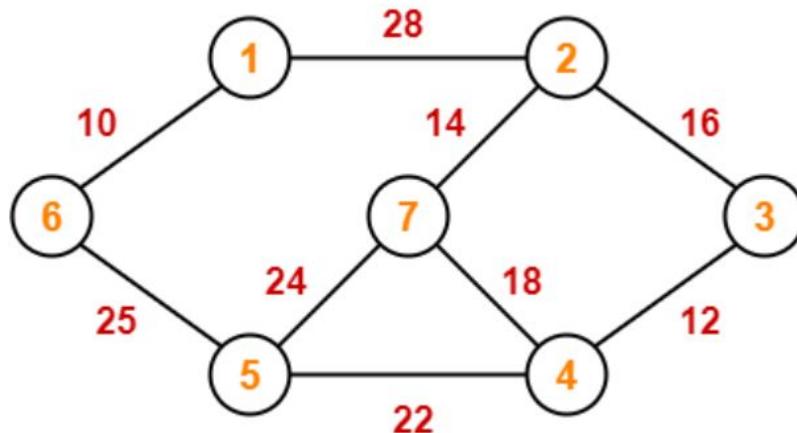


Kruskal Algorithm: example

- Construct the minimum spanning tree (MST) for the given graph using **Kruskal's Algorithm**

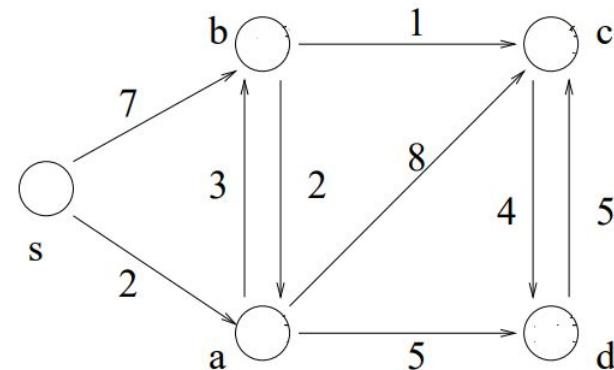
Since all the vertices have been connected / included in the MST, so we stop.

**Weight of the MST = Sum of all edge weights
= $10 + 25 + 22 + 12 + 16 + 14 = 99$ units**



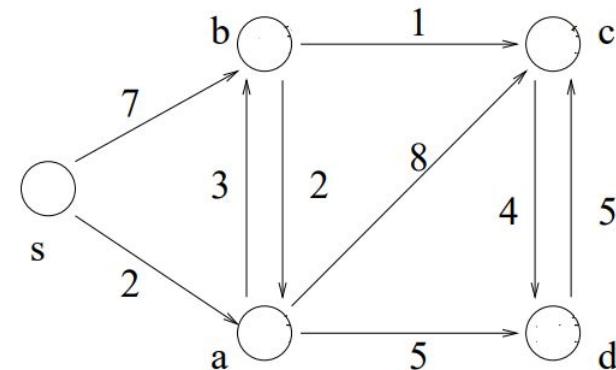
Dijkstra's Algorithm: shortest path problem

- In a weighted graph we need to find the **shortest path** between one vertex to all other vertices.
- **Consider the following graph:**
 - Suppose the starting vertex is S. Then we need to find the shortest path between S and all other vertices.
 - Finding the shortest path is a minimization problem. Also, a minimization problem is an optimization problem.
 - As we know, the optimization problem can be solved using **Greedy Method**.



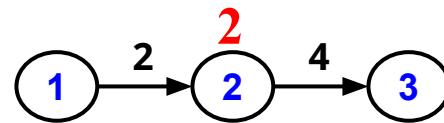
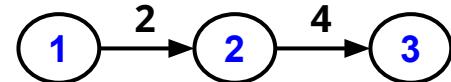
Dijkstra's Algorithm: shortest path problem

- Greedy Method says that to get an optimal solution any problem can be solved in stages:
 - By taking one stage at a time and
 - By considering one input at a time.
- Dijkstra algorithm follow the greedy approach to get an optimal result that is shortest path.
- Dijkstra algorithm work on both directed and undirected graph.
- Let's see the approach of Dijkstra's algorithm.

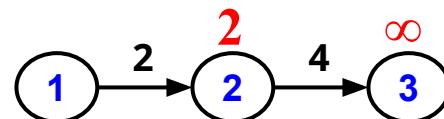


Dijkstra's Algorithm: approach

- Let's say **1**, is the starting vertex in the given graph. We need to find the shortest path to vertices **2** and **3**.
- As we see clearly, there is a path between **1** to **2** with weight **2** so, the shortest path between **1** and **2** is **2**.



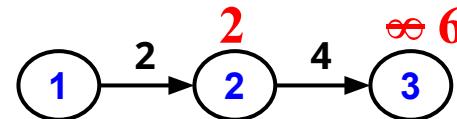
- And, as there is no direct path between **1** to **3** so, the shortest path between **1** and **3** is **∞** .



Dijkstra's Algorithm: approach contd..

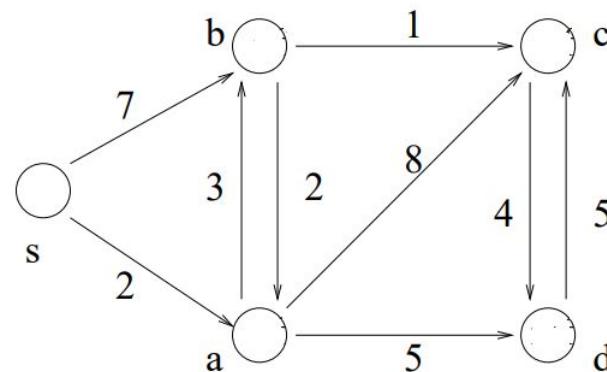
- Once we have selected any shortest path vertex, we check via that vertex is there any other shortest path to other vertices.
- Here, we selected the vertex **1** as shortest path vertex. So, now we check the shortest path between vertex **2** and only vertx of the graph 3.
- We find that distance between **2** and **3** is **4** and hence, we can reach vertex 3 from **1** via **2** in a shortest distance **6**.
- Therefore, the ∞ will be replaced/relaxed by **6** as shown below:

```
Relax(u,v)
{
    if ( $d[u] + w(u, v) < d[v]$ )
    {
         $d[v] = d[u] + w(u, v);$ 
         $pred[v] = u;$ 
    }
}
```



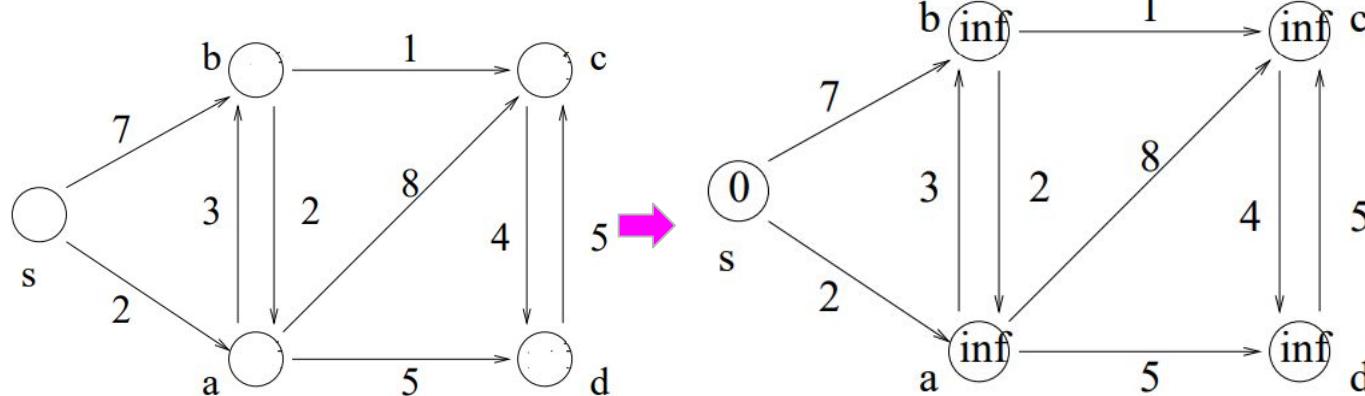
Dijkstra's Algorithm: Problem solving

- Consider the following graph and apply Dijkstra's algorithm to find the shortest path between the starting vertex **S** and other vertices in the graph.



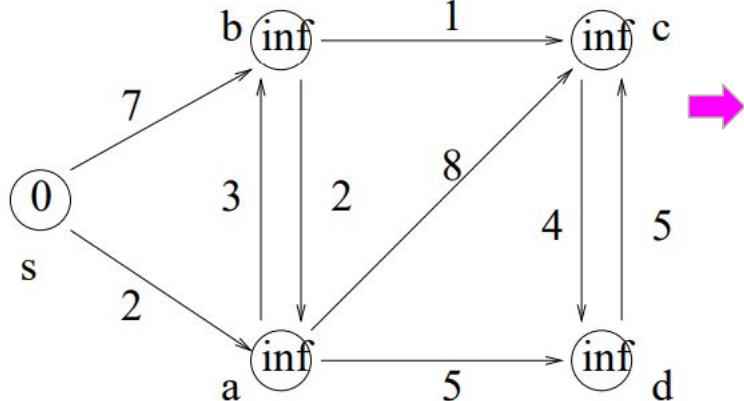
Dijkstra's Algorithm: Problem solving

- Consider the following graph and apply Dijkstra's algorithm to find the shortest path between the starting vertex S and other vertices in the graph.
- Step 1: Initialization:** Find the direct distance between S and all other vertices.
Distances are shown inside the node.
 - Initially, all distances are set to ∞ except the start node itself which is **0**.



Dijkstra's Algorithm: Problem solving

- Consider the following graph and apply Dijkstra's algorithm to find the shortest path between the starting vertex **S** and other vertices in the graph.
- Step 1: Initialization:** Find the direct distance between S and all other vertices.
Distances are shown inside the node.
 - Initially, all distances are set to ∞ except the to start node itself which is **0**.



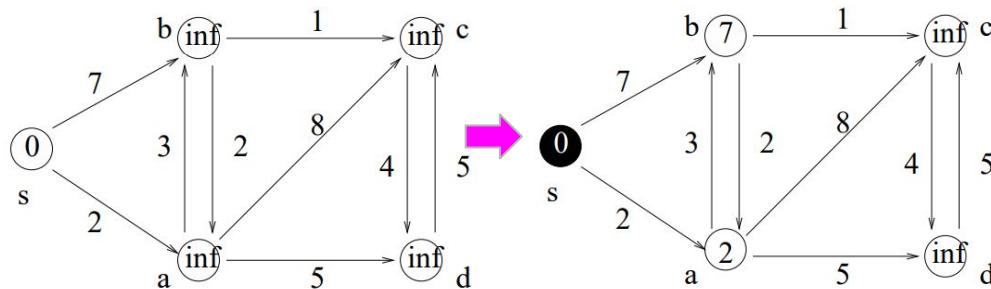
v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil

Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

Dijkstra's Algorithm: Problem solving

- Step 2: As $\text{Adj}[s] = \{a, b\}$ work on **a** and **b** and update information by relaxing weight, if possible.



Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

Priority Queue:

v	a	b	c	d
$d[v]$	2	7	∞	∞

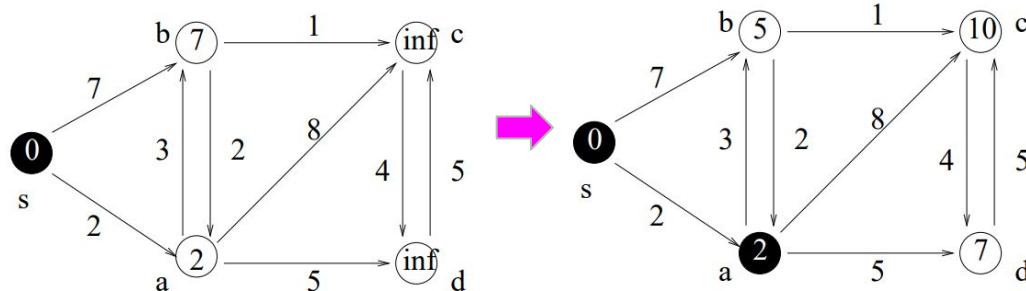
v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil

↓

v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil

Dijkstra's Algorithm: Problem solving

- Step 3: After Step 2, **a** has the minimum key in the priority queue. As $\text{Adj}[a] = \{b, c, d\}$ work on **b,c** and **d** and update information by relaxing weight, if possible.



Priority Queue:	v	a	b	c	d
	$d[v]$	2	7	∞	∞

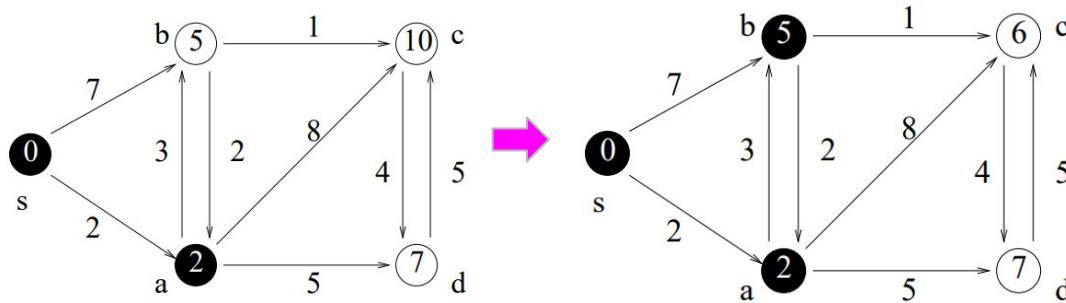
Priority Queue:	v	b	c	d
	$d[v]$	5	10	7

v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil

v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a

Dijkstra's Algorithm: Problem solving

- Step 4: After Step 3, **b** has the minimum key in the priority queue. As $\text{Adj}[b] = \{a, c\}$ work on **a** and **c** and update information by relaxing weight, if possible.



v	b	c	d
$d[v]$	5	10	7
$pred[v]$	s	a	a

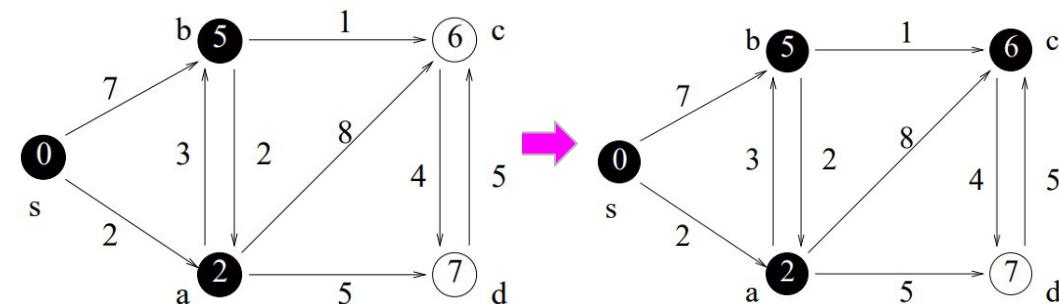
v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a

v	c	d
$d[v]$	6	7

Dijkstra's Algorithm: Problem solving

- Step 5: After Step 4, **c** has the minimum key in the priority queue. As $\text{Adj}[c] = \{d\}$ work on **d** update information by relaxing weight, if possible.



Priority Queue: $\begin{array}{c|cc} v & c & d \\ \hline d[v] & 6 & 7 \end{array}$

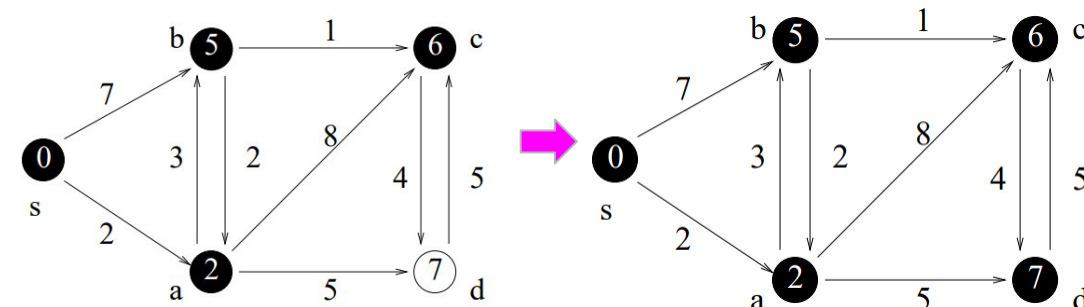
Priority Queue: $\begin{array}{c|c} v & d \\ \hline d[v] & 7 \end{array}$

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$\text{pred}[v]$	nil	s	a	b	a

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$\text{pred}[v]$	nil	s	a	b	a

Dijkstra's Algorithm: Problem solving

- Step 6: After Step 5, **d** has the minimum key in the priority queue. As $\text{Adj}[d] = \{c\}$ work on **c** and update information by relaxing weight, if possible.



Priority Queue: $\frac{v}{d[v]} \mid \frac{d}{7}$



Priority Queue: $Q = \emptyset$.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$\text{pred}[v]$	nil	s	a	b	a

A pink arrow points from the first table to this one.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$\text{pred}[v]$	nil	s	a	b	a

