

18CSC201J

DATA STRUCTURES AND ALGORITHMS

Session 1

Introduction- Basic Terminology

&

Data Structures

Prepared by

Dr. K.Venkatesh

Asst. Professor / Department of IT / SRMIST

Introduction:

- Data
 - Means?
 - Purpose?
- Computer & Data
 - Dependency Relationship
 - Utilization
 - Consequences

Basic Terminology

- Data / Datum
- Elementary Data
- Grouped Data
- Entity and Entity set
- Information
- Record
- File

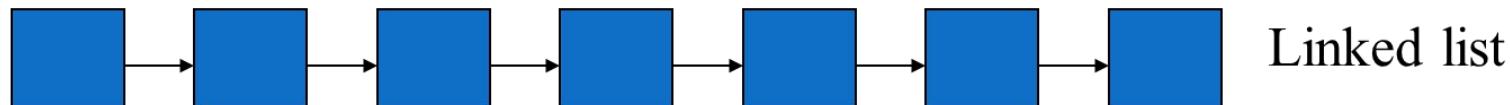
Data Structures

- Definition
 - The logical or mathematical model of organization of data
 - Data Structure reflects the relationships among the collection of data.
 - Data Structure allows processing of data efficiently.
 - Data structure reflects organization of data in the memory of a computer.

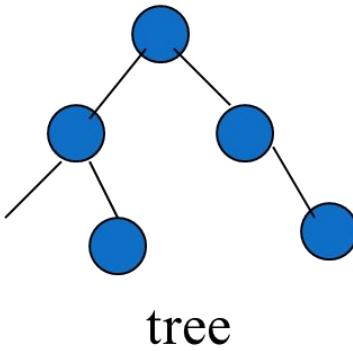
Examples:



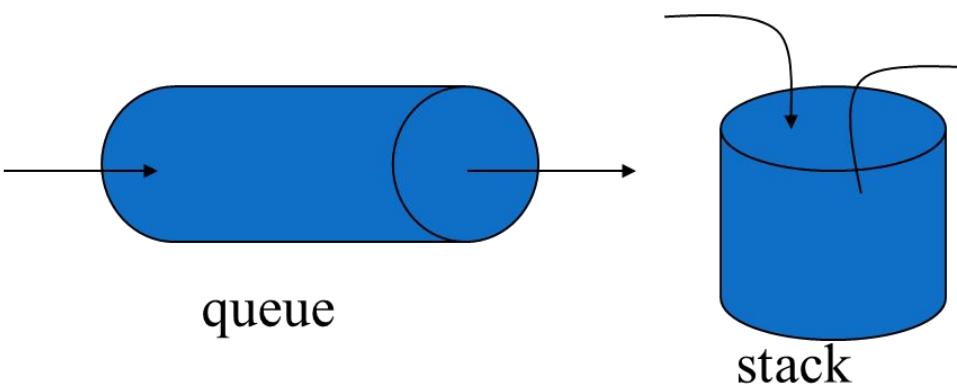
array



Linked list



tree



queue

stack

Types of Data Structure

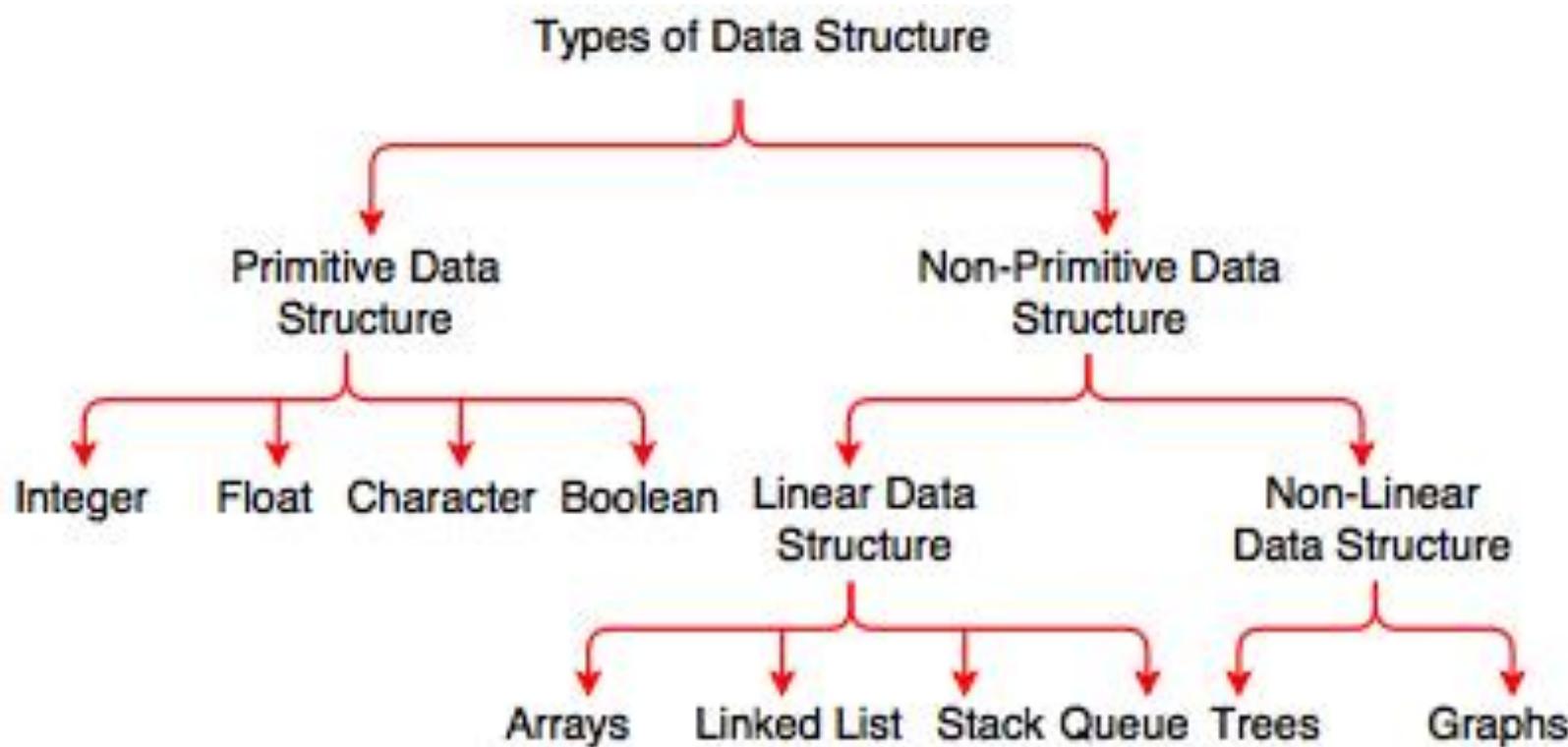


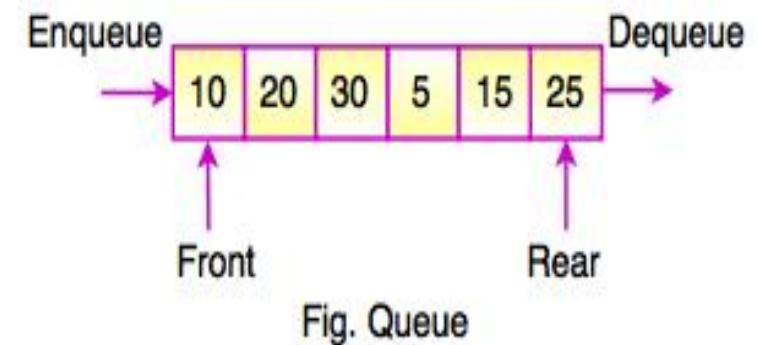
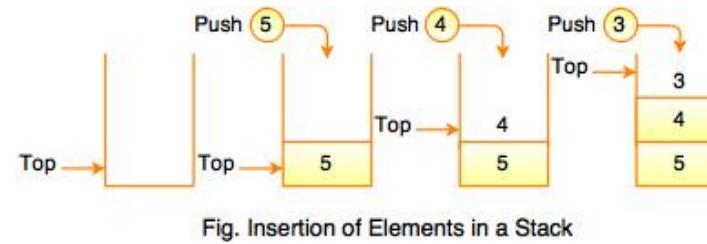
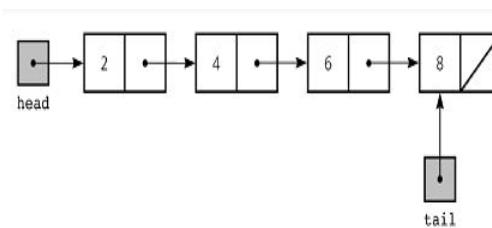
Fig. Types of Data Structure

Primitive Data Type

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

Non Primitive Data Type

- Classified into
 1. Linear Data Structure – Array, Linked List, Stack, Queue



2. Non-Linear Data Structure – Tree & Graph

Thank You

(Questions ????)

18CSC201J

DATA STRUCTURES AND ALGORITHMS

Session 2

Data Structure Operations & ADT

Prepared by

Dr. K.Venkatesh

Asst. Professor / Department of IT / SRMIST

ADT

- ADT means - **Abstract Data Type**
- Definition:
 - It's the logical representation of collection of data in a data structure along with set of operations that are permitted over the data items in the data structure.
 - Example
 - List (Creation, Insert, Delete, Search)
 - Stack (Push, Pop)
 - Queue (Enqueue, Dequeue)

Data Structure Operations

- Array ADT
 - Creation
 - Definition and Declaration
 - Insertion
 - Deletion
 - Sorting
- Searching
- Applications

Data Structure Operations

- Linked List
 - Definition and Declaration (Structure)
 - Creation (Head node)
 - Insertion
 - Deletion
 - Search / Traverse
 - Types (Single, Double, Multi, Circular)
 - Applications

Data Structure Operations

- Stack
 - Definition and Declaration (Array / List)
 - Insertion (Push)
 - Deletion (Pop)
 - Searching
 - Applications

Data Structure Operations

- Queue
 - Definition and Declaration (Array / List)
 - Insertion (Enqueue)
 - Deletion (Dequeue)
 - Types (Linear, Circular)
 - Applications

Data Structure Operations

- Tree
 - Definition and Declaration (Array / List)
 - Insertion
 - Deletion
 - Types (Binary Tree, Binary Search Tree, AVL Tree, RB Tree)
 - Applications

Thank You

(Questions ???)

Session 3

Linear Search and Binary Search

Introduction

What is search?

- The process of determining the value in a given list of values.
- it also finds the index of the searched value.

Types

- Linear Search
- Binary Search

Linear Search

uses brute force technique to find a given item (element).

The time complexity of Linear search is $O(n)$

n is total number of items in the list

Linear Search Algorithm:

- **Step 1** - Take both input, set of numbers and search element from the user.
- **Step 2** - Match the search element with the starting element in the list.
- **Step 3** - If both the elements are matched, then display "Given search item is found" and stop the function
- **Step 4** - If both the elements are not matched, then compare the search element with the next element in the array list.
- **Step 5** - Repeat steps 3 and 4 until search element is compared with last element in the array list.
- **Step 6** - If the end element in the list also doesn't match, then display "Element is not found" and stop the function.

Example

consider the following list

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

Search element is 30

- Step 1:

List:

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

The first element and match element are not same. So it moves with next element of the array.

- Step 2:

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

- Step 3:

List

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

Step 4:

List

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

- Step 5:

List

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

- Matched- the procedure has been stopped and the function displays element found
- return the position of the element is 5.
(If array index starts by 1 otherwise the same will return 4.)

Binary Search

Efficient than Linear search

Prerequisite: Sorted list in ascending order

The time complexity of Linear search is **O(logn)**

n is total number of items in the list

Binary Search Algorithm:

- **Step 1** - Take both the inputs, set of numbers and the search element from the user.
- **Step 2** - Find the middle element in the array of elements (Constraint : list should be sorted list).
- **Step 3** - Compare the search item with the middle element in the array list.
- **Step 4** - If both the elements are equal, then show that, "The Given element is available in the list" and stop the function.
- **Step 5** - If both the elements are not matched, then check whether the search element is lesser or greater than the middle element.

Binary Search Algorithm:

- **Step 6** - If the search element is lesser than the middle element, repeat the steps 2, 3, 4 and 5 for the left partition of sub array of the middle element.
- **Step 7** - If the search element is greater than middle element, repeat the steps 2, 3, 4 and 5 for the right partition of sub array of the middle element.
- **Step 8** - Repeat the same procedure until we find the search element in the list or until sub array contains only one element.
- **Step 9** - If that element also doesn't match with the search element, then display "The Element is not found in the list" and stop the function.

Example

consider the following list

0 1 2 3 4 5 6 7 8

10	12	15	20	30	42	50	58	70
----	----	----	----	----	----	----	----	----

Search element is 20

Example

Step 1:

Find Middle Element = (Start index+ End index)/2
= (0+8)/2 =4 The middle element is (30)

0 1 2 3 4 5 6 7 8

10	12	15	20	30	42	50	58	70
----	----	----	----	----	----	----	----	----

Search element is 20

Example

Step 2:

The search element is compared with middle element and it is smaller than middle element. Searching starts with left sub array. (10,12, 15, 20)

Step 3:

The search element is compared with middle element (12). And it is greater than middle element. search technique starts with right sub array of the middle element. (15,20)

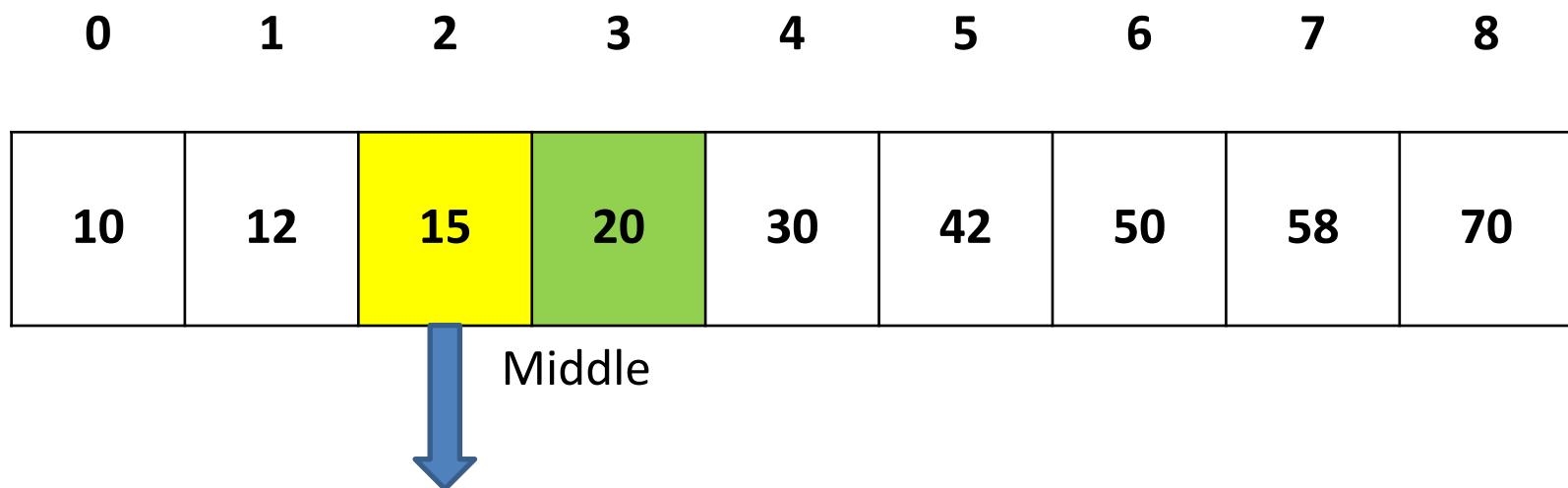
0 1 2 3 4 5 6 7 8

10	12	15	20	30	42	50	58	70
----	----	----	----	----	----	----	----	----

Example

Step 4:

- The search element is compared with middle element (15). And it is greater than middle element. Now the search technique starts with right sub array of the middle element. (20).



Thank you

Session -4 & 5

Linear Search and Binary Search

Meenakshi K

Assistant Professor.

Department of Information Technology
SRM Institute of Science and Technology

Ms. Meenakshi K

Assistant Professor

E - mail: meenaksk@srmist.edu.in

Area: Artificial Intelligence and Machine Learning

Specialization: Machine Learning

Affiliation: Department of Information Technology,

Kattankulathur Campus,

SRM Institute of Science and Technology

Introduction

What is search?

- The process of determining the value in a given list of values.
- it also finds the index of the searched value.

Types

- Linear Search
- Binary Search

Linear Search

uses brute force technique to find a given item (element).

The time complexity of Linear search is $O(n)$

n is total number of items in the list

Linear Search Algorithm:

- **Step 1** - Take both input, set of numbers and search element from the user.
- **Step 2** - Match the search element with the starting element in the list.
- **Step 3** - If both the elements are matched, then display "Given search item is found" and stop the function
- **Step 4** - If both the elements are not matched, then compare the search element with the next element in the array list.
- **Step 5** - Repeat steps 3 and 4 until search element is compared with last element in the array list.
- **Step 6** - If the end element in the list also doesn't match, then display "Element is not found" and stop the function.

Example

consider the following list

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

Search element is 30

- Step 1:

List:

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

The first element and match element are not same. So it moves with next element of the array.

- Step 2:

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

- Step 3:

List

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

Step 4:

List

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

- Step 5:

List

60	20	15	52	30	12	10	28	1
----	----	----	----	----	----	----	----	---

- Matched- the procedure has been stopped and the function displays element found
- return the position of the element is 5.
(If array index starts by 1 otherwise the same will return 4.)

Binary Search

Efficient than Linear search

Prerequisite: Sorted list in ascending order

The time complexity of Linear search is **O(logn)**

n is total number of items in the list

Binary Search Algorithm:

- **Step 1** - Take both the inputs, set of numbers and the search element from the user.
- **Step 2** - Find the middle element in the array of elements (Constraint : list should be sorted list).
- **Step 3** - Compare the search item with the middle element in the array list.
- **Step 4** - If both the elements are equal, then show that, "The Given element is available in the list" and stop the function.
- **Step 5** - If both the elements are not matched, then check whether the search element is lesser or greater than the middle element.

Binary Search Algorithm:

- **Step 6** - If the search element is lesser than the middle element, repeat the steps 2, 3, 4 and 5 for the left partition of sub array of the middle element.
- **Step 7** - If the search element is greater than middle element, repeat the steps 2, 3, 4 and 5 for the right partition of sub array of the middle element.
- **Step 8** - Repeat the same procedure until we find the search element in the list or until sub array contains only one element.
- **Step 9** - If that element also doesn't match with the search element, then display "The Element is not found in the list" and stop the function.

Example

consider the following list

0 1 2 3 4 5 6 7 8

10	12	15	20	30	42	50	58	70
----	----	----	----	----	----	----	----	----

Search element is 20

Example

Step 1:

Find Middle Element = (Start index+ End index)/2
= (0+8)/2 =4 The middle element is (30)

0 1 2 3 4 5 6 7 8

10	12	15	20	30	42	50	58	70
----	----	----	----	----	----	----	----	----

Search element is 20

Example

Step 2:

The search element is compared with middle element and it is smaller than middle element. Searching starts with left sub array. (10,12, 15, 20)

Step 3:

The search element is compared with middle element (12). And it is greater than middle element. search technique starts with right sub array of the middle element. (15,20)

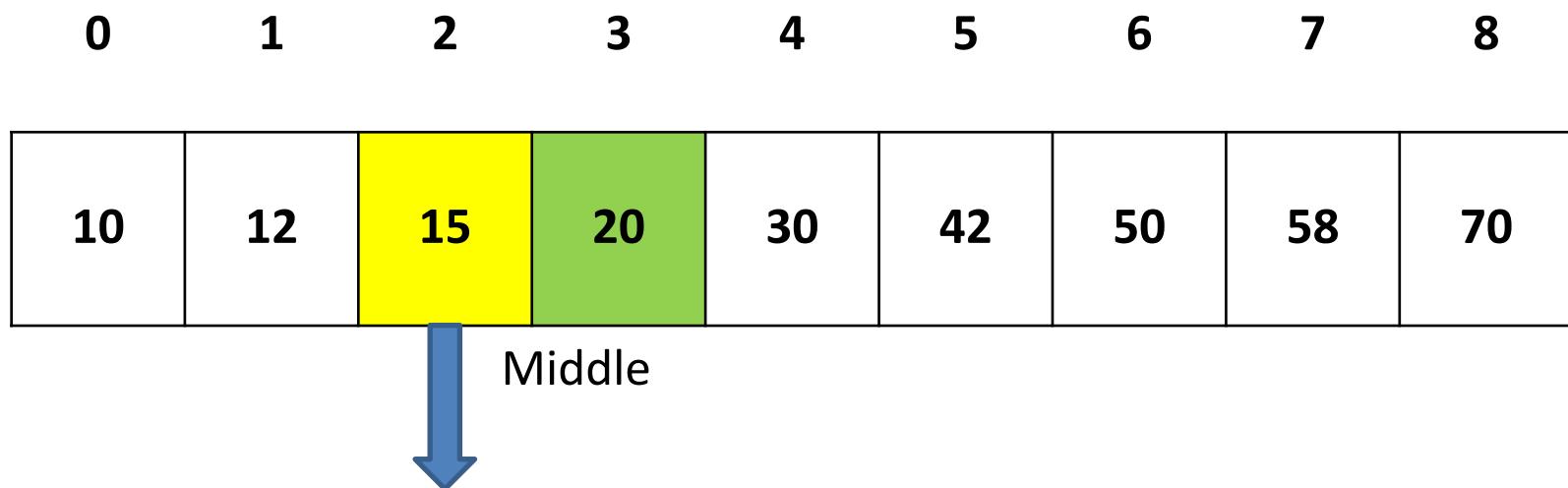
0 1 2 3 4 5 6 7 8

10	12	15	20	30	42	50	58	70
----	----	----	----	----	----	----	----	----

Example

Step 4:

- The search element is compared with middle element (15). And it is greater than middle element. Now the search technique starts with right sub array of the middle element. (20).



Thank you

Sorting Techniques

Sorting

- A **sorting algorithm** is an algorithm used to generate the ordered list in a certain order.
- Sorting is defined as ordering of given data based on different sorting technique



BASIC TYPES :

- Internal Sorting
- External Sorting

Sorting Techniques

- ✓ Bubble sort
- ✓ Insertion sort
- ✓ Select sort
- ✓ Quick sort
- ✓ Merge sort
- ✓ Bucket sort
- ✓ Radix sort

Bubble Sort

- ✓ **Bubble sort**, is referred as sinking sort it is the simple sorting algorithm.
- ✓ Swapping with adjustment element in the list. Until no further swapping is needed which denoted all the elements as sorted in the list
- ✓ The sorting order is small element to largest number like the bubble in the given image



Algorithm

```
procedure bubbleSort( A : list of sortable items ) defined
as: do
    swapped := false
    for each i in 0 to length( A ) - 1
        do: if A[ i ] > A[ i + 1 ] then
            swap( A[ i ], A[ i + 1 ] )
            ) swapped := true
        end if
    end for
    while swapped
end procedure
```

Example

Let us take the array of numbers "5 1 4 2 8"

First Pass:

(5 1 4 2 8) -> (1 5 4 2 8),

//Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 5 4 2 8) -> (1 4 5 2 8), Swap since $5 > 4$

(1 4 5 2 8) -> (1 4 2 5 8), Swap since $5 > 2$

(1 4 2 5 8) -> (1 4 2 5 8),

//Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Example

Second Pass:

(1 4 2 5 8) -> (1 4 2 5 8)

(1 4 2 5 8) -> (1 2 4 5 8), Swap since 4 > 2

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed.
The algorithm needs one whole pass without any swap to know it is sorted.

Example

Third Pass:

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

Time Complexities

Cases	Big - O
Best	$O(n)$
Average	$O(n^2)$
Worst	$O(n^2)$

Insertion Sort

All the data items arranged in one at a time
using Insertion sort algorithm



Algorithm

```
insertionSort(array A)
for i = 1 to length[A]-1 do
begin
    value =
    A[i] j = i-1
    while j >= 0 and A[j] > value do
    begin
        swap( A[j + 1], A[j]
            ) j = j-1
    end
    A[j+1] = value
end
```

Example

Consider the sequence, {3, 7, 4, 9, 5, 2, 6, 1}

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 5 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 6 7 9 1

1 2 3 4 5 6 7 9

Time Complexities

Cases	Big - O
Best	$O(n)$
Average	$O(n^2)$
Worst	$O(n^2)$

Selection Sort

Selection sort worked based comparison between the element

Algorithm

```
for i = 1:n,
```

```
    k = i
```

```
    for j = i+1:n, if a[j] < a[k], k = j
```

```
    → invariant: a[k] smallest of a[i..n]
```

```
    swap a[i,k]
```

```
    → invariant: a[1..i] in final position
```

```
end
```

Example



Time Complexities

Cases	Big - O
Best	$O(n^2)$
Average	$O(n^2)$
Worst	$O(n^2)$

Quick Sort

- Element is picked is known as pivot element . partitions the given array based on pivot element

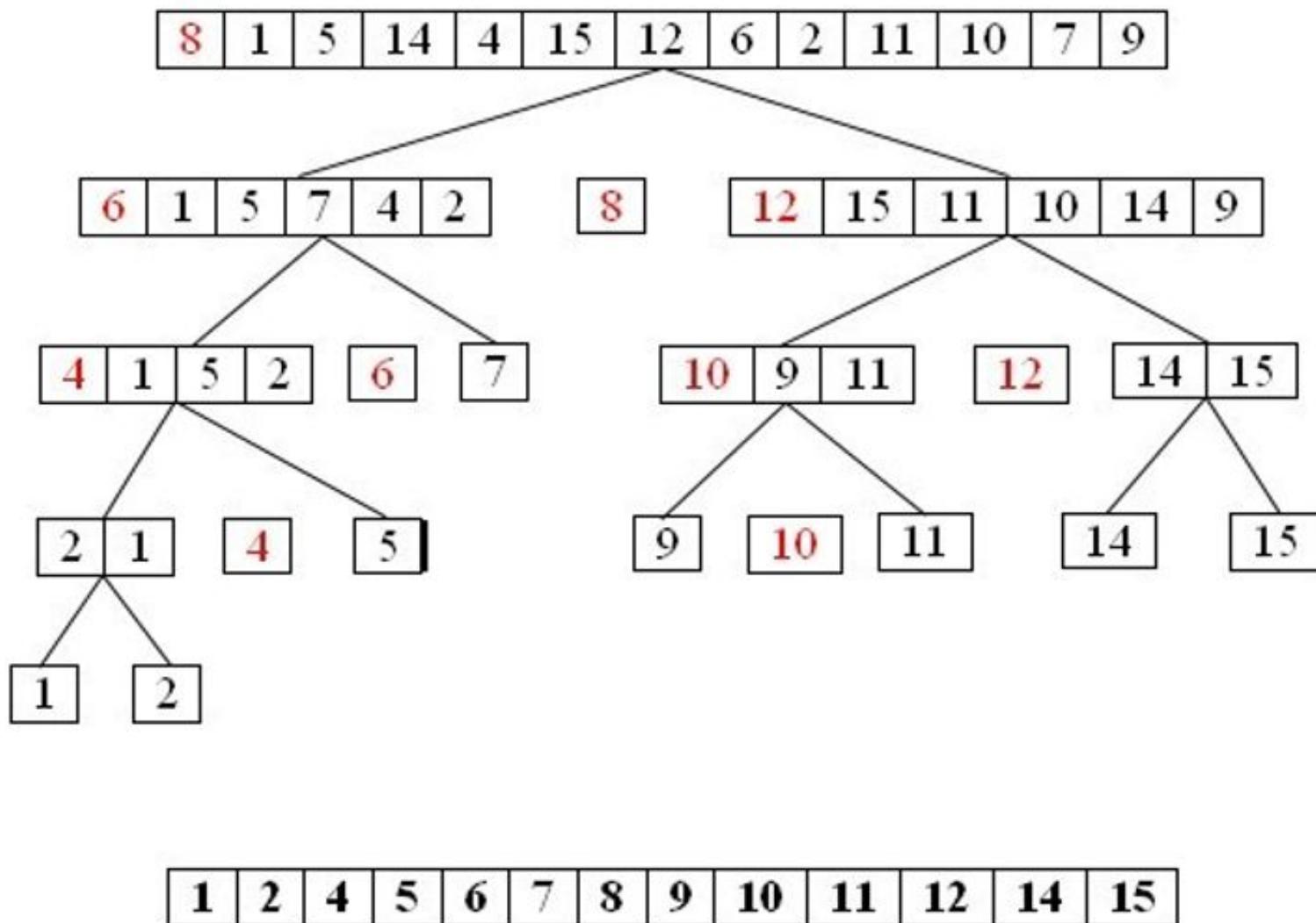
Algorithm

```
function partition(array, left, right, pivotIndex)
    pivotValue := array[pivotIndex]
    swap array[pivotIndex] and array[right] // Move pivot to
    end  storeIndex := left
    for i from left to right ? 1
        if array[i] ? pivotValue
            swap array[i] and array[storeIndex]
            storeIndex := storeIndex + 1
    swap array[storeIndex] and array[right] // Move pivot to its final
    place  return storeIndex
```

Algorithm

```
procedure quicksort(array, left, right)
    if right > left
        select a pivot index (e.g. pivotIndex := left)
        pivotNewIndex := partition(array, left, right,
        pivotIndex) quicksort(array, left, pivotNewIndex - 1)
        quicksort(array, pivotNewIndex + 1, right)
```

Example



Time Complexities

Cases	Big - O
Best	$O(n \log n)$
Average	$O(n \log n)$
Worst	$O(n^2)$

Merge Sort

“MergeSort is based on Divide and Conquer algorithm. It divides given input array in to two halves, calls itself for the two halves and then merges the two sorted halves.”

Algorithm

```
function mergesort(m)
    var list left, right, result
    if length(m) ? 1
        return m
    var middle = length(m) / 2
    for each x in m up to middle
        add x to left
    for each x in m after middle
        add x to right
    left = mergesort(left)
    right = mergesort(right)
    result = merge(left, right)
    return result
```

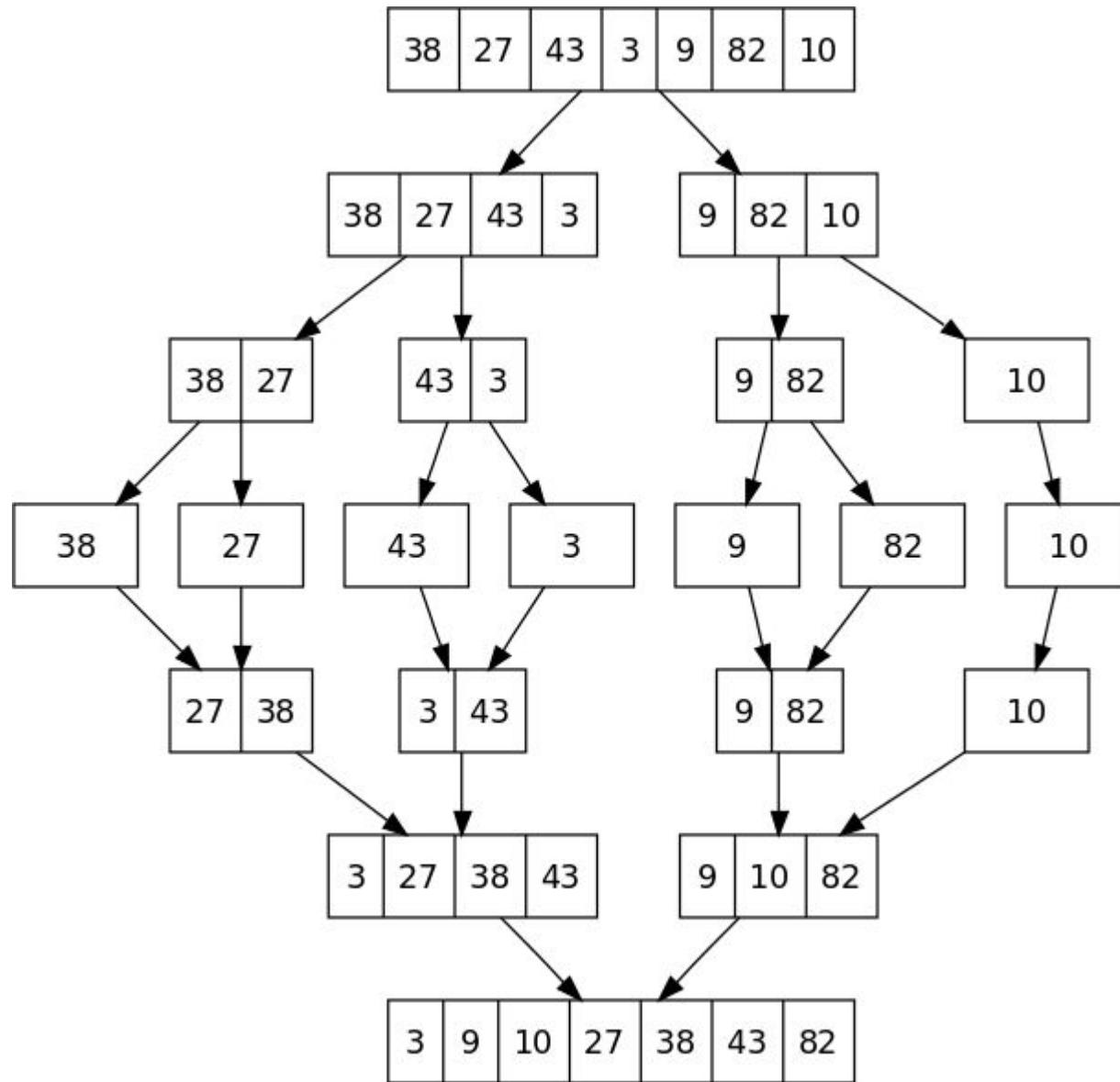
Algorithm

```
function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ? first(right)
            append first(left) to result
            left = rest(left)

        else
            append first(right) to result
            right = rest(right)

    end while
    if length(left) > 0
        append rest(left) to result
    if length(right) > 0
        append rest(right) to result
    return result
```

Example

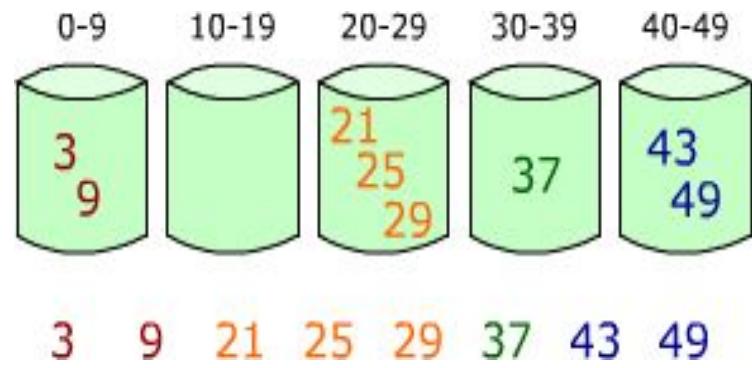
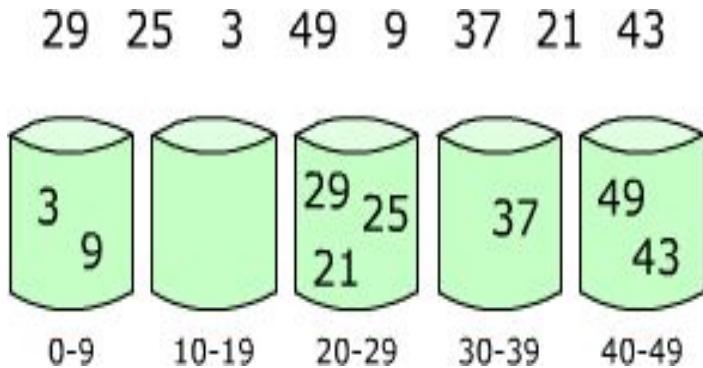


Time Complexities

Cases	Big - O
Best	$O(n \log n)$
Average	$O(n \log n)$
Worst	$O(n \log n)$

Bucket Sort

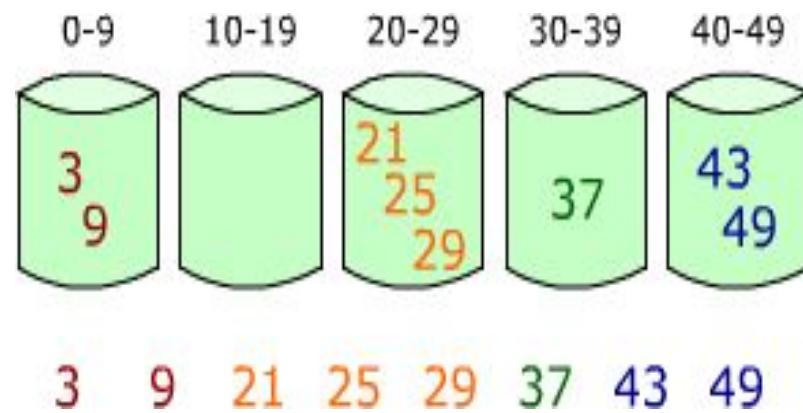
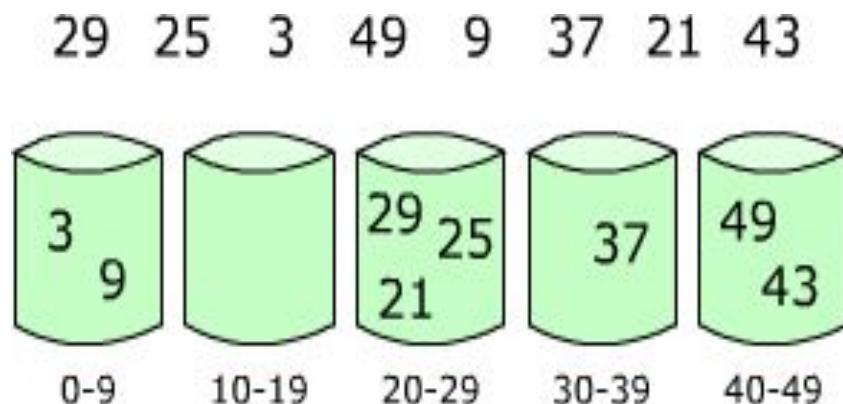
“Bucket sort otherwise called as bin sort, it is partitioning an given array into a number of buckets”



Algorithm: Pseudocode

```
function bucketSort(array, n) is
    buckets  $\leftarrow$  new array of n empty lists
    for i = 0 to (length(array)-1) do
        insert array[i] into buckets[msbits(array[i], k)]
    for i = 0 to n - 1 do
        nextSort(buckets[i]);
    return the concatenation of buckets[0], ..., buckets[n-1]
```

Example



Time Complexities

Cases	Big - O
Average	$O(n + k)$
Worst	$O(n^2)$

Radix Sort

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

Algorithm

RADIX SORT (A,
d)

```
for i ← 1 to d do
    use a stable sort to sort A on digit i
    // counting sort will do the job
```

Example

Following example shows how Radix sort operates on seven 3-digits number.

Input	1 st pass	2 nd pass	3 rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Time Complexities

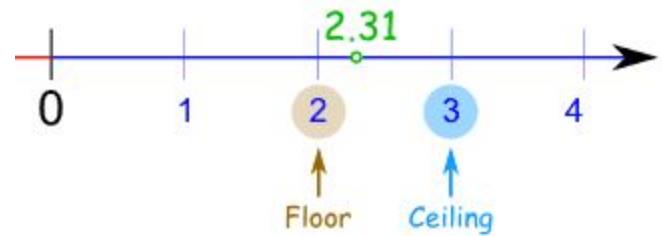
Cases	Big - O
Worst case time complexity	$O(n*k)$
Worst Case space complexity	$O(n + k)$

SESSION- 7 & 8

**MATHEMATICAL NOTATIONS, ASYMPTOTIC
NOTATION - BIG O , OMEGA, THETA
MATHEMATICAL FUNCTIONS**

1. FLOOR AND CEILING FUNCTIONS:

Floor Function: the greatest integer that is less than or equal to x



Ceiling Function: the least integer that is greater than or equal to x

Notation

$$\lfloor x \rfloor$$

$\text{floor}(x)$

$$\lceil x \rceil$$

$\text{ceil}(x)$

2. Remainder function : Modular arithmetic

Let x be any integer and let M be a positive integer.

Then $x \pmod{M}$

Eg:

$$13 \pmod{5} = 3$$

3. Integer and Absolute Value functions

$\text{INT}(x)$ converts x into an integer by deleting the fractional part of the number

$\text{ABS}(x)$ or $|x|$ gives the greater of x or $-x$

Eg:

$$\text{INT}(3.14) = 3$$

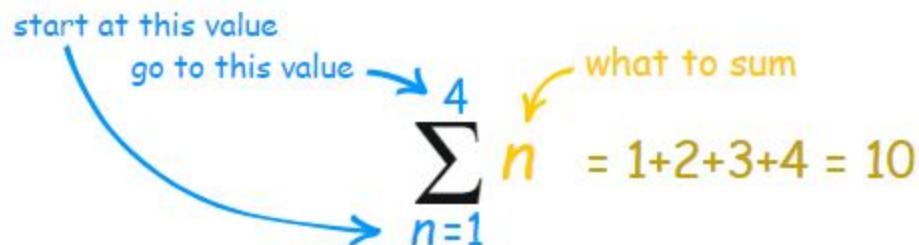
$$|15| = 15$$

$$|-0.33| = 0.33$$

4. Summation Symbol: sums

Summation symbol : Σ

This symbol (called **Sigma**) means "sum up"


$$\sum_{n=1}^4 n = 1+2+3+4 = 10$$

5. Factorial Function

The **factorial function** (symbol: !) says to **multiply all whole numbers** from our chosen number down to 1.

$$n! = 1 \cdot 2 \cdot 3 \dots (n-2) \cdot (n-1)$$

E.g: $2! = 1 \cdot 2 = 2$

6. Permutations

A permutation of a set of n elements is an arrangement of the elements in a given order .

E.g: Permutations of the elements a, b and c are
abc, acb, bac, bca, cab, cba

7. Exponents and Logarithms

Exponents are also called Powers or Indices.

Example: $5^3 = 5 \times 5 \times 5 = 125$

Logarithm of any positive number x to the base b , written as

$$\begin{aligned} &\log_b(x) \\ \Rightarrow &y = \log_b(x) \\ \Rightarrow &b^y = x \end{aligned}$$

Asymptotic Notation– Big, Omega , Theta

Asymptotic Notations refers to computing the running time of any operation in mathematical units of computation.

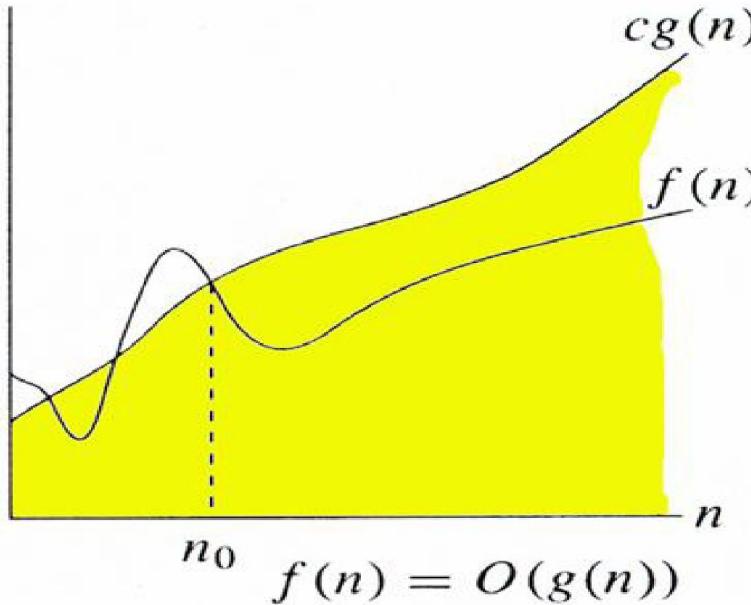
- **O Notation (Big)**
- **Ω Notation (Omega)**
- **Θ Notation (Theta)**

•Big Oh Notation, O

- It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

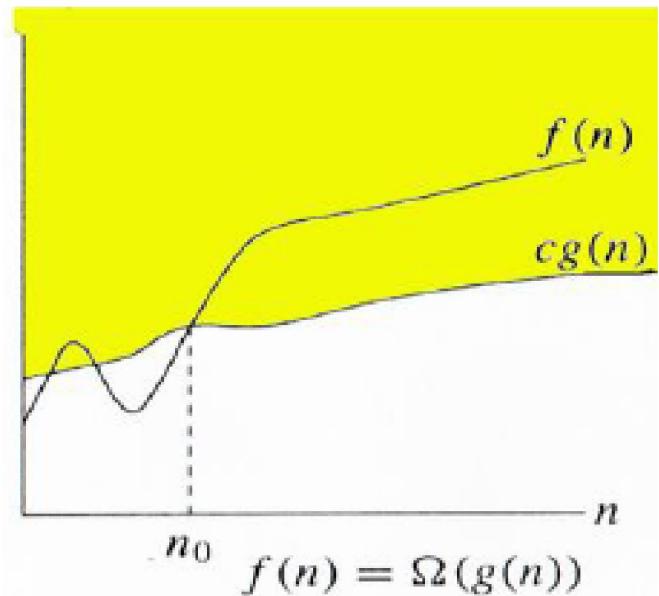


• Ω Notation:

- Ω notation provides an asymptotic lower bound

$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$

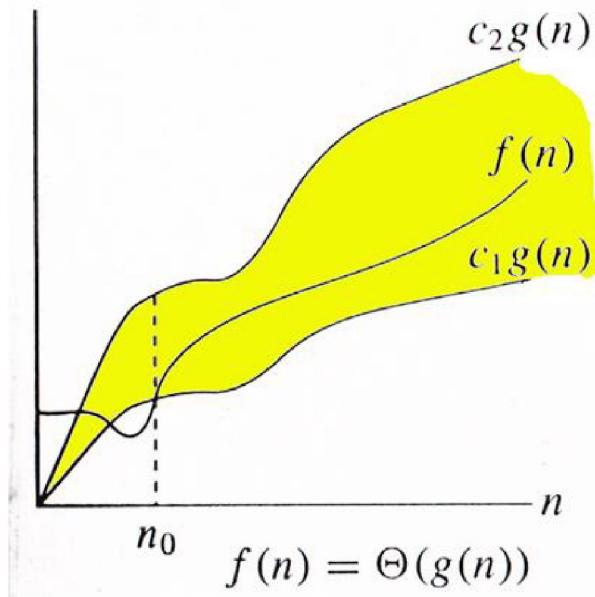
$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$



• Θ Notation:

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that}$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$



Mathematical Analysis

• For Non-recursive Algorithms

- There are four rules to count the operations:
 - **Rule 1: for loops - the size of the loop times the running time of the body**
 - Find the running time of statements when executed only once
 - Find how many times each statement is executed
 - **Rule 2 : Nested loops**
 - The product of the size of the loops times the running time of the body
 - **Rule 3: Consecutive program fragments**
 - The total running time is the maximum of the running time of the individual fragments
 - **Rule 4: If statement**
 - The running time is the maximum of the running times of if stmt and else stmt.

Exercise

a.

```
sum = 0;  
for( i = 0; i < n; i++)  
    for( j = 0; j < n * n; j++)  
        sum++;
```

Ans : $O(n^3)$

b.

```
sum = 0;  
for( i = 0; i < n; i++)  
    for( j = 0; j < i; j++)  
        sum++;
```

Ans : $O(n^2)$

c.

```
sum = 0;  
for( i = 0; i < n; i++)  
    for( j = 0; j < i*i; j++)  
        for( k = 0; k < j; k++)  
            sum++;
```

Ans : $O(n^5)$

d.

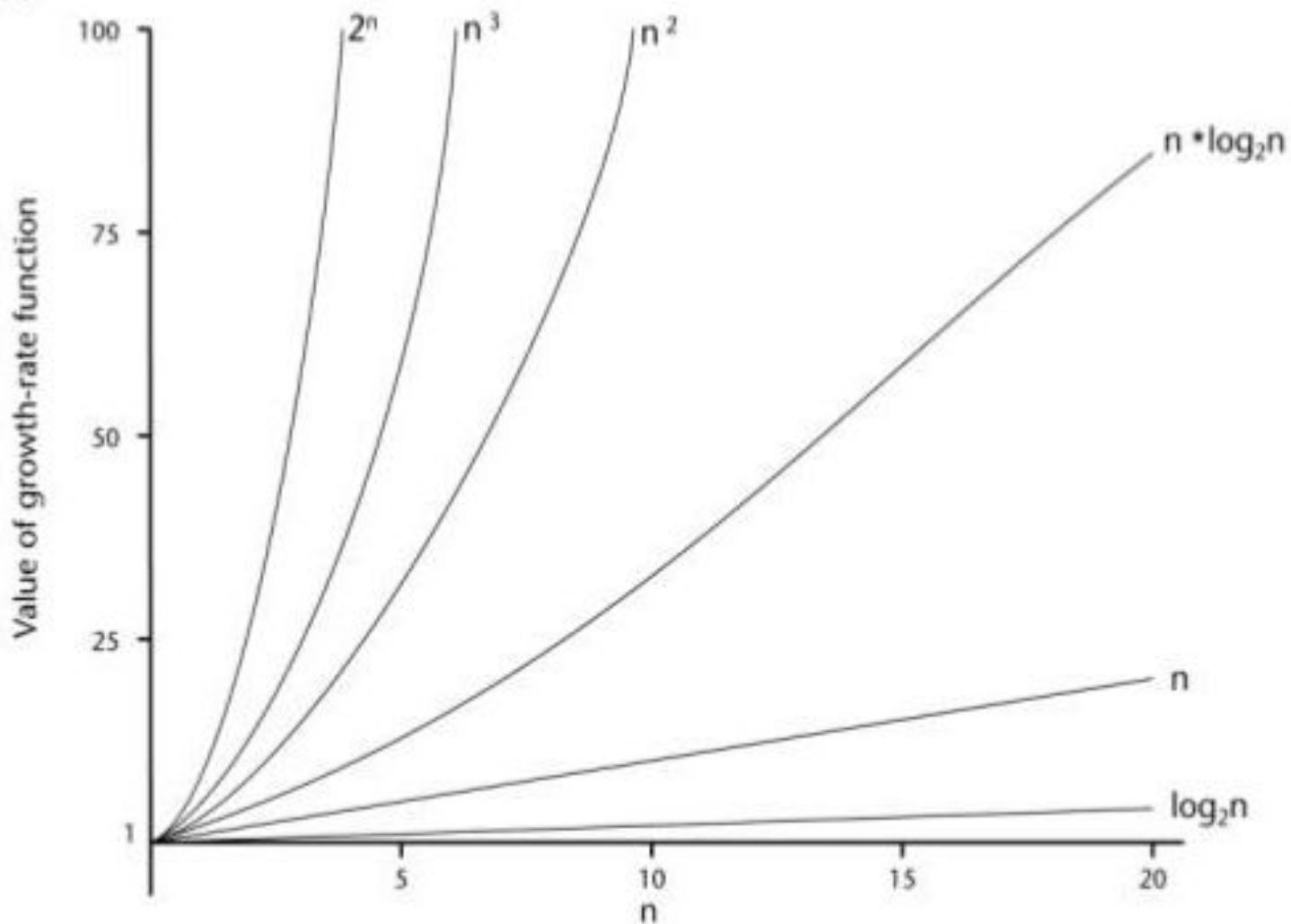
```
sum = 0;  
for( i = 0; i < n; i++)  
    sum++;  
val = 1;  
for( j = 0; j < n*n; j++)  
    val = val * j;
```

Ans : $O(n^2)$

Order of Growth Function

	<i>constant</i>	<i>logarithmic</i>	<i>linear</i>	<i>N-log-N</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
<i>n</i>	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

(b)



18CSC201J – Data Structures Session 9 - Bubblesort

Lecture by

V.LAVANYA

Department of Information Technology
SRM Institute of Science and technology

BUBBLE SORT

- Bubble Sort is the simplest sorting algorithm
- Works by repeatedly swapping the adjacent elements if they are in wrong order.
- Easier to implement, but slower than Insertion sort



Example:

- Let us consider the input array of size, N=5
- $A[5] = \{5, 1, 4, 2, 8\}$

First Pass:

- $(5 \ 1 \ 4 \ 2 \ 8) \rightarrow (1 \ 5 \ 4 \ 2 \ 8)$,
- Here, algorithm compares the first two elements, and swaps since $5 > 1$.
 $(1 \ 5 \ 4 \ 2 \ 8) \rightarrow (1 \ 4 \ 5 \ 2 \ 8)$, Swap since $5 > 4$
 $(1 \ 4 \ 5 \ 2 \ 8) \rightarrow (1 \ 4 \ 2 \ 5 \ 8)$, Swap since $5 > 2$
 $(1 \ 4 \ 2 \ 5 \ 8) \rightarrow (1 \ 4 \ 2 \ 5 \ 8)$,
- Now, since these elements are already in order ($8 > 5$), algorithm does not swap them

Second Pass

- $(1\ 4\ 2\ 5\ 8) \rightarrow (1\ 4\ 2\ 5\ 8)$
 $(1\ 4\ 2\ 5\ 8) \rightarrow (1\ 2\ 4\ 5\ 8)$, Swap since $4 > 2$
 $(1\ 2\ 4\ 5\ 8) \rightarrow (1\ 2\ 4\ 5\ 8)$
 $(1\ 2\ 4\ 5\ 8) \rightarrow (1\ 2\ 4\ 5\ 8)$
- Now, the array is already sorted, but the algorithm does not know if it is completed.
- The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

- $(1\ 2\ 4\ 5\ 8) \rightarrow (1\ 2\ 4\ 5\ 8)$
- $(1\ 2\ 4\ 5\ 8) \rightarrow (1\ 2\ 4\ 5\ 8)$
- $(1\ 2\ 4\ 5\ 8) \rightarrow (1\ 2\ 4\ 5\ 8)$
- $(1\ 2\ 4\ 5\ 8) \rightarrow (1\ 2\ 4\ 5\ 8)$

Pseudo-code: bubble sort

Bubble_sort(A , N)

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

Step 2: Repeat for J=0 to N - I

Step 3: IF A[J] > A[J + 1]

 SWAP A[J] and A[J + 1]

 [END OF INNER LOOP]

 [END OF OUTER LOOP]

Step 4: EXIT

Bubble sort program in C

```
/* Bubble sort code */
```

```
#include <stdio.h>
int main( )
{
    int array[100], n, c, d, t, swap;
    printf("Enter number of elements:\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (t = 0; t < n; t++)
    {
        scanf("%d", &array[t]);
    }
```

```
for (c = 0 ; c < n - 1; c++)
{
    for (d = 0 ; d < n - c - 1; d++)
    {
        if (array[d] > array[d+1])
        /* For decreasing order use '<' */
        {
            swap = array[d];
            array[d] = array[d+1];
            array[d+1] = swap;
        }
    }
}

printf("Sorted list in ascending order:\n");
for (c = 0; c < n; c++) {
    printf("%d\n", array[c]);
}
return 0;
}
```



Time Complexity:

- **Worst and Average Case Time Complexity:** $O(n^2)$.
The worst-case occurs when an array is reversely sorted.
For example: input array to be sorted is 5,4,3,2,1
- **Best Case Time Complexity:** $O(n)$.
The best-case occurs when an array is already sorted.
For example: input array to be sorted is 1,2,3,4,5



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

*Thank
you*



18CSC201J

DATA STRUCTURES AND ALGORITHMS

Session 11

Data Structures and its Types

Prepared by

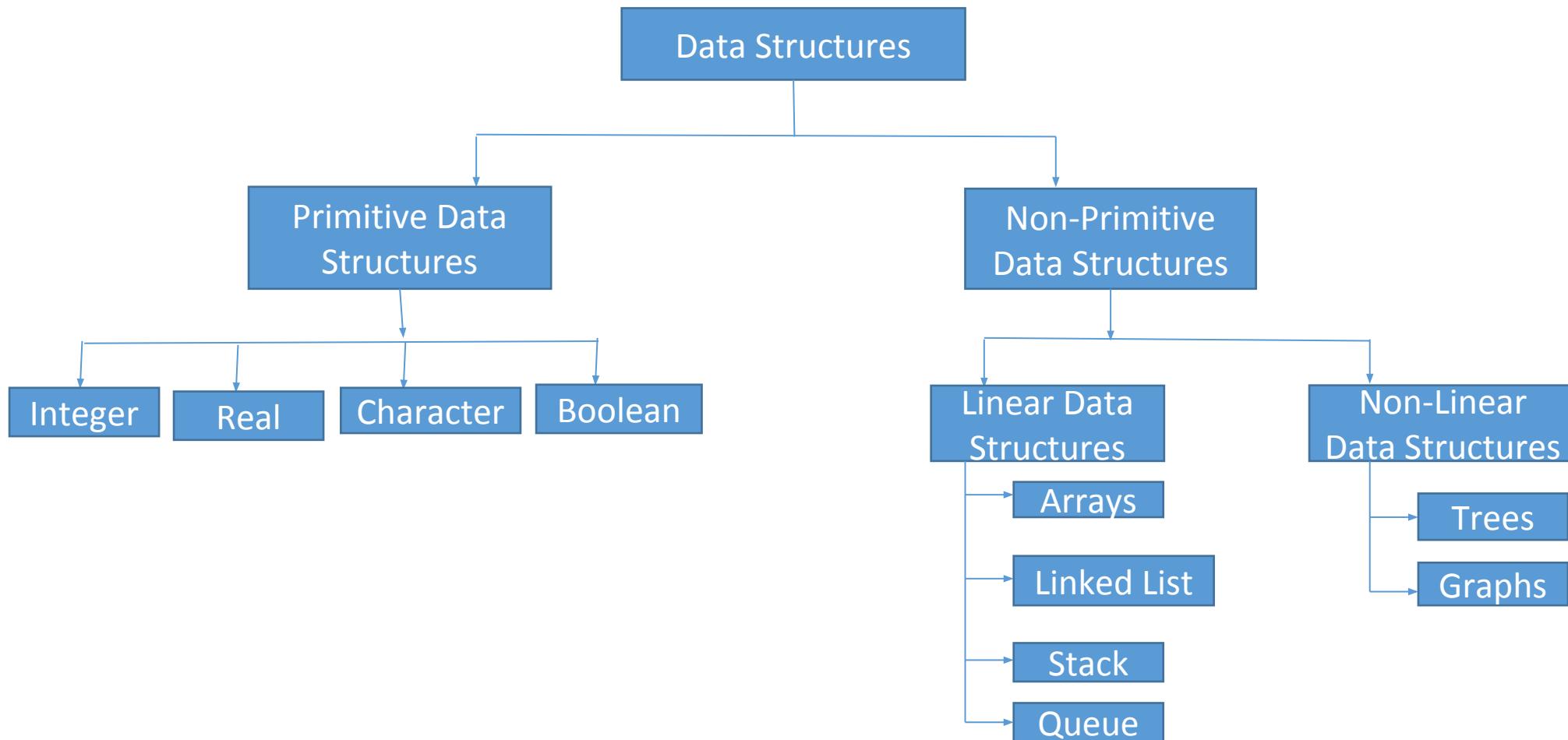
S. Sivasankari

Assistant Professor, Department of IT, SRM IST

Data Structure

- Collecting and Organizing data
- Perform operations on data in effective way
- Data refers to a value or set of values.
- Data can be organized in many ways
 - logical
 - Mathematical

Types of Data Structures



Types of Data Structures

1. Primitive

- Integer
- Real
- Character
- Boolean

2. Non-Primitive

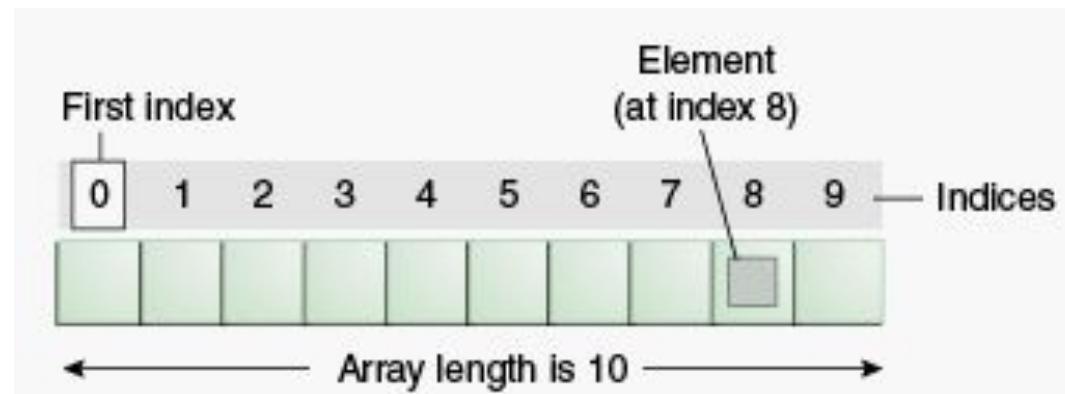
- Linear
- Non-Linear

Linear and Non-Linear Data Structures

- A data structure is considered as linear if the elements are arranged in a linear fashion(elements form a sequence).
- Examples: Arrays, Linked List, Stacks and Queues
- A data structure is considered as Non-Linear if the elements not arranged in a linear fashion(not in sequence).
- Examples: Trees, Graphs

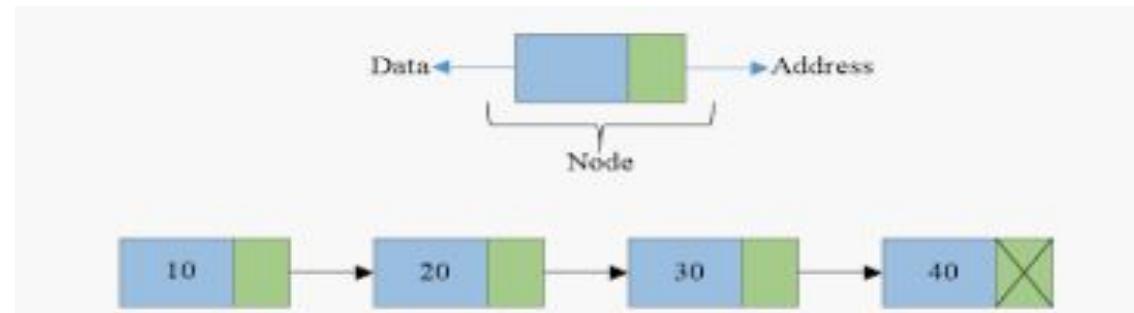
Arrays

- Sequential collection of elements
- Similar data type.
- Adjacent memory locations.



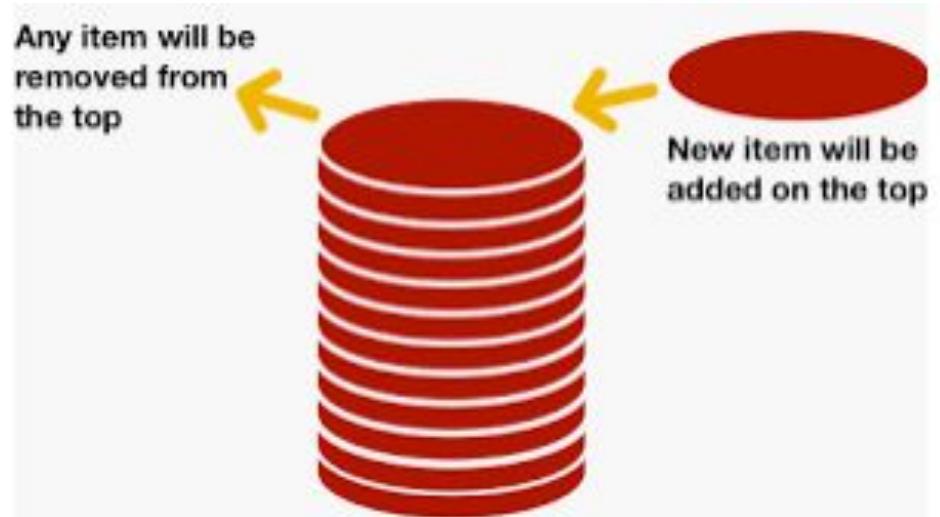
Linked List

- Dynamic memory management
- Run time memory allocated
- No memory wastage
- Direct access to elements restricted



Stack

- LIFO - Last In First Out
- Elements inserted at last will be removed first.
- Application of Stack includes
 - Solving Recursion
 - Towers of Hanoi
 - Evaluating Postfix Expression
 - Depth First Search



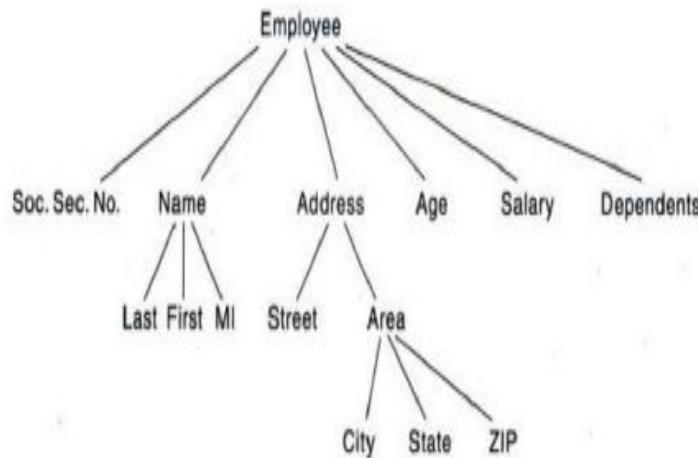
Queue

- FIFO - First In First Out
- The element inserted first will be removed first.
- The variants of Queue includes
 - Double Ended Queue
 - Circular Queue
 - Priority Queue.
- Application of Queue includes
 - Shared Resource access
 - Multiprogramming and
 - Message Queue



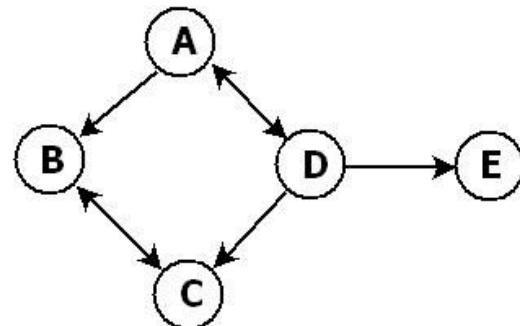
Trees

- Hierarchical in nature.
- Root - Top element of tree
- It is an useful data structure to store hierarchical information.
- Example – Directory structure of file system



Graph

- Networked data structure.
- It consists of a collection of nodes(Vertices) connected by edges.
- An edge is a path/link between two nodes.
- The edge can be directed/undirected.
- If a path is directed then you can travel in one direction alone.
- If a path is undirected then you can travel in both directions.



Thank You.....

18CSC201J

DATA STRUCTURES AND ALGORITHMS

Session 12

1D, 2D Array Initialization and Accessing using Pointers

Prepared by

S. Sivasankari

Assistant Professor, Department of IT, SRM IST

1D Array Initialization and Accessing using Pointers

- The pointer can be used to store address of arrays.
- Consider we have an array of integer then

Step1 : Declare the integer array

```
int a[] = {12,34,42,11,10};
```

Step 2: Declare integer pointer

```
int *p;
```

Step 3: Initialize pointer with base address of array of integers.

```
p = a; // return base address of array 'a'(using array name)
```

or

```
p = &a[0]; // return base address of array 'a'(using address of first  
element of array )
```

1D Array Initialization and Accessing using Pointers

Step 4: To access the array element we use $*(p + n)$
where n is the number of element.

i.e. $*(p+0)$ returns $a[0]$ value

$*(p+1)$ returns $a[1]$ value

.

.

.

$*(p+n)$ returns $a[n]$ value

1D Array Initialization and Accessing using Pointers

Sample C Code:

```
#include <stdio.h>

int main()
{
    int a[]={12,34,42,11,10};
    int *p, i;
    p = a; //initialize pointer
    for(i=0; i<5; i++)
        printf("%d\n",*(p+i));
    return 0;
}
```

Output:

12

34

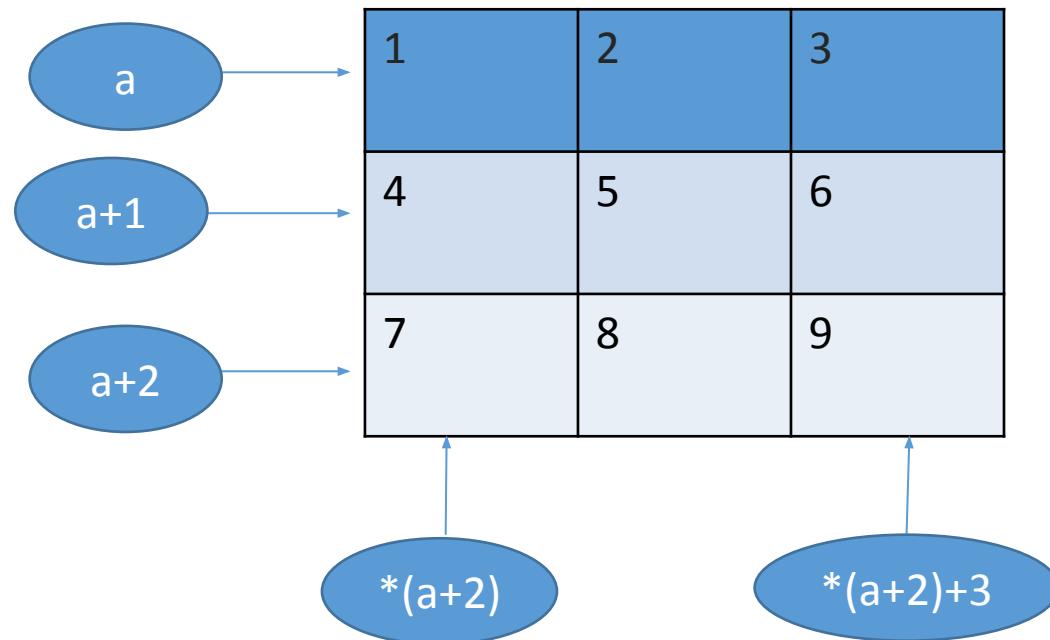
42

11

10

2D Array Initialization and Accessing using Pointers

- We can access an 2D array using pointers as $\ast(\ast(a+i) + j)$
- The expression $\ast(a + i)$ provides the base address of i^{th} 1D array.
- $\ast(a + i) + j$ provides the address of j^{th} element of i^{th} 1D array



2D Array Initialization and Accessing using Pointers

Sample Code:

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

```
int main()
```

```
{  int a[3][4] = { { 11, 10, 31, 40 },  
                   { 15, 61, 72, 81 },  
                   { 9, 1, 1, 2 } };
```

```
int k, l;
```

```
for (k = 0; k < 3; k++)
```

```
{   for (l = 0; l < 4; l++)
```

```
    printf("%d ", *(*(a + k) + l));
```

```
    printf("\n"); }
```

```
return 0; }
```

Output:

11	10	31	40				
15	61	72	81				
9	1	1	2				

Thank You

Session 14-15: Implement Structures using pointers

Pointer to structure declaration

Syntax

```
struct structname {  
    data type member1;  
    data type member2;  
}variable1, *ptr_variable1
```

Accessing structure variables

Syntax 1

(*ptrvariable).member1;

Syntax 2

*ptrvariable -> member1;

Example -1

```
struct book                                ptr = &b1 ;
{                                           printf ( "\n%s %s %d", b1.name, b1.author,
char name[25];                           b1.callno ) ;
char author[25];                         printf (" \n%s %s %d", ptr->name,
int callno;                             ptr->author, ptr->callno ) ;
};

struct book b1 = { "Let us C",
"ABC", 101 } ;

struct book *ptr ;
```

Example - 2

```
struct dog
{
    char name[10];
    char breed[10];
    int age;
    char color[10];
};
```

```
struct dog my_dog = {"tyke", "Bulldog", 5, "white"};
struct dog *ptr_dog;
ptr_dog = &my_dog;

printf("Dog's name: %s\n", ptr_dog->name);
printf("Dog's breed: %s\n", ptr_dog->breed);
printf("Dog's age: %d\n", ptr_dog->age);
printf("Dog's color: %s\n", ptr_dog->color);

strcpy(ptr_dog->name, "jack");
ptr_dog->age++;
printf("Dog's new name is: %s\n", ptr_dog->name);
printf("Dog's age is: %d\n", ptr_dog->age);
```

Example -3

```
#include <stdio.h>
int main()
{
    struct student{
        char name[40];
        struct avg{
            int sub1, sub2, sub3;
            float average;
        }avg;
    }stud1, *studptr = &stud1;
    //Pointer variable stores the
    //address of structure variable
```

```
int total;
printf("Enter the Name of the student : ");
scanf("%s", (*studptr).name);

printf("\nEnter sub1 marks : ");
scanf("%d",&studptr->avg.sub1);

printf("\nEnter sub2 marks : ");
scanf("%d",&studptr->avg.sub2);

printf("\nEnter sub3 marks : ");
scanf("%d",&studptr->avg.sub3);

total = (studptr->avg.sub1 + studptr->avg.sub2 +
studptr->avg.sub3);
```

Example -3 continues

```
studptr->avg.average = total / 3;  
printf("\n-----Student Details-----\n ");  
printf("Name: %s", studptr->name);  
printf("\nsub1: %d ", studptr->avg.sub1);  
printf("\nsub2: %d ", studptr->avg.sub2);  
printf("\nsub3: %d ", studptr->avg.sub3);  
printf("\nAverage: %f ", studptr->avg.average);  
return 0;  
}
```



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-1



Syllabus

- Array implementation of List – Traversing
- Insertion – Deletion – Application of List
- Linked list – Implementation – Insertion – Deletion and Search
- Double linked list
- Circular Linked List – Applications – Josephus Problem
- Cursor based implementation
- Polynomial Arithmetic
- Multidimensional Arrays-Sparse Matrix



Outline

- Abstract data Type (ADT)
- List ADT
- List ADT with Array Implementation
- Linked lists
- Basic operations of linked lists
 - Insert, find, delete, print, etc.
- Variations of linked lists
 - Circular linked lists
 - Doubly linked lists



ADT = properties + operations

- An **ADT** describes a set of objects sharing the same properties and behaviors
 - The **properties** of an ADT are its **data** (representing the internal state of each object)
 - `double d;` -- bits representing exponent & mantissa are its data or state
 - The **behaviors** of an ADT are its **operations or functions** (operations on each instance)
 - `sqrt(d) / 2;` //operators & functions are its behaviors



The List ADT

- A sequence of zero or more elements

$A_1, A_2, A_3, \dots A_N$

- N: length of the list
- A_1 : first element
- A_N : last element
- A_i : position i
- If $N=0$, then empty list
- Linearly ordered
 - A_i precedes A_{i+1}
 - A_i follows A_{i-1}



Operations

- printList: print the list
- makeEmpty: create an empty list
- find: locate the position of an object in a list
 - list: 34,12, 52, 16, 12
 - find(52) → 3
- insert: insert an object to a list
 - insert(x,3) → 34, 12, 52, x, 16, 12
- remove: delete an element from the list
 - remove(52) → 34, 12, x, 16, 12
- findKth: retrieve the element at a certain position



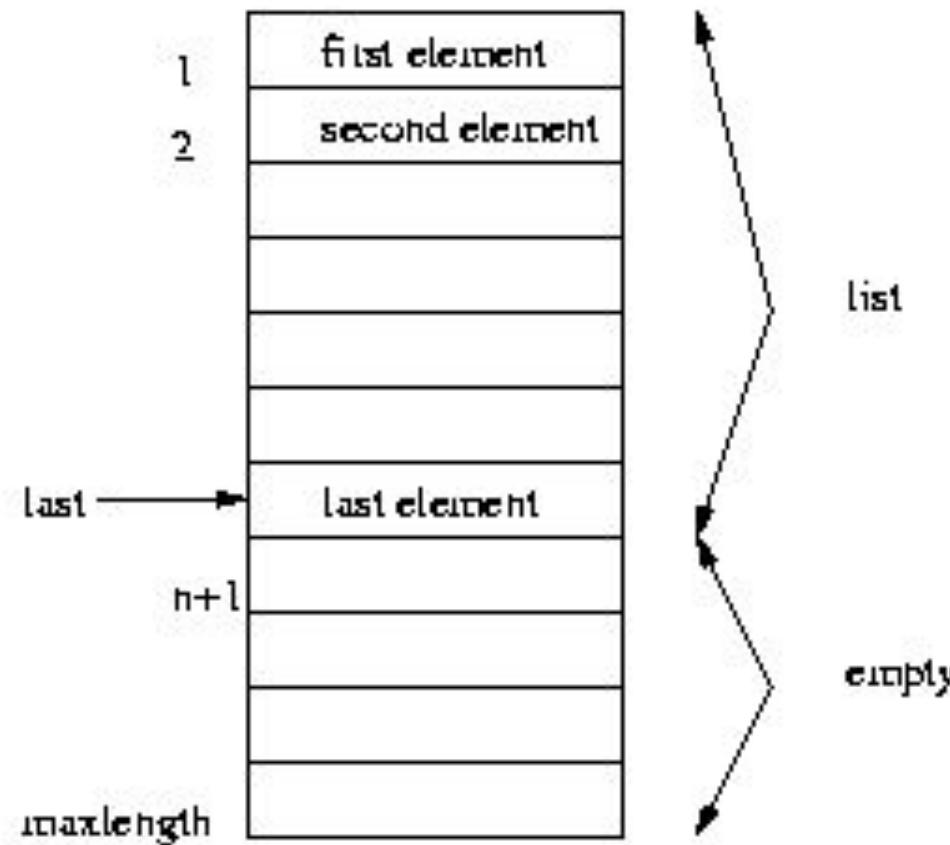
Implementation of an ADT

- Choose a **data structure** to represent the ADT
 - E.g. arrays, records, etc.
- Each operation associated with the ADT is implemented by one or more subroutines
- Two standard implementations for the list ADT
 - Array-based
 - Linked list



Array Implementation

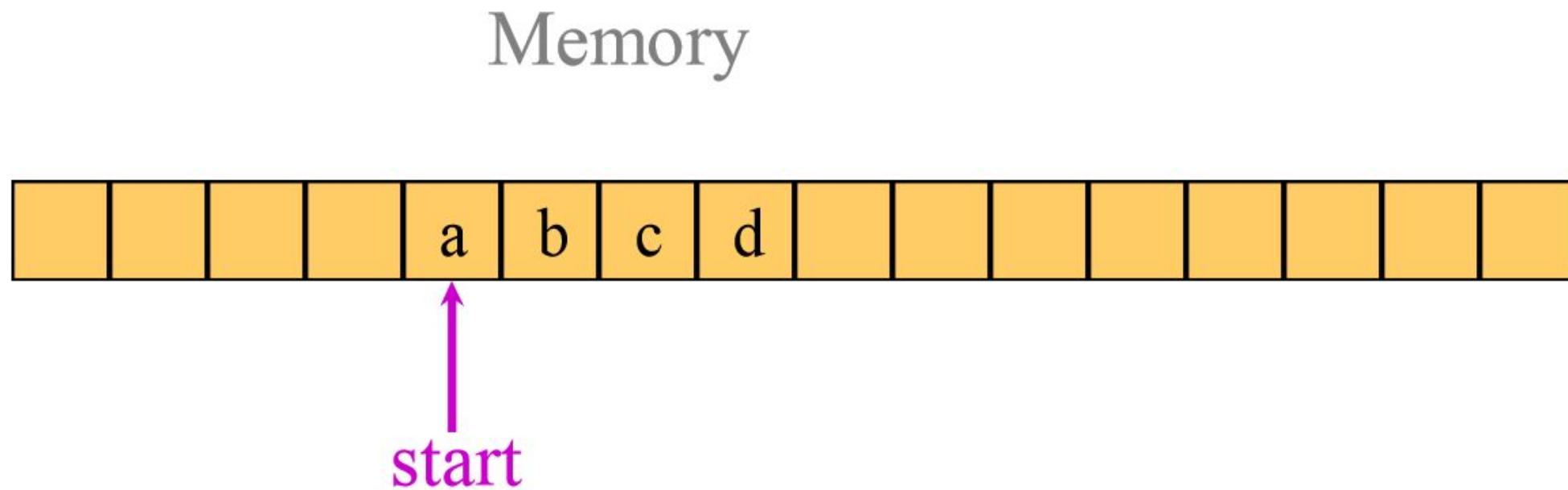
- Elements are stored in contiguous array positions





1D Array Representation

- 1-dimensional array $x = [a, b, c, d]$
- map into contiguous memory locations





2D Arrays

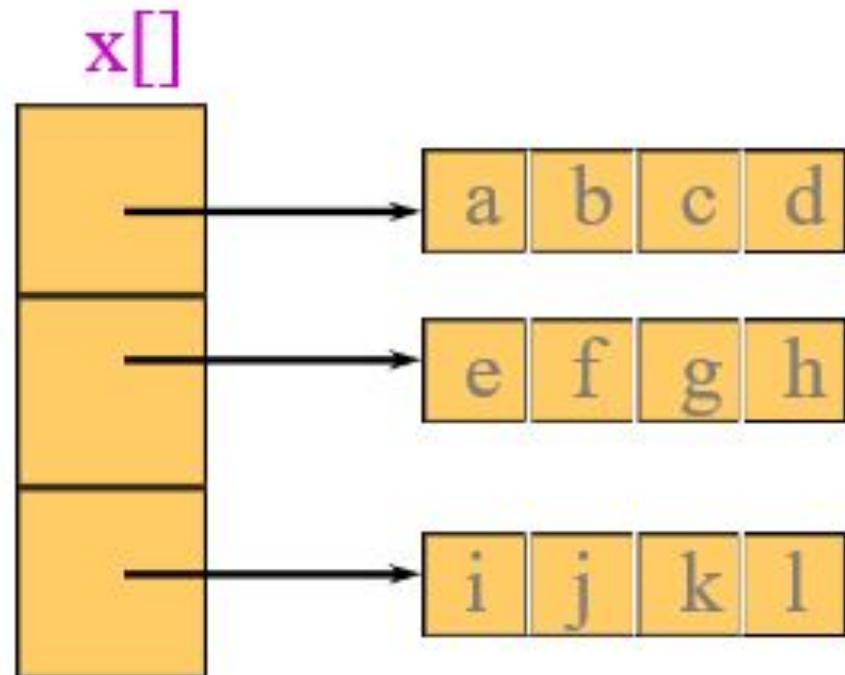
- The elements of a 2-dimensional array a declared as:

```
int a[3][4] ;
```

- may be shown as a table

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

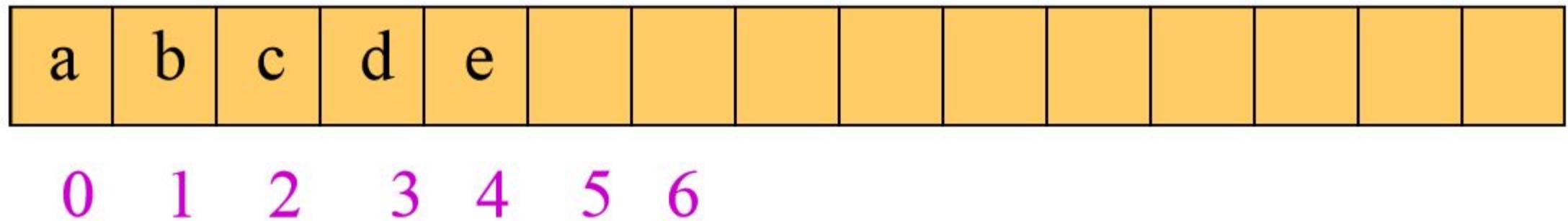
2D Array Representation





Linear List Array Representation

- use a one-dimensional array called `alphabet[]`



- List = (a, b, c, d, e)
- Store element i of list in `alphabet[i]`



Add An Element

size = 5

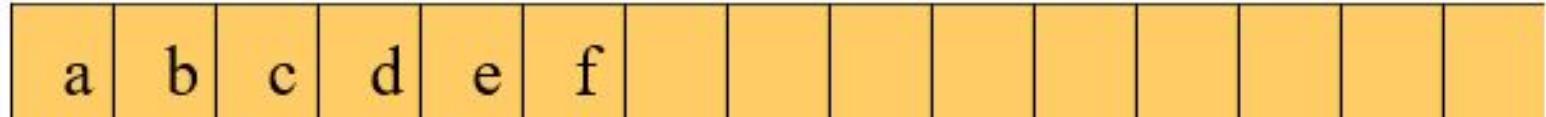


add(6,f)

add(k,val)
for(i=size-1,i>=k,i--)
x[i+1]=x[i];

size = 6

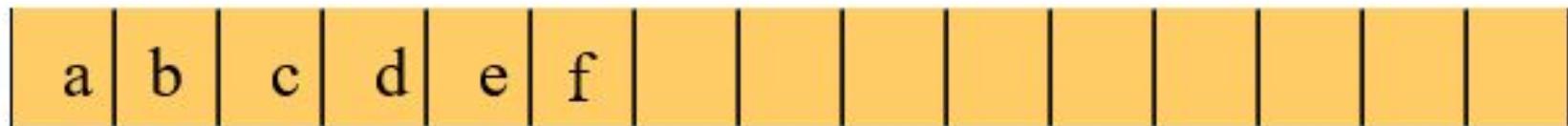
x[k]=val; size++;





Remove An Element

size = 6

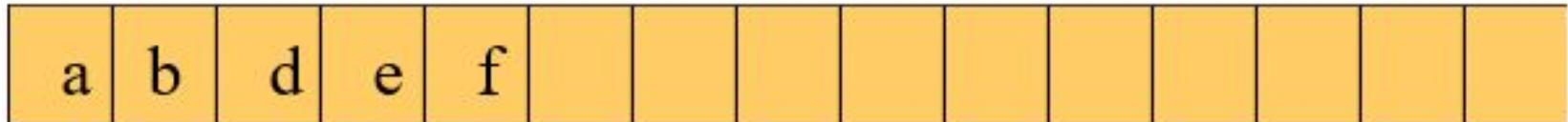


remove(2)

```
remove(k)
val=x[k];
for(i=k,i<size,i++)
    x[i]=x[i+1];
```

size = 5

```
size--; return(val);
```





References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-2



1. Applications on Arrays
2. Multidimensional Arrays- Sparse Matrix



Array Applications

- Stores Elements of Same Data Type
- Array Used for Maintaining multiple variable names using single name
- Array Can be Used for Sorting Elements
- Array Can Perform Matrix Operation
- Array Can be Used in CPU Scheduling
- Array Can be Used in Recursive Function



Array Implementation...

- Requires an estimate of the maximum size of the list
 - waste space
- printList and find: $O(n)$
- findKth: $O(k)$
- insert and delete: $O(n)$
 - e.g. insert at position 0 (making a new element)
 - requires first pushing the entire array down one spot to make room
 - e.g. delete at position 0
 - requires shifting all the elements in the list up one
 - On average, half of the lists needs to be moved for either operation



- Array Declaration

- Int arr[10];
- b=10;
- int arr1[b];
- int [b+5];

- 2-D Array
- int arr2[4][5];



Dynamic array declaration

- int size_arr;
- scanf("&d",&size_arr);
- int *arr1 = (int*)malloc(sizeof(int)*size_arr);
- int p11[size];
- for(i = 0; i < size; i++)
 - { scanf("%d",&p11[i])
 - }
- Or
- for(i = 0; i < size; i++)
 - { scanf("%d",*(p11+i));
 - }

Multi-dimensional SPARSE Matrix



- A matrix is a two-dimensional data object made of m rows and n columns, therefore having $m \times n$ values. When $m=n$, we call it a **square matrix**.
- There may be a situation in which a matrix contains **more number of ZERO** values than NON-ZERO values. Such matrix is known as **sparse matrix**.
- When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix.
- For example, consider a matrix of size 100×100 containing only 10 non-zero elements.

Contd...



- In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero.
- totally we allocate $100 \times 100 \times 2 = 20000$ bytes of space to store this integer matrix.
- to access these 10 non-zero elements we have to make scanning for 10000 times.
- If most of the elements in a matrix have the value 0, then the matrix is called sparse matrix.

Example For 3 X 3 Sparse Matrix:

-

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 4 & 0 \end{vmatrix}$$

Sparse Matrix Representations



- A sparse matrix can be represented by using TWO representations, those are as follows...
- Triplet Representation
- Linked Representation



TRIPLET REPRESENTATION (ARRAY REPRESENTATION)

- In this representation, only non-zero values are considered along with their row and column index values.
- The array index [0,0] stores the total number of rows, [0,1] index stores the total number of columns and [0,2] index has the total number of non-zero values in the sparse matrix.
- For example, consider a matrix of size 5×6 containing 6 number of non-zero values.

TRIPLET REPRESENTATION


$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$


Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2



Array Implementation

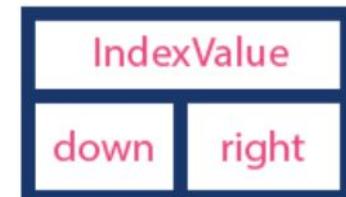
```
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d ",Arr[i][j]);
    }
    printf("\n");
}
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        if(Arr[i][j]!=0)
        {
            B1[s1][0]=Arr[i][j];
            B1[s1][1]=i;
            B1[s1][2]=j;
            s1++; } }
}
```



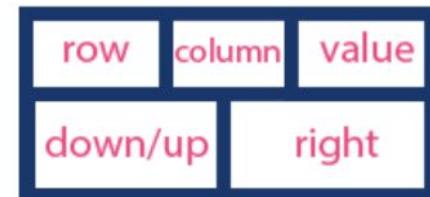
Linked List Representation

- In linked list representation, a linked list data structure is used to represent a sparse matrix.
- In linked list, each node has four fields. These four fields are defined as:
 - **Row:** Index of row, where non-zero element is located
 - **Column:** Index of column, where non-zero element is located
 - **Value:** Value of the non zero element located at index – (row,column)
 - **Next node:** Address of the next node

Header Node

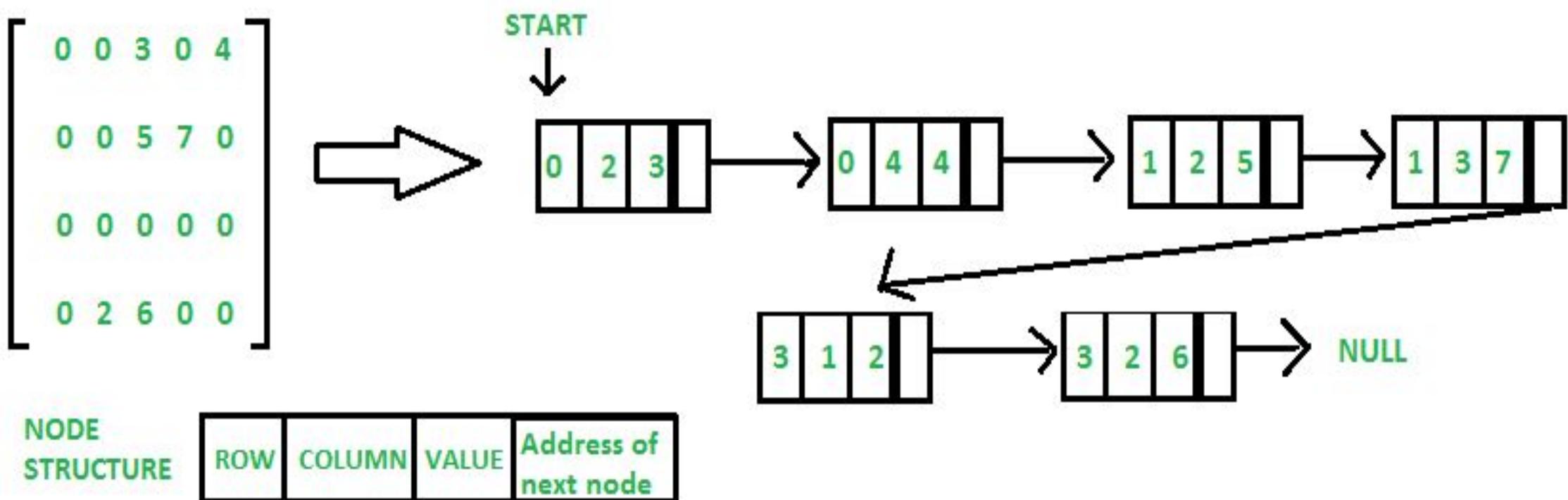


Element Node





Linked List Representation



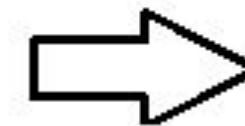


Node Declaration

```
struct Node
{
    int value;
    int row_position;
    int column_postion;
    struct Node *next;
};
```



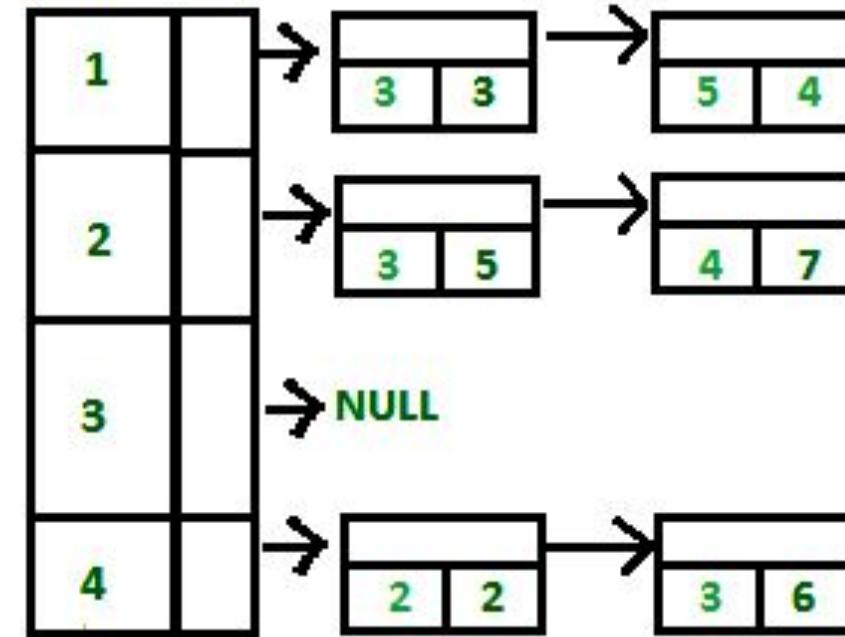
0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0



Value - List
NODE
STRUCTURE

Address of next node	
Column index	Value

Row - List





References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-3

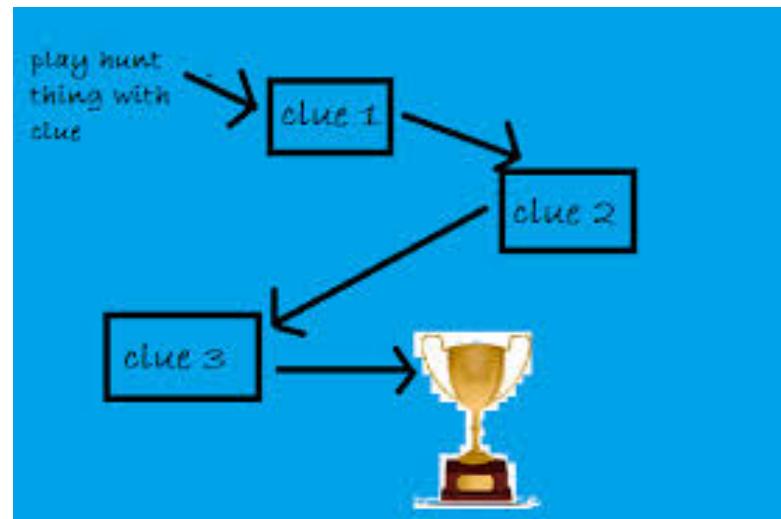


1. Linked List Implementation – Insertion
2. Linked List- Deletion and Search



Linked List

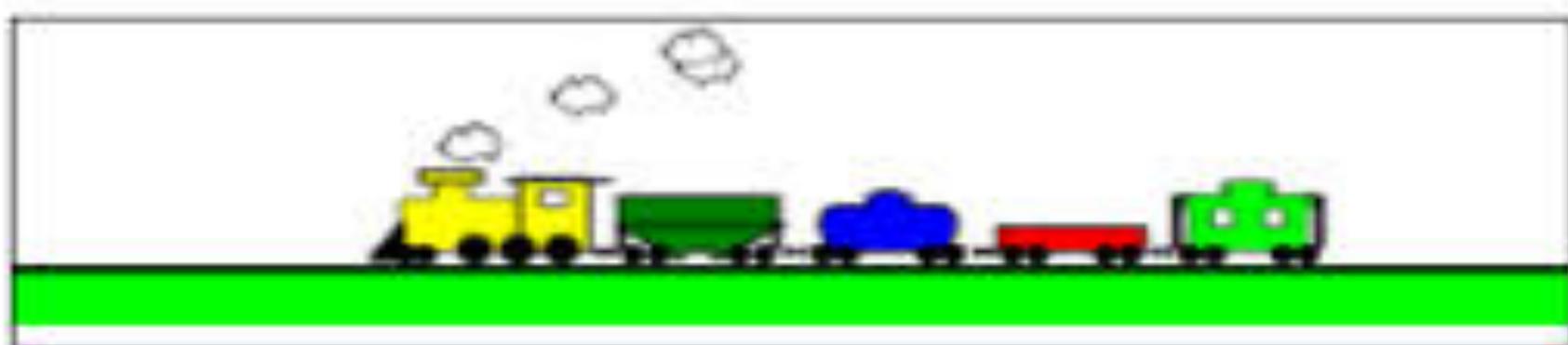
- Alternate approach to maintaining an array of elements
- Rather than allocating one large group of elements, allocate elements as needed
- Q: how do we know what is part of the array?
A: have the elements keep track of each other
 - use pointers to connect the elements together as a *LIST* of things





Linked List ADT

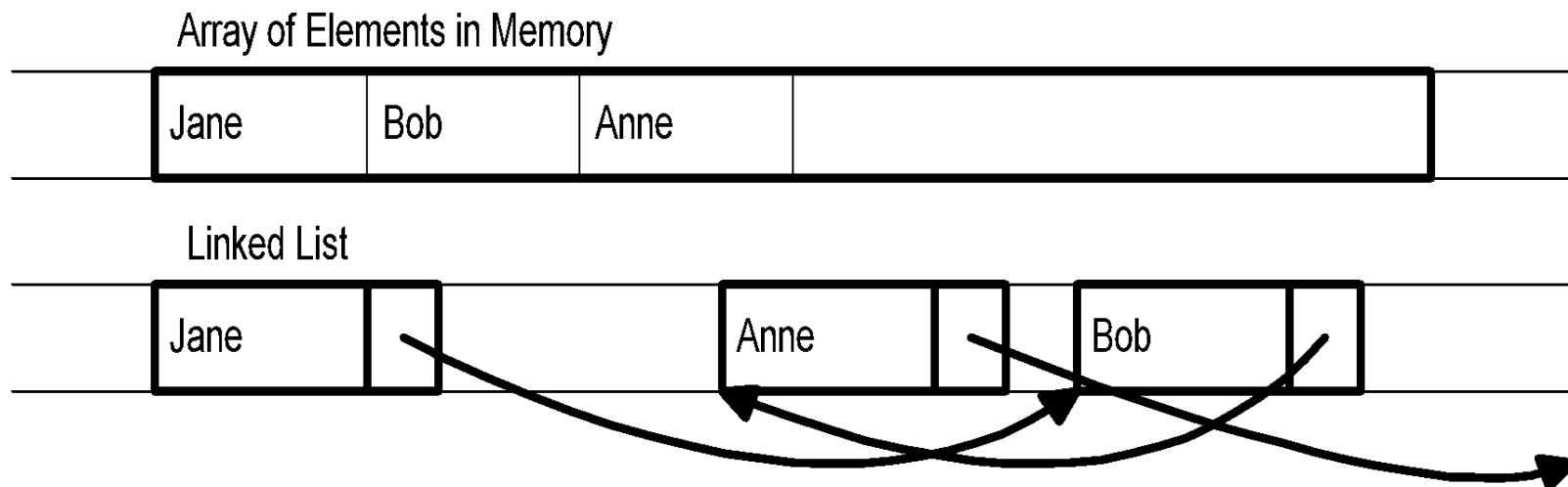
- A linked list is a series of connected *nodes*, where each node is a data structure.
- A linked list can grow or shrink in size as the program runs
- Insertion and deletion of nodes is quicker





Dynamically Allocating Elements

- Allocate elements one at a time as needed, have each element keep track of the *next* element
- Result is referred to as linked list of elements, track next element with a pointer





Pointer Implementation (Linked List)

- Ensure that the list is not stored contiguously
 - use a linked list
 - a series of structures that are not necessarily adjacent in memory

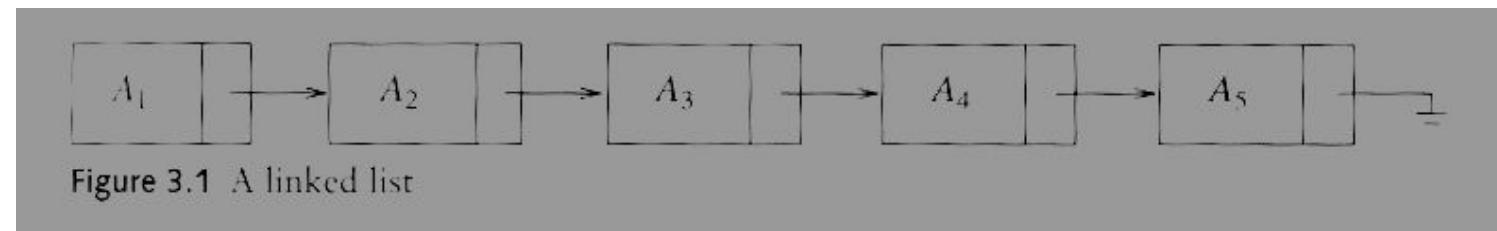
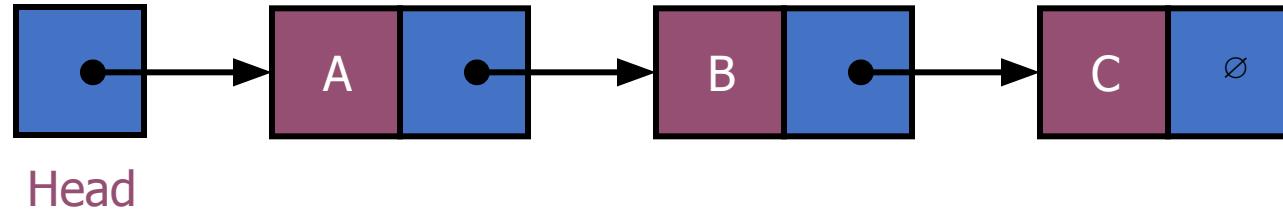


Figure 3.1 A linked list

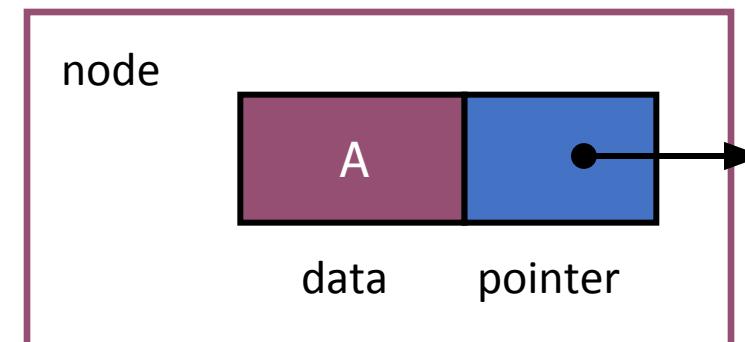
- Each node contains the element and a pointer to a structure containing its successor
 - the last cell's next link points to NULL
- Compared to the array implementation,
 - ✓ the pointer implementation uses only as much space as is needed for the elements currently on the list
 - but requires space for the pointers in each cell



Linked Lists



- A *linked list* is a series of connected *nodes*
- Each node contains at least
 - A piece of data (any type)
 - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to NULL





Linked Implementation...

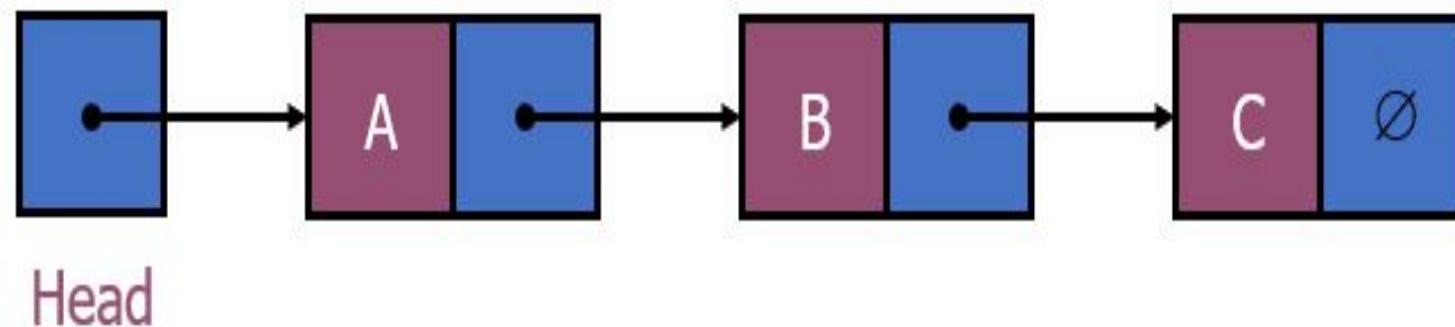
- Requires no estimate of the maximum size of the list
 - No wasted space
- printList and find: $O(n)$
- findKth: $O(k)$
- insert and delete: $O(1)$
 - e.g. insert at position 0 (making a new element)
 - Insert does not require moving the other elements
 - e.g. delete at position 0
 - requires no shifting of elements
- Insertion and deletion becomes easier, but finding the Kth element moves from $O(1)$ to $O(n)$



Variations of Linked Lists

- *Singly linked lists*

- Each node points has a successor
- There will be NULL value in the next pointer field of the last node in the list

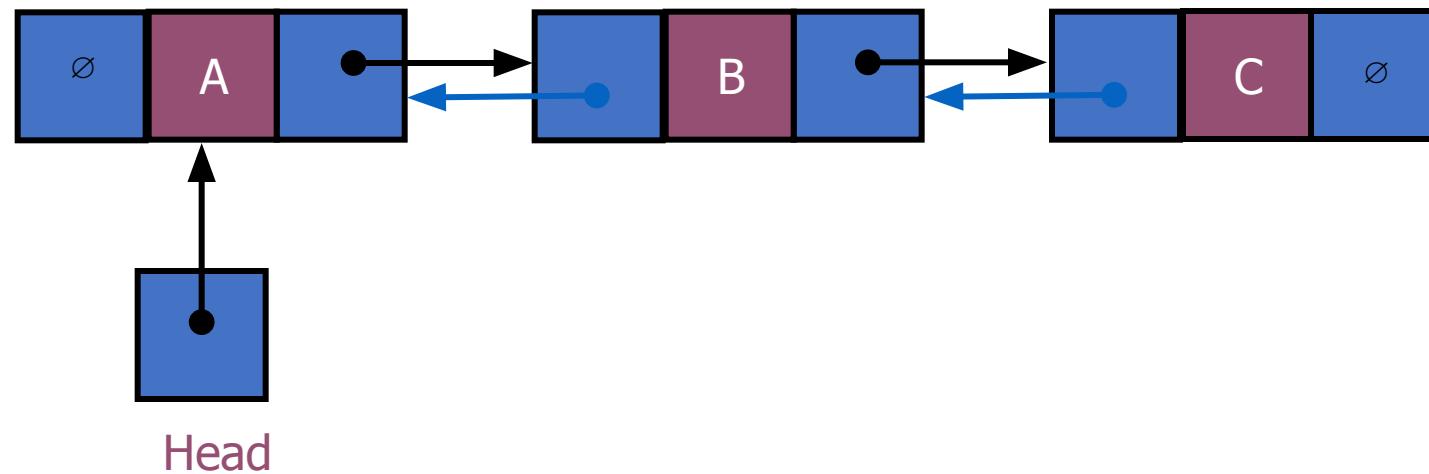




Variations of Linked Lists

- *Doubly linked lists*

- Each node points to not only successor but the predecessor
- There are two NULL: at the first and last nodes in the list
- Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists **backwards**

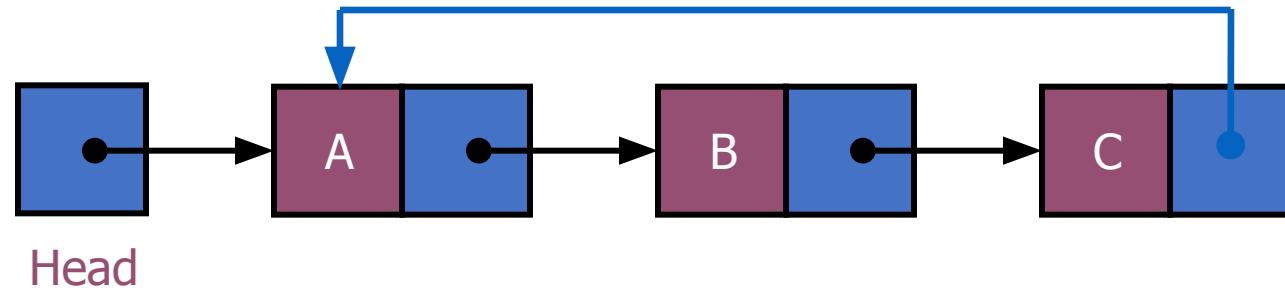




Variations of Linked Lists

- *Circular linked lists*

- The last node points to the first node of the list



Head

- How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)



Singly Linked List (Pointer-Based)

- Creating the structure of a node in C program using pointers

```
struct Node
{
    int data;
    struct Node *next;
}*head = NULL;
```

- A node is dynamically allocated

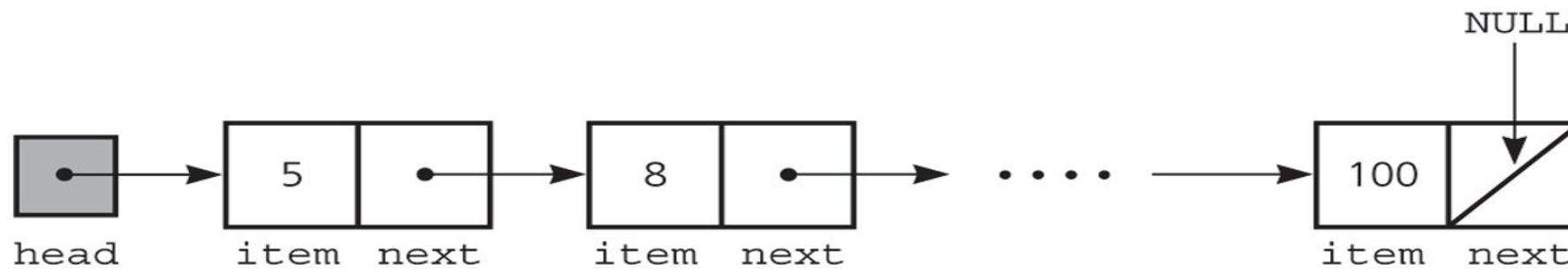
```
node *p;
p = malloc(sizeof(Node));
```



- The head pointer points to the first node in a linked list
- If head is NULL, the linked list is empty

head=NULL

head=malloc(sizeof(Node))





CREATION OF A NODE TO THE LIST

- To append a node to a linked list means to add the node to the end of the list.

Begin Node Creation

Create a new node.

Store data in the new node.

If there are no nodes in the list

Make the new node the first node.

Else

Traverse the List to Find the last node.

Add the new node to the end of the list.

End If.

End Node Creation



A Simple Linked List Class

- Operations of List
 - IsEmpty: determine whether or not the list is empty
 - InsertNode: insert a new node at a particular position
 - FindNode: find a node with a given value
 - DeleteNode: delete a node with a given value
 - DisplayList: print all the nodes in the list



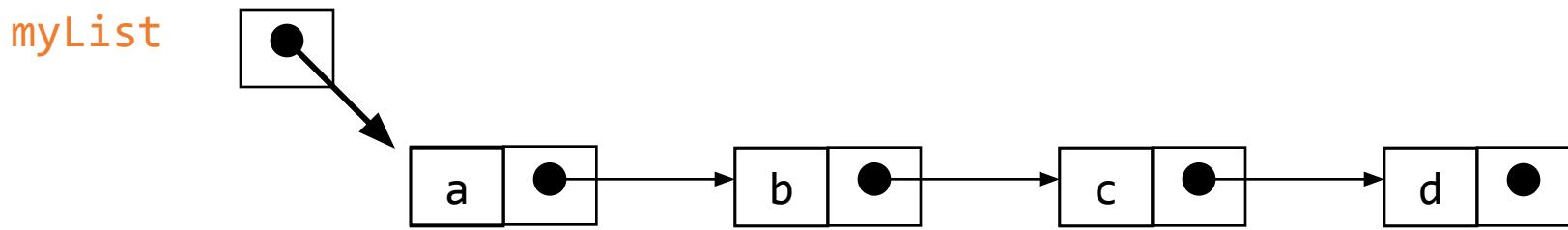
Inserting a new node

- Possible cases of InsertNode
 1. Insert into an empty list
 2. Insert in front
 3. Insert at back
 4. Insert in middle
- But, in fact, only need to handle two cases
 - Insert as the first node (Case 1 and Case 2)
 - Insert in the middle or at the end of the list (Case 3 and Case 4)



Singly-linked lists

- Here is a singly-linked list (SLL):

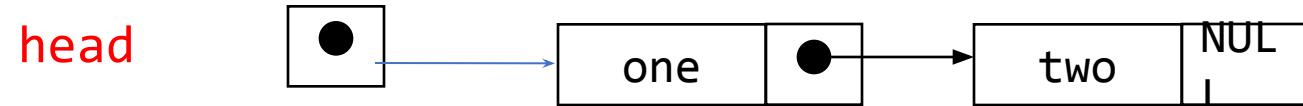


- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty)



Using a header node

- The entry point into a linked list is called the **head** of the list.
- It should be noted that head is not a separate node, but the reference to the first node.
- If the list is empty then the head is a null reference.



Inserting At Beginning of the list



- We can use the following steps to insert a new node at beginning of the single linked list...
- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then,
set **newNode→next = NULL** and **head = newNode**.
- **Step 4:** If it is **Not Empty** then,
set **newNode→next = head** and **head = newNode**.



CREATING and INSERTING A NODE

- Creating a new node

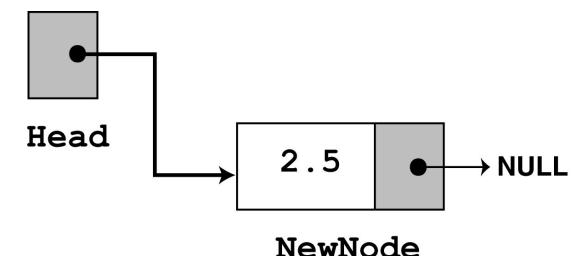
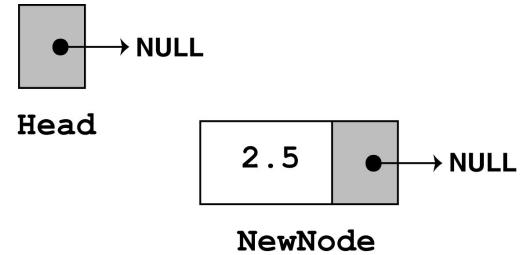
```
newNode = new ListNode;  
newNode->value = num;
```

If it's the first node set it to point **header**

```
head=newNode;  
head->next = NULL;
```

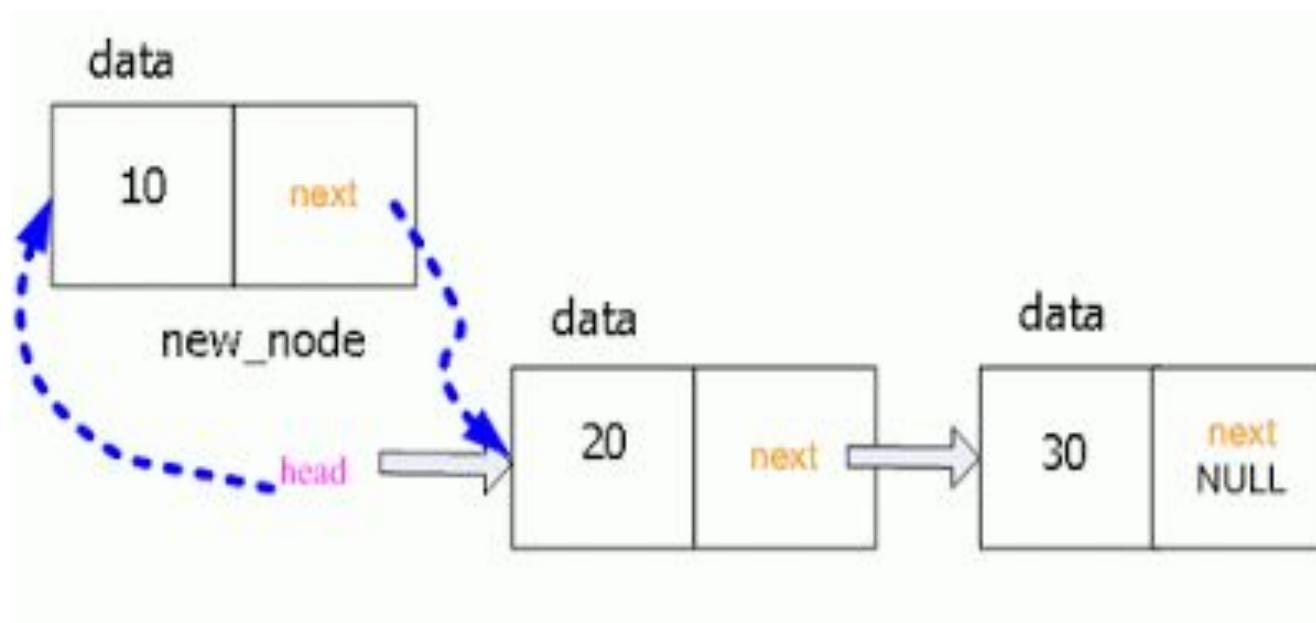
If already elements are there in the list

```
newNode->next = head;  
head=newNode;
```





Example



Contd...



```
void insertAtBeginning(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(head == NULL)
    {
        newNode->next = NULL;
        head = newNode;
    }
    else
    {
        newNode->next = head;
        head = newNode;
    }
    printf("\nOne node inserted!!!\n");
}
```

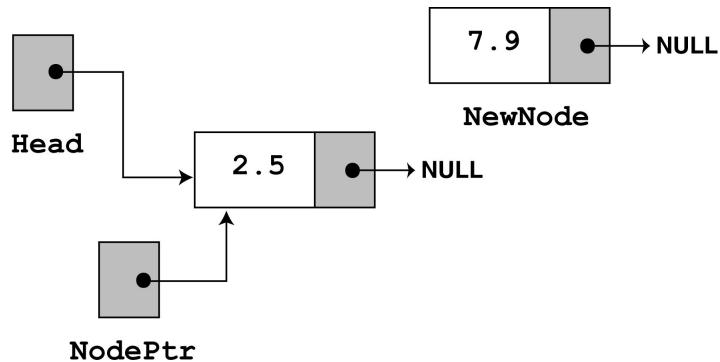


Inserting At End of the list

- **Step 1:** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2:** Check whether list is **Empty** (**head == NULL**).
- **Step 3:** If it is **Empty** then, set **head = newNode**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6:** Set **temp → next = newNode**.

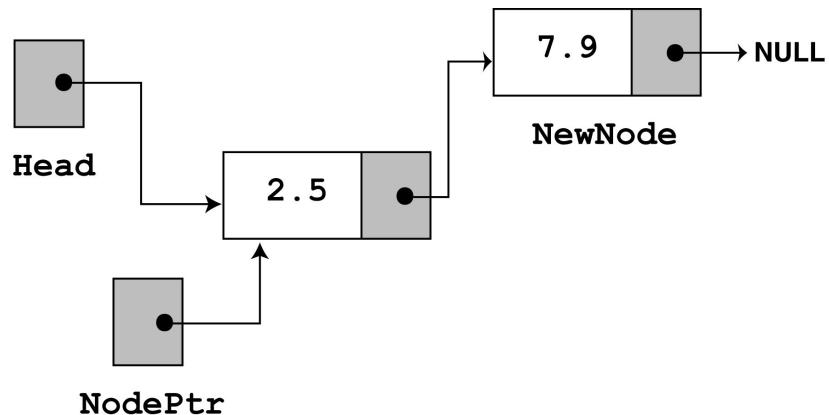


INSERTING A NEW NODE AT END



Use a Node Pointer to trace to the current position

Move the Node pointer to traverse till the end of the list



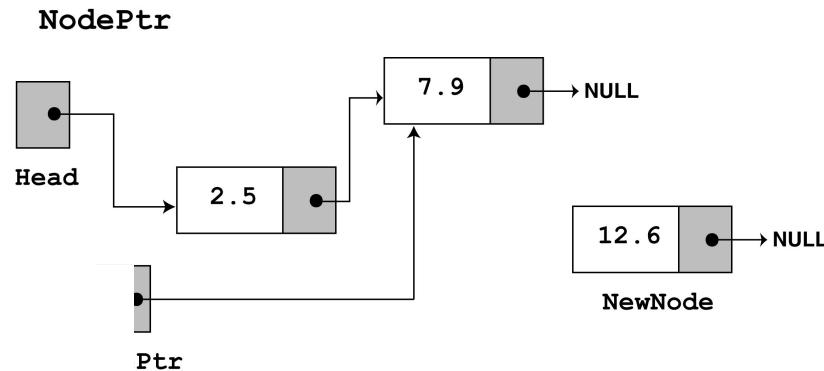
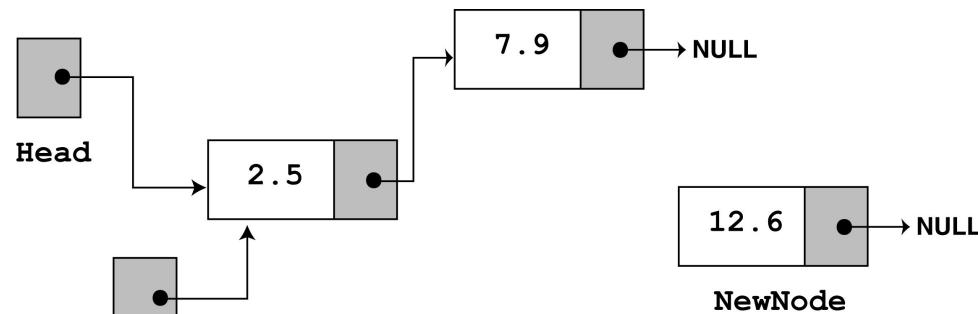
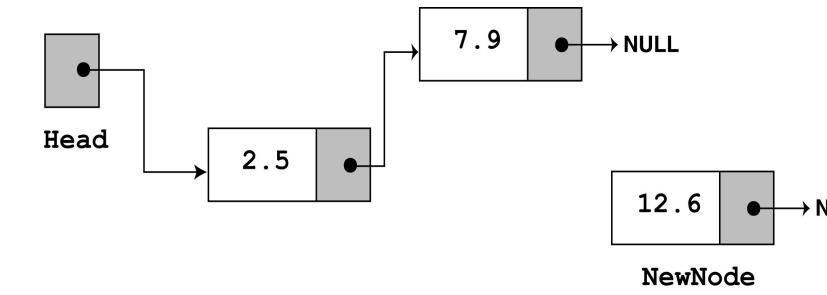
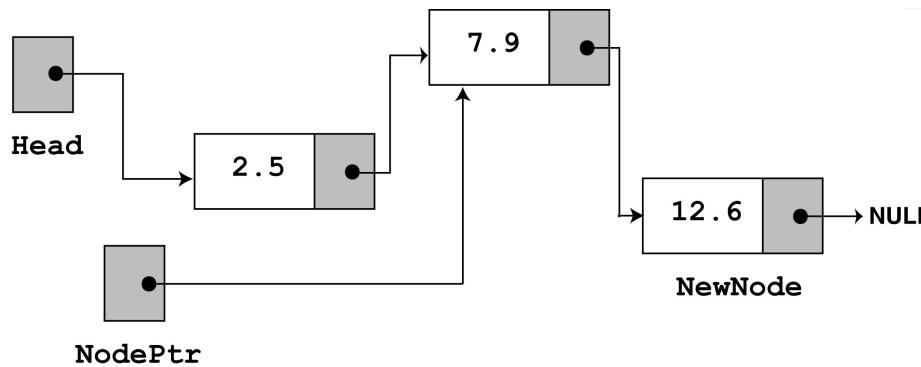


- If the Node pointer does not points to NULL then**
- **Move the Node pointer to the next node until it points to NULL**

```
nodePtr=nodePtr->next;
```

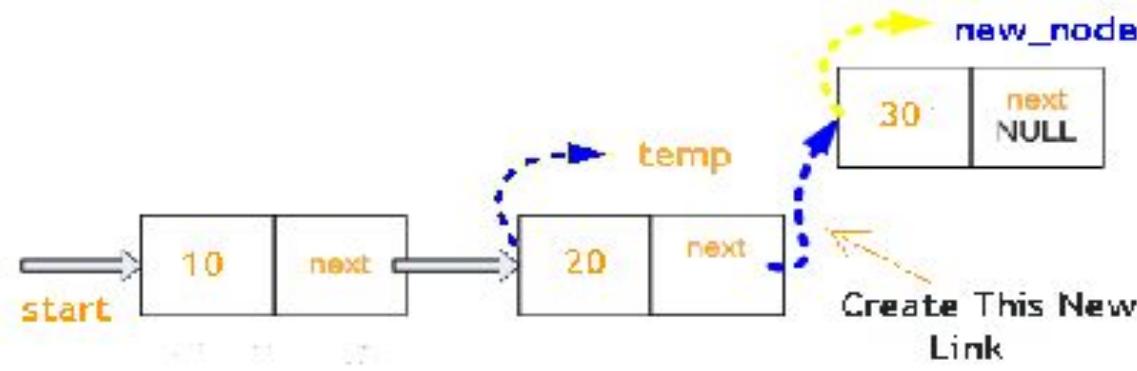
- **When it reaches NULL append the newNode at the last**

```
nodePtr ->next=newNode;  
newNode->next=NULL;
```





Example



Contd.....



```
void insertAtEnd(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if(head == NULL)
        head = newNode;
    else
    {
        struct Node *temp = head;
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    printf("\nOne node inserted!!!\n");
}
```

INSERTING A NEW NODE IN THE MIDDLE

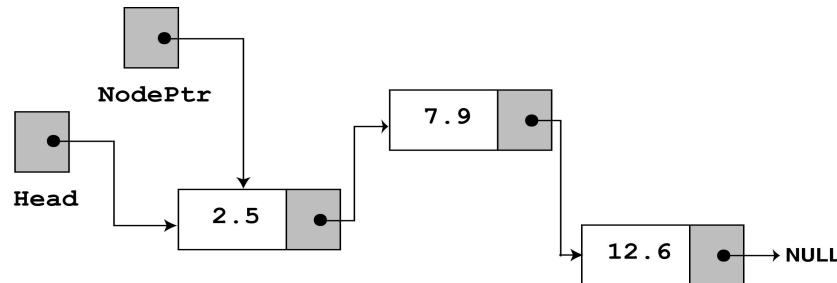


- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then, set **newNode → next = NULL** and **head = newNode**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp → data** is equal to **location**, here **location** is the node value after which we want to insert the **newNode**).
- **Step 6:** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.
- **Step 7:** Finally, Set '**newNode → next = temp → next**' and '**temp → next = newNode**'



INSERTING A NEW NODE IN THE MIDDLE

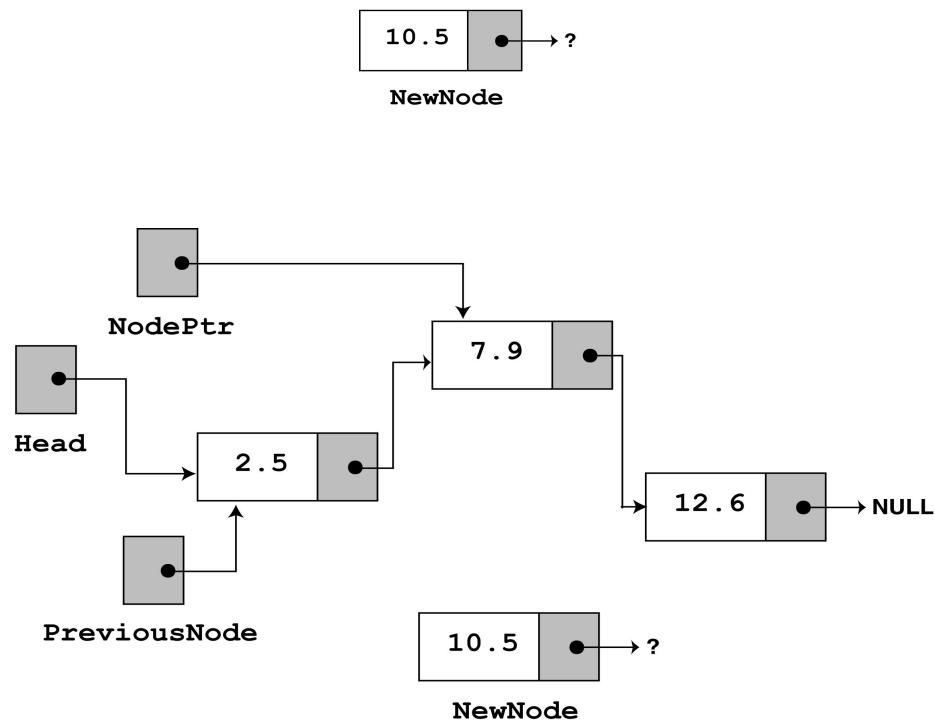
- Use a Node Pointer to trace to the current position



- Also to store the value of the next address

- Use a Previous Node Pointer to trace to the current position

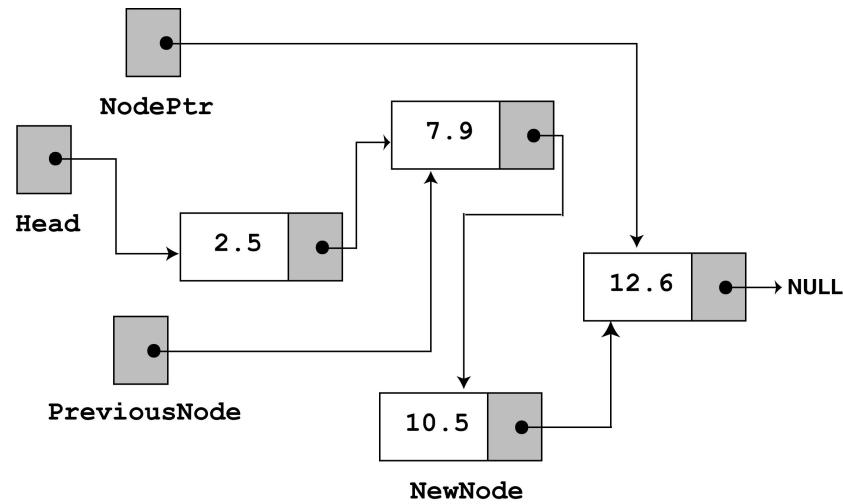
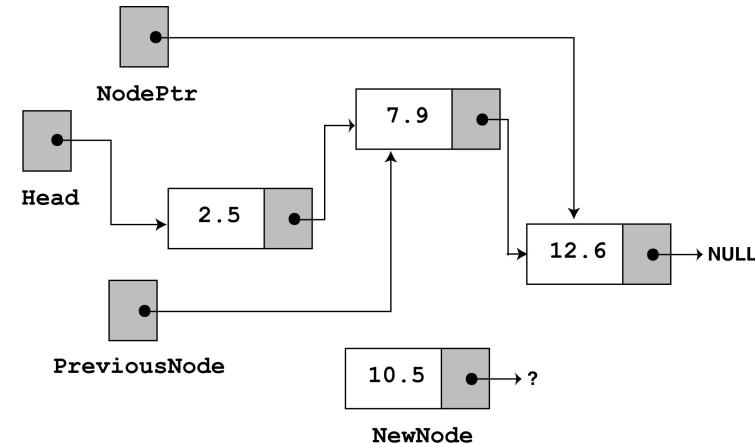
```
previousNode=nodePtr;  
nodePtr=nodePtr->next;
```





The New Node Pointer find the next node to insert the node

```
previousNode->next=newNode;  
newnode->next=nodePtr;
```



Contd...



```
void insertBetween(int value, int loc)
{
    struct Node *newNode, *prev_ptr, *cur_ptr;
    newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = value;
    cur_ptr=head;
    if(head == NULL)
    {
        newNode->next = NULL;
        head = newNode;
    }
    else
    {
        for(i=1;i<loc;i++)
        {
            prev_ptr=cur_ptr;
            cur_ptr = cur_ptr->next;
        }
        prev_ptr->next = newNode;
        newNode->next = cur_ptr;
    }
    printf("\nOne node inserted!!!\n");
}
```



Deleting a node

In a linked list, deleting a node has the following possible cases.

1. Delete at the front
2. Delete at the back
3. Delete in the middle

Deleting a node from a SLL



- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header



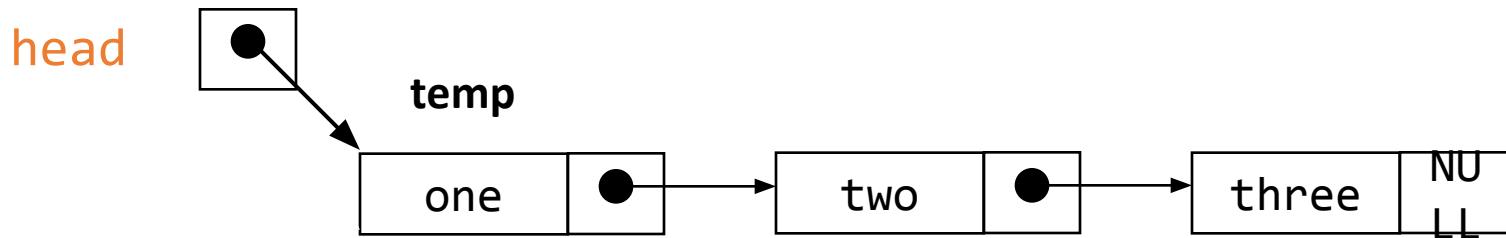
Deleting from Beginning of the list

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **head**.
- **Step 4:** Check whether list is having only one node (**temp → next == NULL**)
- **Step 5:** If it is **TRUE** then set **head = NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6:** If it is **FALSE** then set **head = temp → next**, and delete **temp**.



Deleting an element from a SLL

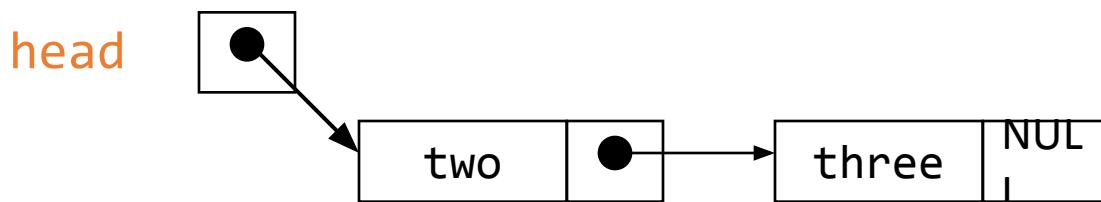
- To delete the first element, change the link in the header



- Assign **head** to a **temp**. **temp=head;**
- If the **temp** is not **NULL**, check whether the data to be deleted is **head**.
- If true then delete the **head** node and make the next node as **head**.

head=temp->next;

free(temp);



Contd...



```
void removeBeginning()
{
    if(head == NULL)
        printf("\n\nList is Empty!!!!");
    else
    {
        struct Node *temp = head;
        if(head->next == NULL)
        {
            head = NULL;
            free(temp);
        }
        else
        {
            head = temp->next;
            free(temp);
            printf("\nOne node deleted!!!\n\n");
        }
    }
}
```

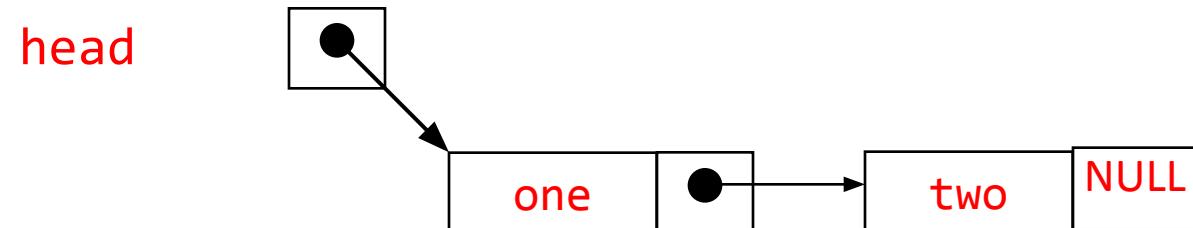
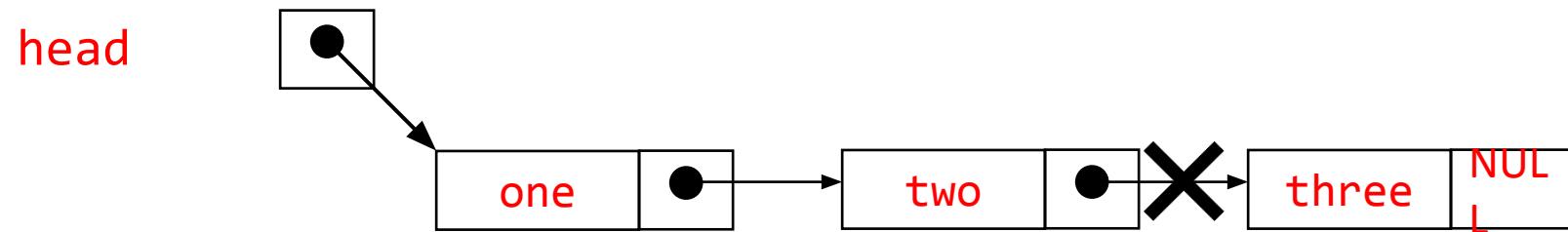
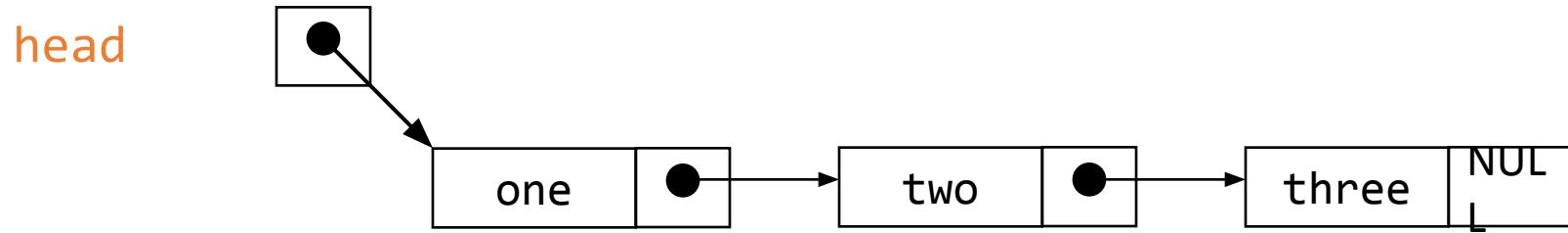


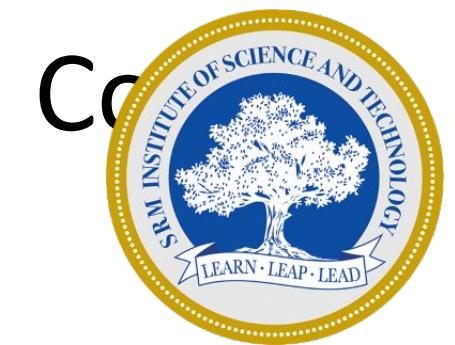
Deleting from End of the list

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Check whether list has only one Node (**temp1 → next == NULL**)
- **Step 5:** If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6:** If it is **FALSE**. Then, set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next == NULL**)
- **Step 7:** Finally, Set **temp2 → next = NULL** and delete **temp1**.



Deleting at the end





```
void removeEnd()
{
    if(head == NULL)
    {
        printf("\nList is Empty!!!\n");
    }
    else
    {
        struct Node *temp1 = head,*temp2;
        if(head->next == NULL)
            head = NULL;
        else
        {
            while(temp1->next != NULL)
            {
                temp2 = temp1;
                temp1 = temp1->next;
            }
            temp2->next = NULL;
        }
        free(temp1);
        printf("\nOne node deleted!!!\n\n"); } }
```



Deleting a Specific Node from the list

- **Step 1:** Check whether list is **Empty (head == NULL)**
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5:** If it is reached to the last node then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.
- **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not



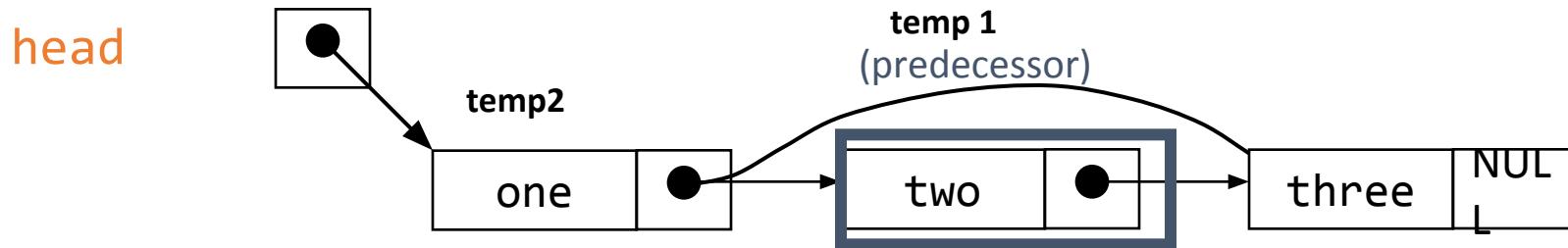
Contd....

- **Step 7:** If list has only one node and that is the node to be deleted, then set **head = NULL** and delete **temp1 (free(temp1))**.
- **Step 8:** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9:** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
- **Step 10:** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
- **Step 11:** If **temp1** is last node then set **temp2 → next = NULL** and delete **temp1 (free(temp1))**.
- **Step 12:** If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1 (free(temp1))**.



Deleting an element from a SLL

- To delete some other element, change the link in its predecessor



□ Use a Node Pointer to trace to the current position

□ Also to store the value of the next address

□ Use a Previous Node Pointer to trace to the current position

```
temp2=temp1;  
temp1=temp1->next;
```

- If the deleted element is found then

```
temp2->next=temp1->next;  
free( temp1);
```

Contd...



```
void removeSpecific(int delValue)
{
    struct Node *temp1 = head, *temp2;
    while(temp1->data != delValue)
    {
        if(temp1 -> next == NULL) {
            printf("\nGiven node not found in the list!!!!");
        }
        temp2 = temp1;
        temp1 = temp1 -> next;
    }
    temp2 -> next = temp1 -> next;
    free(temp1);
    printf("\nOne node deleted!!!!\n\n");
}
```



Displaying a Single Linked List

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!!**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **head**.
- **Step 4:** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5:** Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).



```
void display()
{
    if(head == NULL)
    {
        printf("\nList is Empty\n");
    }
    else
    {
        struct Node *temp = head;
        printf("\n\nList elements are - \n");
        while(temp->next != NULL)
        {
            printf("%d -->",temp->data);
            temp = temp->next;
        }
        printf("%d -->NULL",temp->data);
    }
}
```



References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-6



1. Applications of Linked List
2. Polynomial Arithmetic



Applications of linked list

- Used to implement **stacks and queues**.
- Used to implement the **Adjacency list representation** of graphs.
- Used to perform **dynamic memory allocation**.
- Used to **Maintain directory of names**.
- Used to **perform arithmetic operations** on long integers.
- Used to **manipulate polynomials** to store constants.
- Used to **represent sparse matrices**.



Polynomials

- Representing Polynomials As Singly Linked Lists
 - The manipulation of symbolic polynomials, has a classic example of list processing.
 - In general, we want to represent the polynomial:
$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$
 - Where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that
$$e_{m-1} > e_{m-2} > \dots > e_1 > e_0 \geq 0 .$$
 - We will represent each term as a node containing coefficient and exponent fields, as well as a pointer to the next term.



Polynomials

- Assuming that the coefficients are integers, the type declarations are:

```
struct link{  
    int coeff;  
    int pow;  
    struct link *next;  
};
```

- Draw *poly_nodes* as:

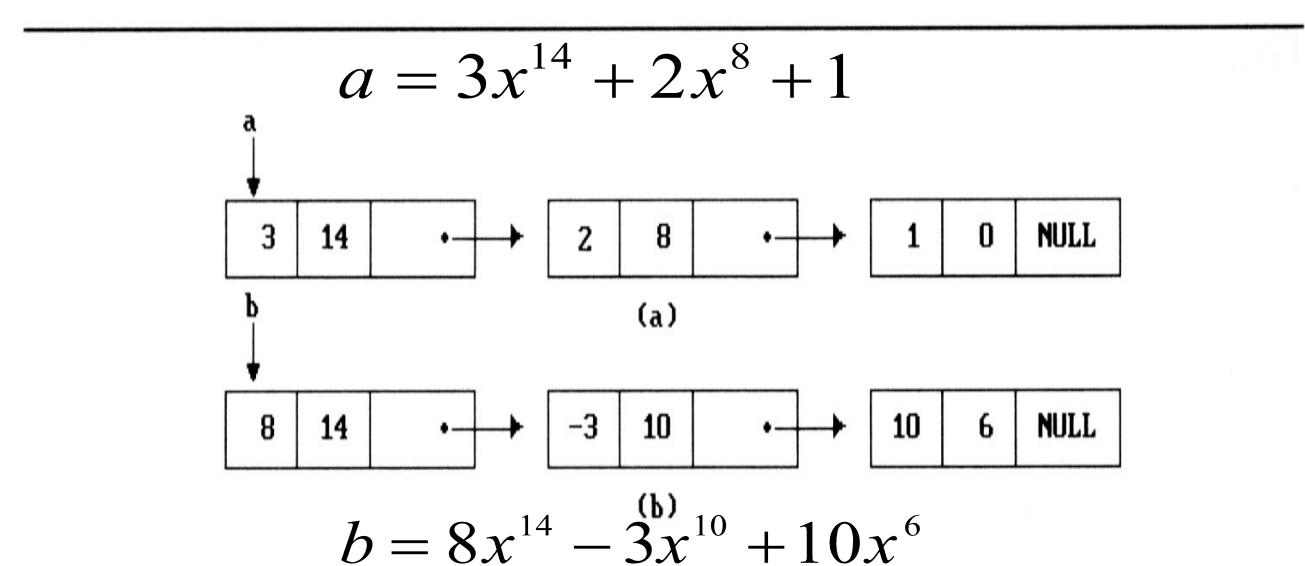


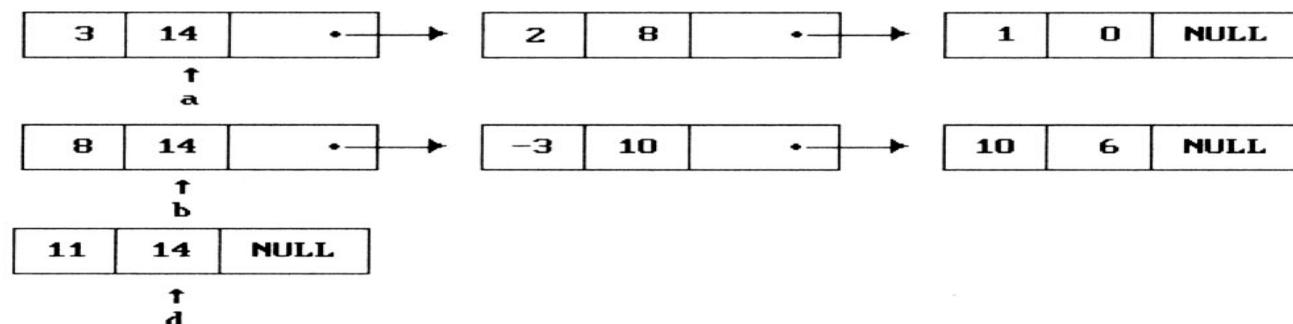
Figure 4.11: Polynomial representation

coef	expon	link
------	-------	------

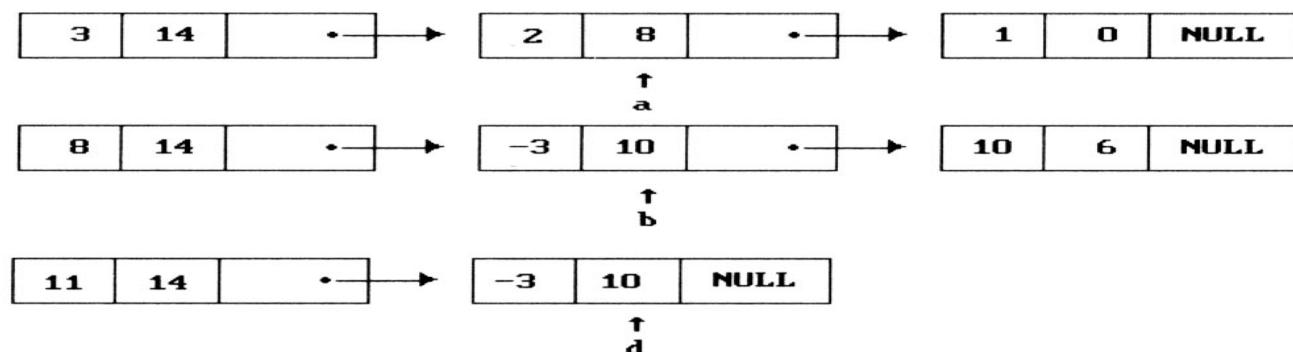


Polynomial Additions

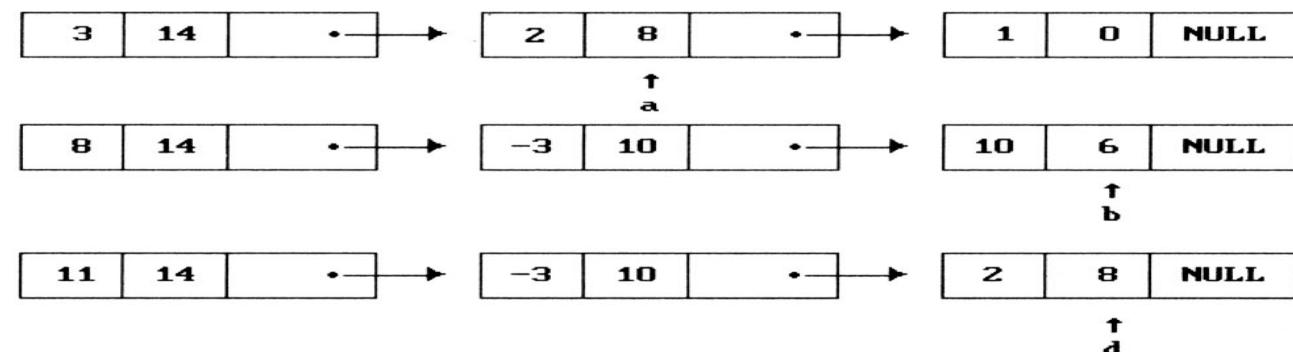
- Adding Polynomials
 - To add two polynomials, we examine their terms starting at the nodes pointed to by a and b .
 - If the exponents of the two terms are equal
 1. add the two coefficients
 2. create a new term for the result.
 - If the exponent of the current term in a is less than b
 1. create a duplicate term of b
 2. attach this term to the result, called d
 3. advance the pointer to the next term in b .
 - We take a similar action on a if $a->expon > b->expon$.
 - Figure 4.12 generating the first three term of $d = a+b$ (next page)



(a) $a \rightarrow \text{expon} == b \rightarrow \text{expon}$



(b) $a \rightarrow \text{expon} < b \rightarrow \text{expon}$



(c) $a \rightarrow \text{expon} > b \rightarrow \text{expon}$

Figure 4.12: Generating the first three terms of $d = a + b$



Polynomial - Creation

```
void create(struct link *node)
{
    char ch;
    do
    {
        printf("\n enter coeff:");
        scanf("%d",&node->coeff);
        printf("\n enter power:");
        scanf("%d",&node->pow);
        node=(struct link*)malloc(sizeof(struct link));
        node->next=NULL;
        printf("\n continue(y/n):");
        ch=getch();
    }
    while(ch=='y' || ch=='Y');
}
```



Polynomial - Display

```
void show(struct link *node)
{
    while(node->next!=NULL)
    {
        printf("%dx^%d",node->coeff,node->pow);
        node=node->next;
        if(node->next!=NULL)
            printf("+");
    }
}
```



Polynomial Add

```
void polyadd(struct link *poly1,struct link
*poly2,struct link *poly)
{
    while(poly1->next && poly2->next)
    {
        if(poly1->pow>poly2->pow)
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff;
            poly1=poly1->next;
        }
        else if(poly1->pow<poly2->pow)
        {
            poly->pow=poly2->pow;
            poly->coeff=poly2->coeff;
            poly2=poly2->next;
        }
        else
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff+poly2->coeff;
            poly1=poly1->next;
            poly2=poly2->next;
        }
        poly->next=(struct link *)malloc(sizeof(struct
link));
        poly=poly->next;
        poly->next=NULL;
    }
}
```

Contd...



```
while(poly1->next || poly2->next)
{
    if(poly1->next)
    {
        poly->pow=poly1->pow;
        poly->coeff=poly1->coeff;
        poly1=poly1->next;
    }
    if(poly2->next)
    {
        poly->pow=poly2->pow;
        poly->coeff=poly2->coeff;
        poly2=poly2->next;
    }
    poly->next=(struct link *)malloc(sizeof(struct link));
    poly=poly->next;
    poly->next=NULL;
}
}
```



References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-7



- 1. Cursor Based Implementation – Methodology**
- 2. Cursor Based Implementation**



Cursor Implementation

- Some programming languages doesn't support pointers.
- But we need linked list representation to store data.
- Solution:
 - **Cursor implementation** – implementing linked list on an array.
- Main idea is:
 - Convert a linked list based implementation to an array based implementation.
 - Instead of pointers, array index could be used to keep track of node.
 - Convert the node structures (collection of data and next pointer) to a global array of structures.
 - Need to find a way to perform dynamic memory allocation performed using malloc and free functions.



Declarations for cursor implementation of linked lists

```
typedef unsigned int node_ptr;

struct node
{
    element_type element;
    node_ptr next;
};

typedef node_ptr LIST;
typedef node_ptr position;
struct node CURSOR_SPACE[ SPACE_SIZE ];
```



Initializing Cursor Space

- `Cursor_space[list] = 0;`
- To check whether `cursor_space` is empty:

```
int is_empty( LIST L ) /* using a header node */
{
    return( CURSOR_SPACE[L].next == 0
}
```

- To check whether `p` is last in a linked list

```
int is_last( position p, LIST L) /* using a header node */
{
    return( CURSOR_SPACE[p].next == 0
}
```

An initialized cursor space



Slot	Element	Next
0	?	1
1	?	2
2	?	3
3	?	4
4	?	5
5	?	6
6	?	7
7	?	8
8	?	9
9	?	10
10	?	0



Insert Routine

```
/* Insert (after legal position p); */
/* header implementation assumed */

void insert( element_type x, LIST L, position p )
{
    position tmp_cell;
    /*1*/ tmp_cell = cursor_alloc( )
    /*2*/ if( tmp_cell ==0 )
    /*3*/ fatal_error("Out of space!!!");
    else
    {
        /*4*/ CURSOR_SPACE[tmp_cell].element = x;
        /*5*/ CURSOR_SPACE[tmp_cell].next = CURSOR_SPACE[p].next;
        /*6*/ CURSOR_SPACE[p].next = tmp_cell;
    }
}
```



Search Routine – Cursor Implementation

```
position find( element_type x, LIST L) /* using a header node */
{
    position p;
    /*1*/ p = CURSOR_SPACE[L].next;
    /*2*/ while( p && CURSOR_SPACE[p].element != x )
    /*3*/ p = CURSOR_SPACE[p].next;
    /*4*/ return p;
}
```



Cursor implementation of linked lists - Example

Slot	Element	Next
0	-	6
1	b	9
2	f	0
3	header	7
4	-	0
5	header	10
6	-	4
7	c	8
8	d	2
9	e	0
10	a	1

- If L= 5, then L represents list (A, B, E)
- If M= 3, then M represents list (C, D, F)



Deletion Routine

```
void delete( element_type x, LIST L )
{
    position p, tmp_cell;
    p = find_previous( x, L );

    if( !is_last( p, L ) )
    {
        tmp_cell = CURSOR_SPACE[p].next;
        CURSOR_SPACE[p].next = CURSOR_SPACE[tmp_cell].next;
        cursor_free( tmp_cell );
    }
}
```



References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-8

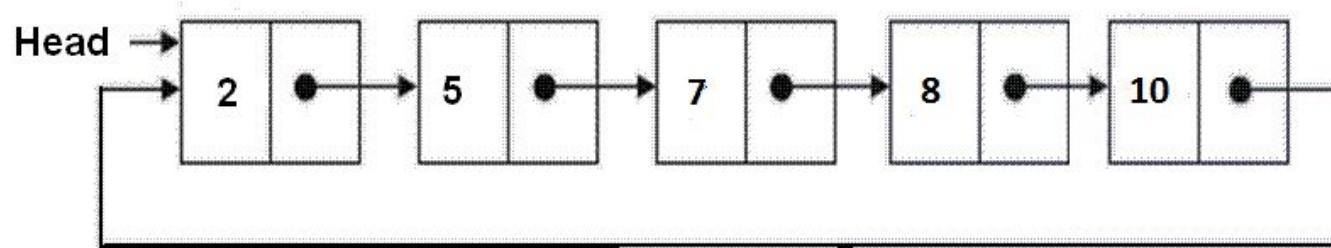


- 1. Circular Linked List**
- 2. Circular Linked List - Implementation**



Circular Linked List

- ***Circular linked list*** is a linked list where all nodes are connected to form a circle.
- There is no NULL at the end.
- A circular linked list can be a singly circular linked list or doubly circular linked list.





Advantages of Circular Linked Lists

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- Useful for implementation of queue.
- For example, when multiple applications are running on a PC



OPERATIONS ON CIRCULARLY LINKED LIST

- Insertion
- Deletion
- Display

Circular linked list – Creation

```
temp=head;  
temp->next = new;  
new -> next = head;
```

Insertion in a Circular Linked List



1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list



Inserting At Beginning of the list

Steps to **insert a new node at beginning** of the circular linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head == NULL**)

Step 3 - If it is **Empty** then, set **head = newNode** and **newNode→next = head** .

Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next == head**').

Step 6 - Set '**newNode → next =head**', '**head = newNode**' and '**temp → next = head**'.



Inserting At the End of the list

- Steps to insert a new node at end of the circular linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head == NULL**).

Step 3 - If it is **Empty** then, set **head = newNode** and **newNode → next = head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next == head**).

Step 6 - Set **temp → next = newNode** and **newNode → next = head**.

Inserting At Specific location in the list (After a Node)



Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head == NULL**)

Step 3 - If it is **Empty** then, set **head = newNode** and **newNode → next = head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp → data** is equal to **location**, here **location** is the node value after which we want to insert the **newNode**).

Step 6 - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.

Step 7 - If **temp** is reached to the exact node after which we want to insert the **newNode** then check whether it is last node (**temp → next == head**).

Step 8 - If **temp** is last node then set **temp → next = newNode** and **newNode → next = head**.

Step 8 - If **temp** is not last node then set **newNode → next = temp → next** and **temp → next = newNode**.



CLL – Insertion (ROUTINE)

Insertion at first

```
new->next = head
```

```
head =new;
```

```
temp->next= head;
```

Insertion at middle

```
n1->next = new;
```

```
new->next = n2;
```

```
n2->next = head;
```

Insertion at last

```
n2->next=new;
```

```
new->next = head;
```



CLL – Deletion Operation

- In a circular linked list, the deletion operation can be performed in three ways those are as follows...
 1. Deleting from Beginning of the list
 2. Deleting from End of the list
 3. Deleting a Specific Node

DELETING FROM BEGINNING OF THE LIST



- The following steps to delete a node from beginning of the circular linked list...

Step 1 - Check whether list is **Empty** (**head == NULL**)

Step 2 - If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.

Step 4 - Check whether list is having only one node (**temp1 → next == head**)

Step 5 - If it is **TRUE** then set **head = NULL** and delete **temp1** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next == head**)

Step 7 - Then set **head = temp2 → next**, **temp1 → next = head** and delete **temp2**.



DELETING FROM END OF THE LIST

- The following steps to delete a node from end of the circular linked list...

Step 1 - Check whether list is **Empty** (**head == NULL**)

Step 2 - If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.

Step 4 - Check whether list has only one Node (**temp1 → next == head**)

Step 5 - If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)

Step 7 - Set **temp2 → next = head** and delete **temp1**.

DELETING A SPECIFIC NODE FROM THE LIST



- The following steps to delete a specific node from the circular linked list...

Step 1 - Check whether list is **Empty** (**head == NULL**)

Step 2 - If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.

Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5 - If it is reached to the last node then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next == head**)

Step 7 - If list has only one node and that is the node to be deleted then set **head = NULL** and delete **temp1** (**free(temp1)**).

Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).

Step 9 - If **temp1** is the first node then set **temp2 = head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next = head** and delete **temp1**.

Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).

Step 11 - If **temp1** is last node then set **temp2 → next = head** and delete **temp1** (**free(temp1)**).

Step 12 - If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1** (**free(temp1)**).



CLL - Deletion Operation

```
//delete first item
struct node * deleteFirst() {
    //save reference to first link
    struct node *tempLink = head;
    if(head->next == head){
        head = NULL;
        return tempLink;
    }
    //mark next to first link as first
    head = head->next;
    //return the deleted link
    return tempLink;
}
```



Deleting first node from Singly Circular Linked List

- Given a Circular Linked List. The task is to write programs to delete nodes from this list present at:

- First position.
- Last Position.
- At any given position i.

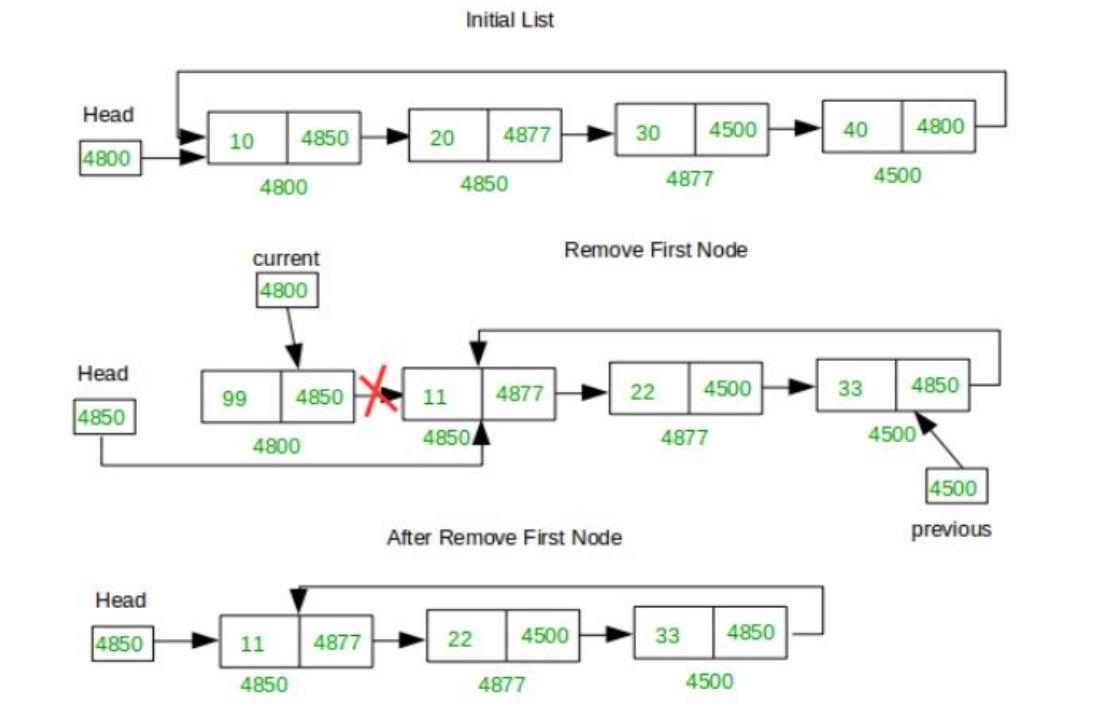
Examples:

Input : 99->11->22->33->44->55->66

Output : 11->22->33->44->55->66

Input : 11->22->33->44->55->66

Output : 22->33->44->55->66





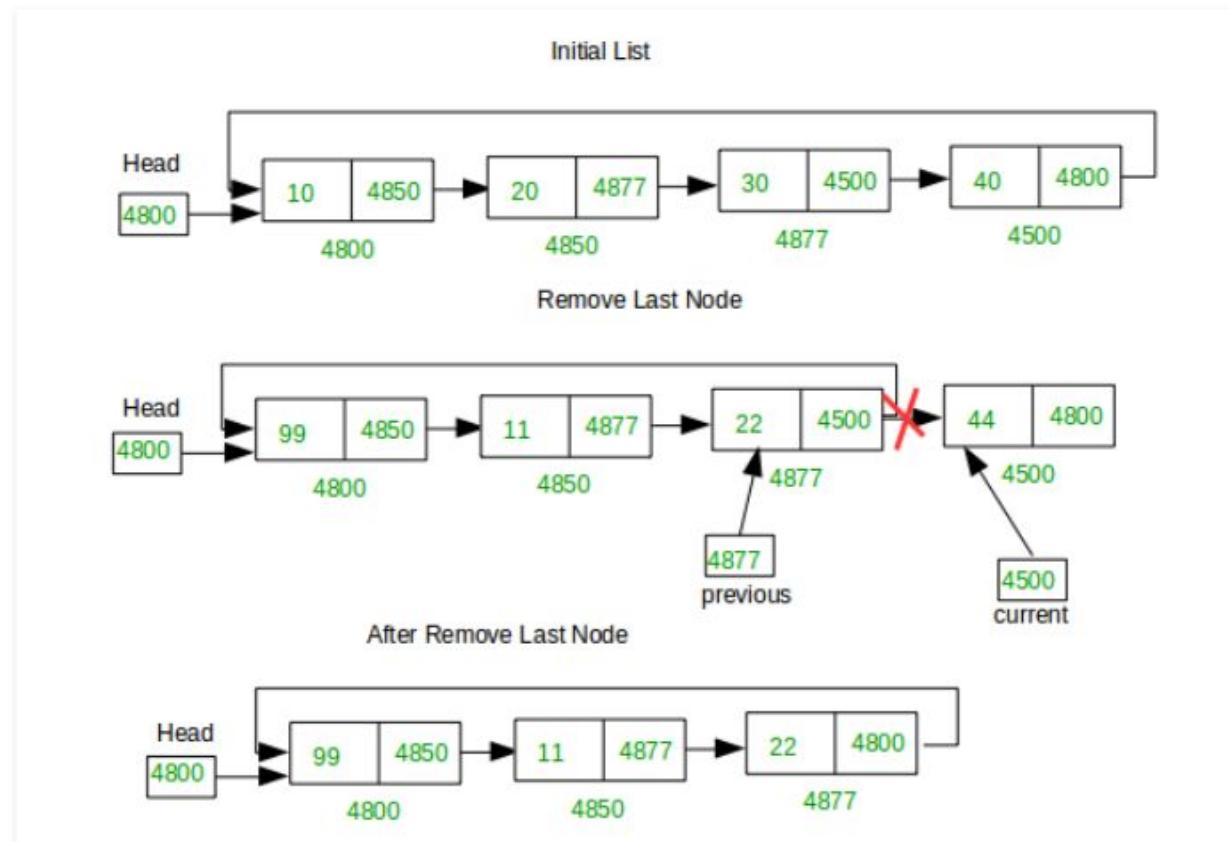
Deleting the last node of the Circular Linked List

Input : 99->11->22->33->44->55->66

Output : 99->11->22->33->44->55

Input : 99->11->22->33->44->55

Output : 99->11->22->33->44





Deleting nodes at given index in the Circular linked list

Input : 99->11->22->33->44->55->66

Index= 4

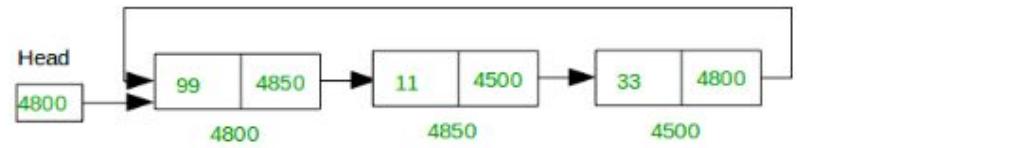
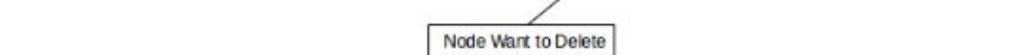
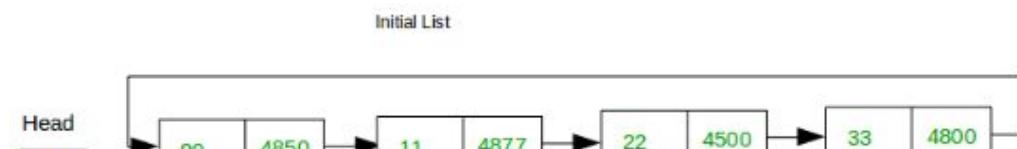
Output : 99->11->22->33->55->66

Input : 99->11->22->33->44->55->66

Index= 2

Output : 99->11->33->44->55->66

Note: 0-based indexing is considered for the list.





Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
 - **Dynamic:** a linked list can easily grow and shrink in size.
 - We don't need to know how many nodes will be in the list. They are created in memory as needed.
 - In contrast, the size of a C++ array is fixed at compilation time.
 - **Easy and fast insertions and deletions**
 - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
 - With a linked list, no need to move other nodes. Only need to reset some pointers.



References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-11



1. Applications of Circular List -Joseph Problem
2. Doubly Linked List

Josephus Circle using circular linked list



- There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number m which indicates that m-1 persons are skipped and m-th person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

Examples :

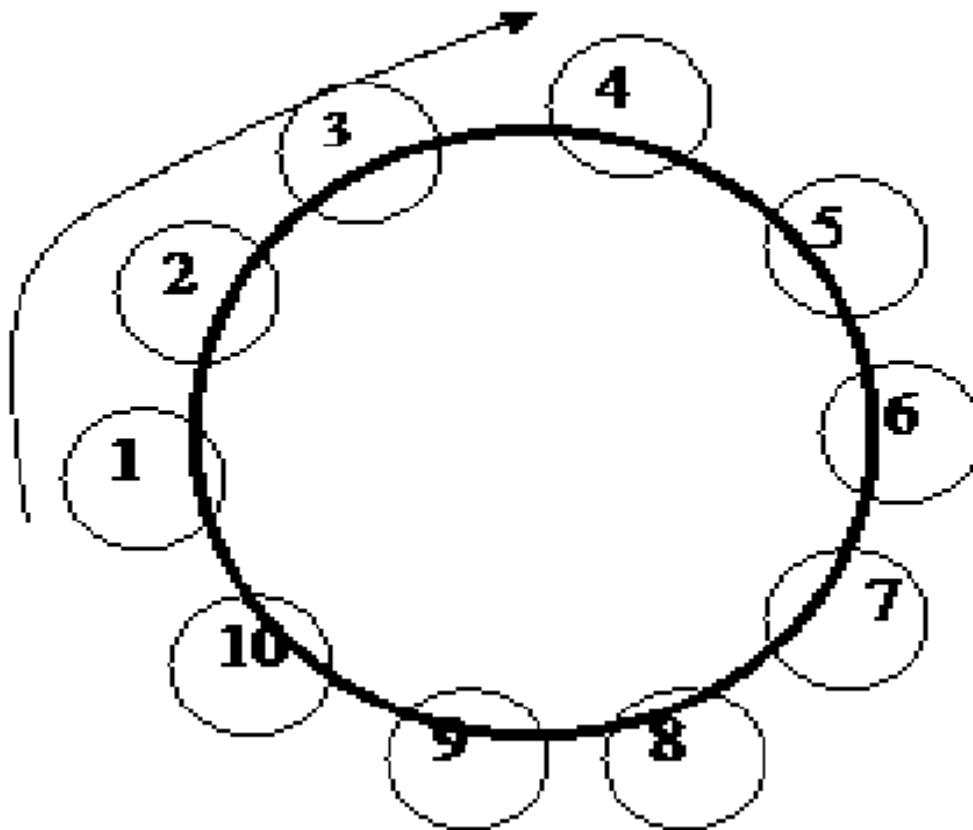
```
Input : Length of circle : n = 4  
        Count to choose next : m = 2  
Output : 1
```

```
Input : n = 5  
        m = 3  
Output : 4
```

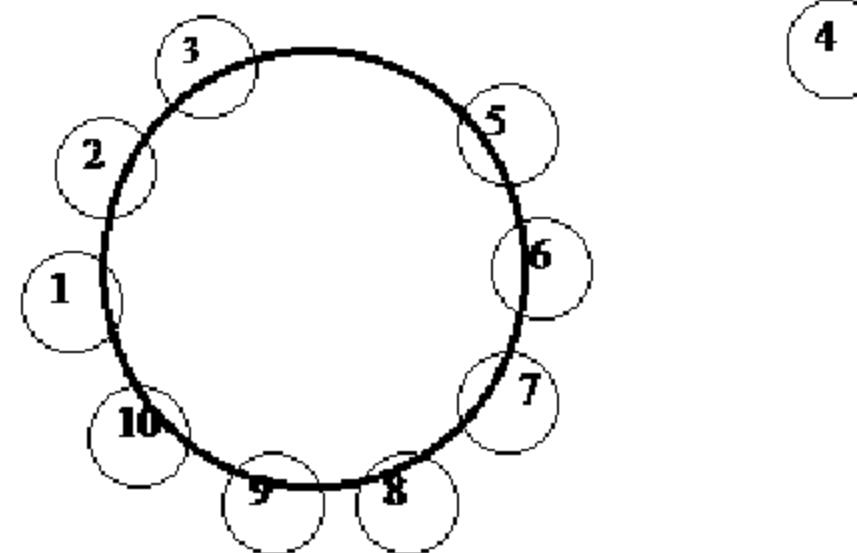


Example

N=10, M=3

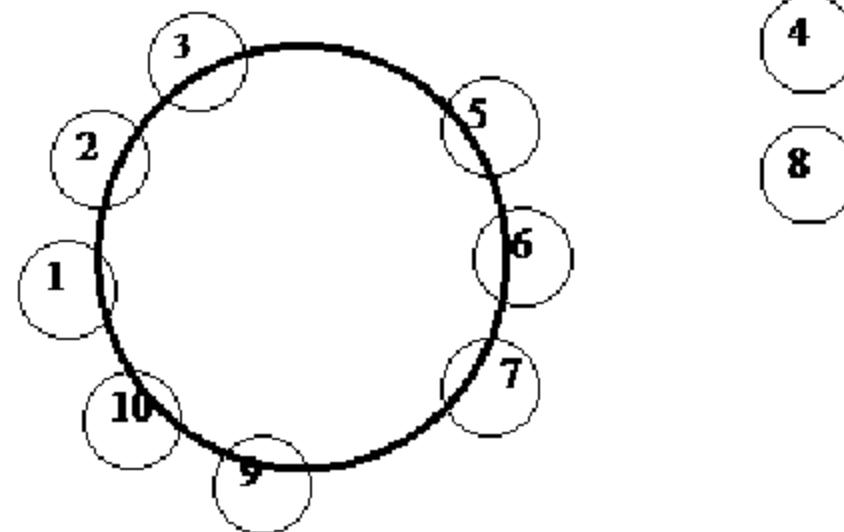


N=10, M=3



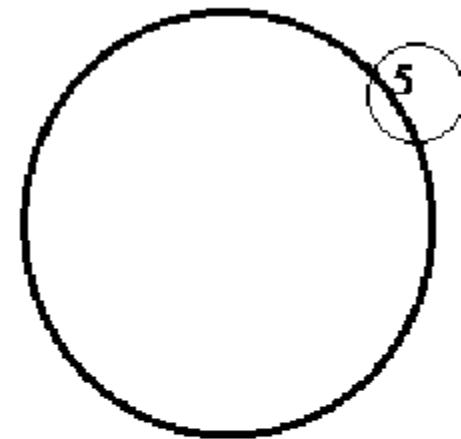
Eliminated

N=10, M=3





N=10, M=3



Eliminated

- 4
- 8
- 2
- 7
- 3
- 10**
- 9**
- 1**
- 6**

Josephus Problem - Snippet



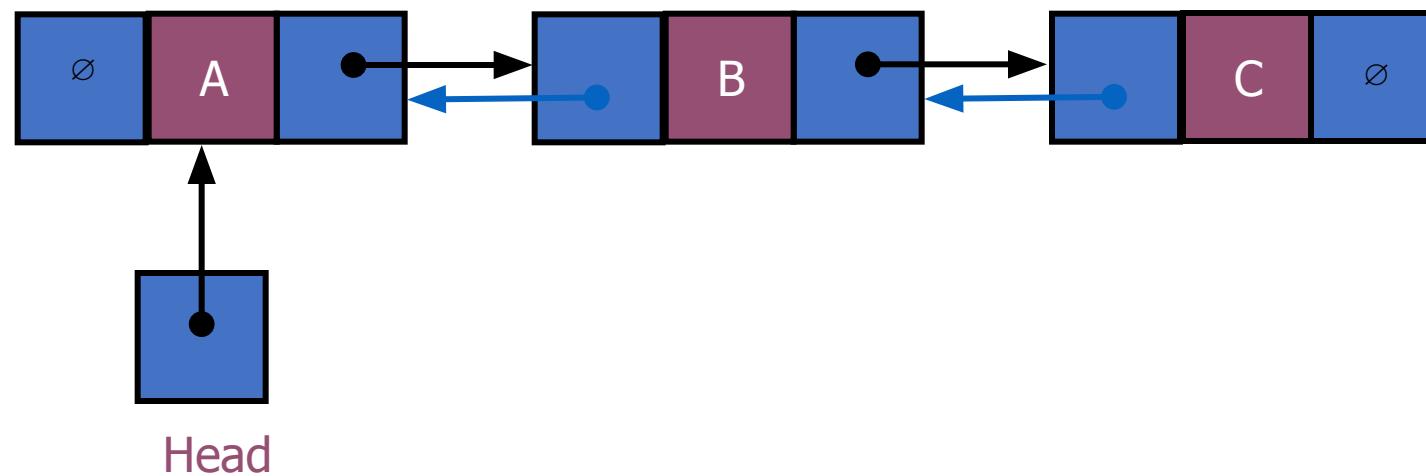
```
int josephus(int m,node *head)
{
    node *f;
    int c=1;
    while(head->next!=head)
    {
        c=1;
        while(c!=m)
        {
            f=head;
            head=head->next;
            c++;
        }
        f->next=head->next;
        printf("%d->",head->data); //sequence in which nodes getting deleted
        head=f->next;
    }
    printf("\n");
    printf("Winner is:%d\n",head->data);
    return;
}
```



Doubly Linked Lists

- *Doubly linked lists*

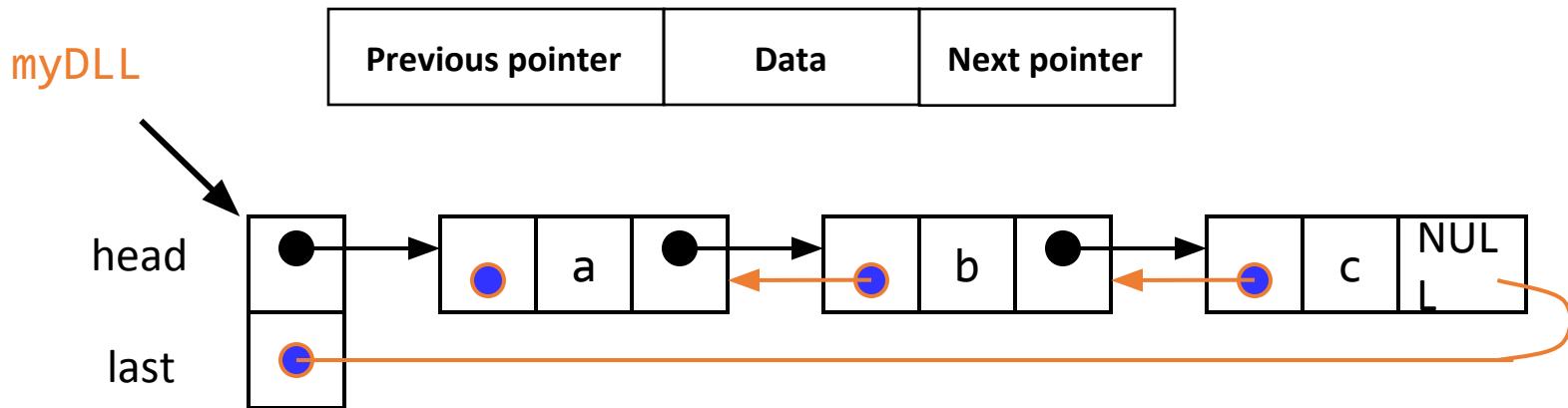
- Each node points to not only successor but the predecessor
- There are two NULL: at the first and last nodes in the list
- Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists **backwards**





Doubly-linked lists

- Here is a doubly-linked list (DLL):

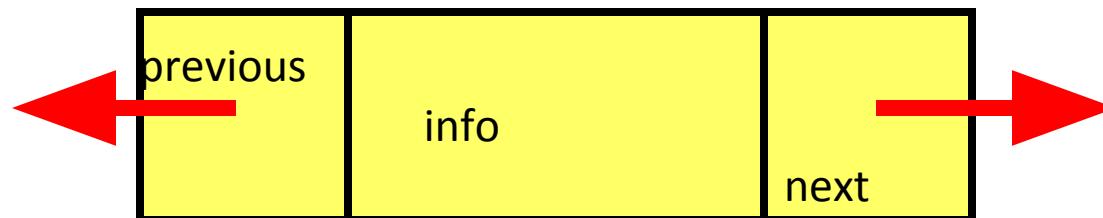


- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)



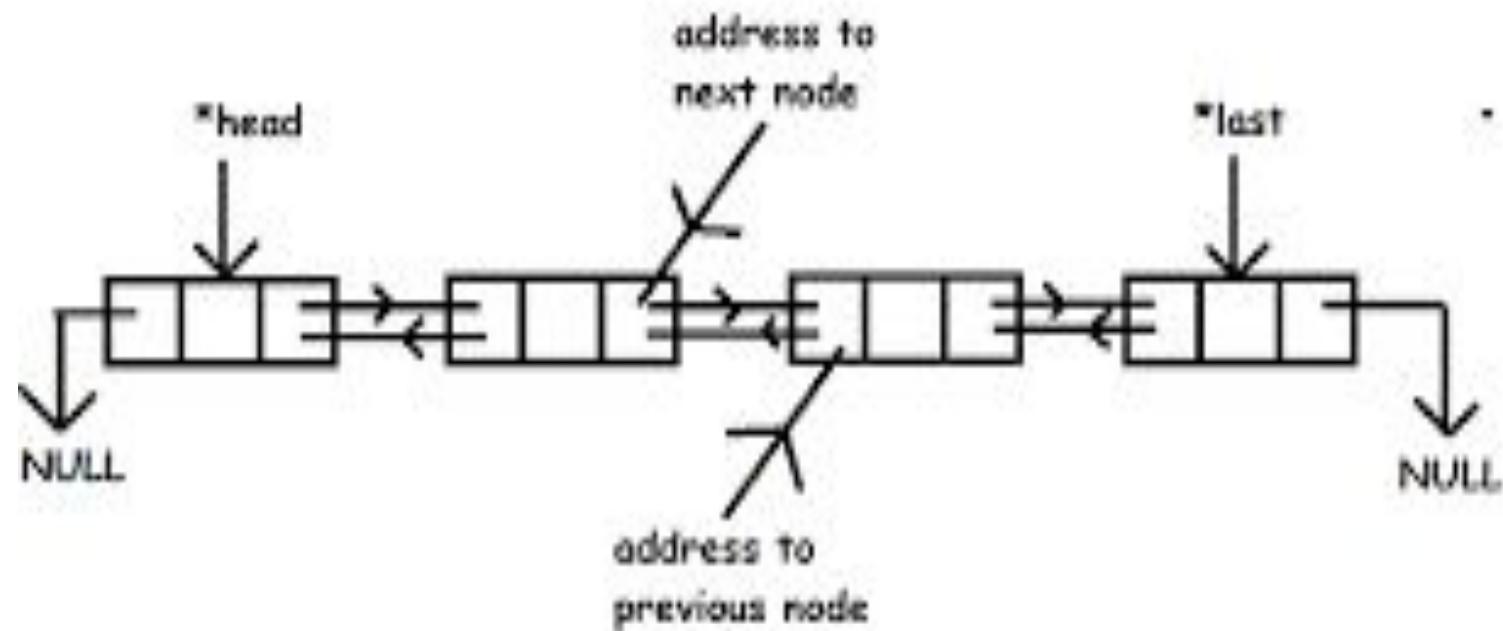
Node data

- info: the user's data
- next, previous: the address of the next and previous node in the list





DLL





Advantages and Disadvantages of Doubly Linked List

- **Advantages:**

1. We can traverse in both directions i.e. from starting to end and as well as from end to starting.
2. It is easy to reverse the linked list.
3. If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

Disadvantages:

1. It requires more space per node because one extra field is required for pointer to previous node.
2. Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list.



References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-12



Doubly Linked List Insertion



Node Creation

```
void create()
{
    int data;

    temp = (struct node *)malloc(sizeof(struct node));
    temp->prev = NULL;
    temp->next = NULL;
    printf("\n Enter value to node : ");
    scanf("%d", &data);
    temp->n = data;
    count++;
}
```

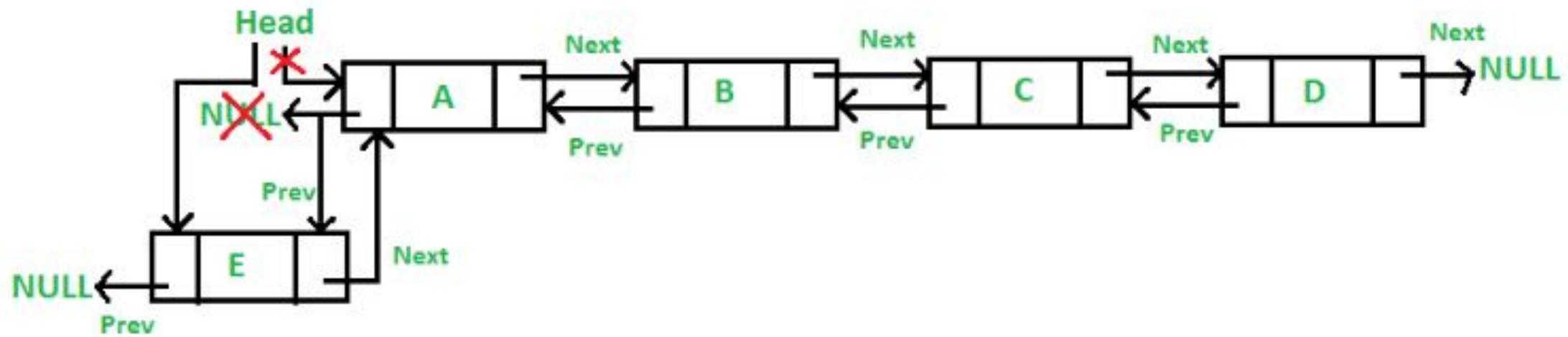
Insert at beginning



```
void insert_begning(int value)
{
    struct node *newNode, *temp;
    newNode= (struct node *)malloc(sizeof(struct node));
    newNode->data=value;
    if(head==NULL)
    {
        head=newNode;
        head->previous=NULL;
        head->next=NULL;
        last=head;
    }
    else
    {
        temp=newNode;
        temp->previous=NULL;
        temp->next=head;
        head->previous=temp;
        head=temp;
    }
}
```



At the front of the DLL



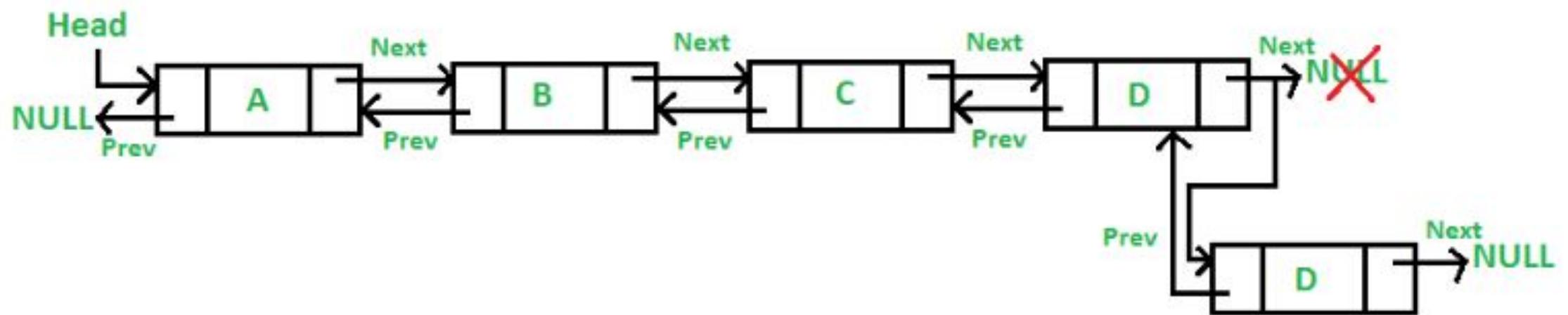


Insert at End

```
void insert_end(int value) {  
    struct node  
        *newNode, *last, *next,  
        *previous;  
  
    newNode=(struct node  
        *)malloc(sizeof(struct node));  
  
    newNode->data=value;  
    if(head==NULL)  
    { head=newNode;  
        head->previous=NULL;  
        head->next=NULL;  
        last=head;  
    }  
}
```

```
else  
{  
    last=head;  
    While (last->next!=NULL)  
    {  
        last=last->next;  
    }  
    last->next=newNode;  
    newNode->previous=last;  
    newNode->next=NULL;  
    last=newNode;  
}  
}
```

At the end



INSERT IN THE MIDDLE

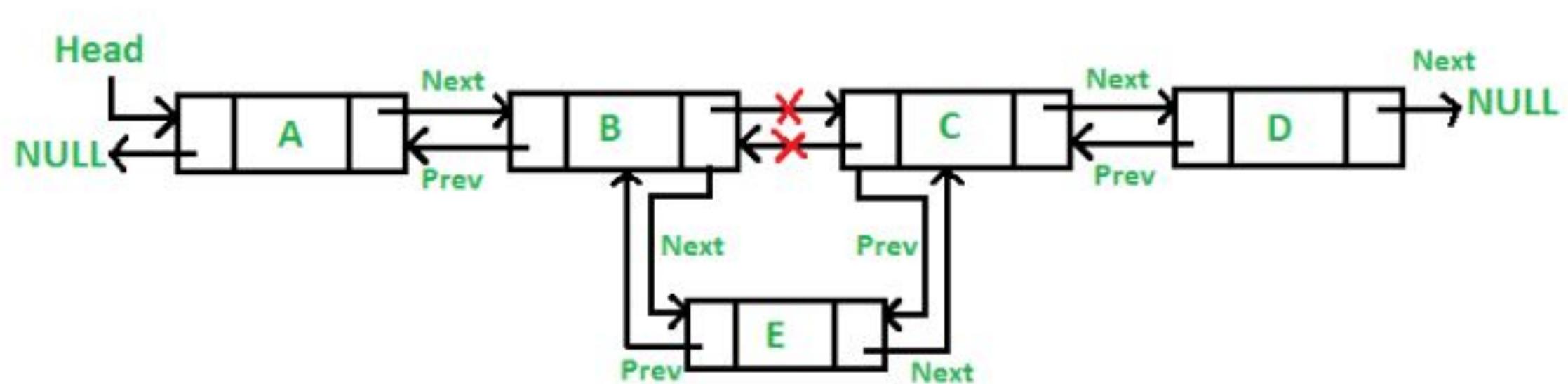
```
int insert_after(int value, int loc)
{
    struct node *temp,*newNode,*cur_ptr,
    prev_ptr, *next,*previous;
    newNode=(struct node
              *)malloc(sizeof(struct node));
    newNode->data=value;
    cur_ptr=head;
    if(head==NULL)
    {
        head=newNode;
        head->previous=NULL;
        head->next=NULL;
    }
    else
    {
```

```
        for(i=1;i<loc;i++)
        {
            prev_ptr=cur_ptr;
            cur_ptr = cur_ptr->next;
        }
        prev_ptr->next = newNode;
        newNode->previous = prev_ptr;
        newNode->next = cur_ptr;
        cur_ptr->previous=newNode;
    }
    last=head;
    while(last->next!=NULL)
    {
        last=last->next;
    }
}
```





At the middle





References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



18CSC201J-Data Structures and Algorithms

UNIT-II-ARRAYS AND LISTS

SESSION-13

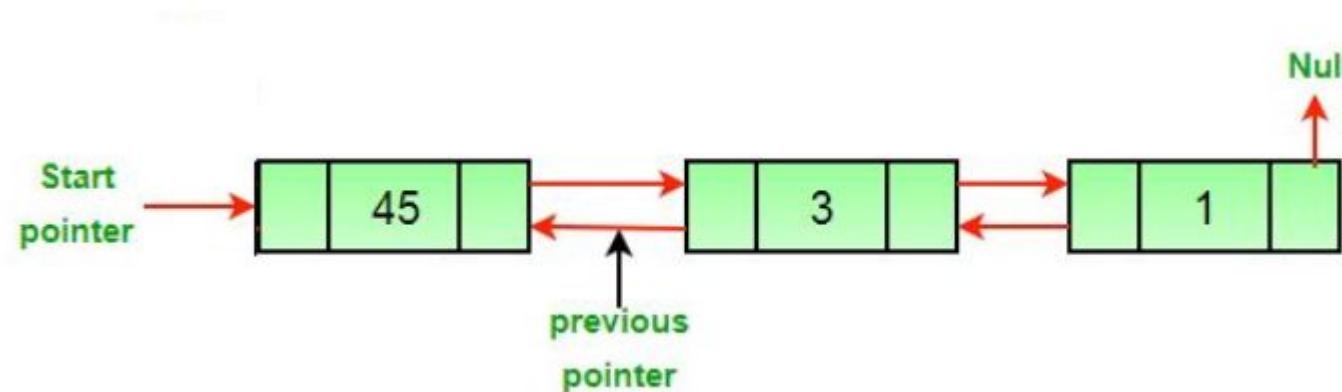
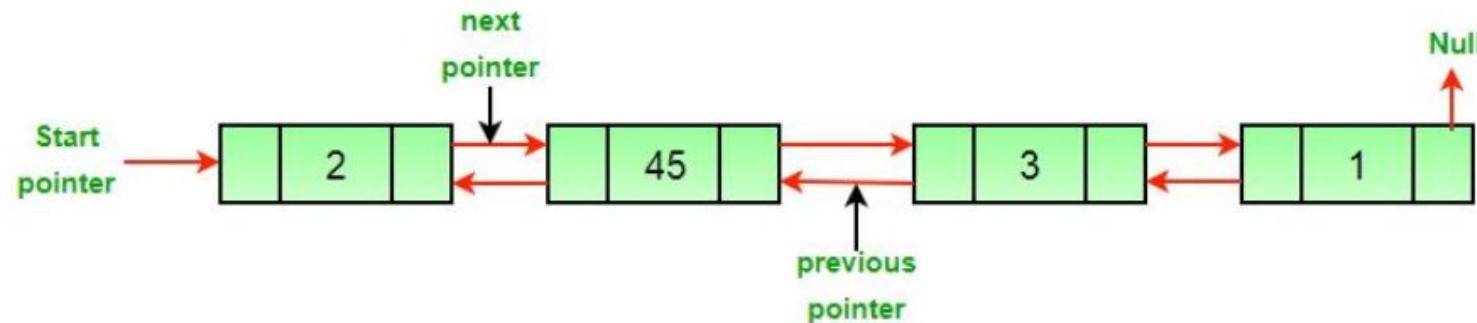


Doubly Linked List Deletion

Doubly Linked List Search



Deletion at the beginning



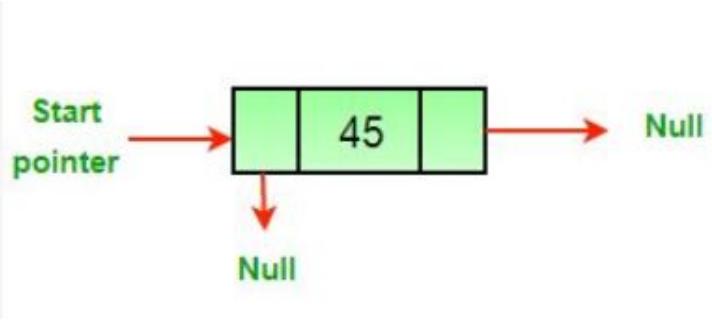


Delete at end

```
int delete_from_end()
{
    struct node *temp;
    temp=last;
    if(temp->previous==NULL)
    {
        free(temp);
        head=NULL;
        last=NULL;
        return 0;
    }
    printf("\nData deleted from list is %d \n",last->data);
    last=temp->previous;
    last->next=NULL;
    free(temp);
    return 0;
}
```



Deletion at the end



1. Let the node to be deleted be *del*.
2. If node to be deleted is head node, then change the head pointer to next current head.

```
if headnode == del then  
    headnode = del.nextNode
```

3. Set *next* of previous to *del*, if previous to *del* exists.

```
if del.nextNode != none  
    del.nextNode.previousNode = del.previousNode
```

4. Set *prev* of *next* to *del*, if *next* to *del* exists.

```
if del.previousNode != none  
    del.nextNode.nextNode = del.next
```

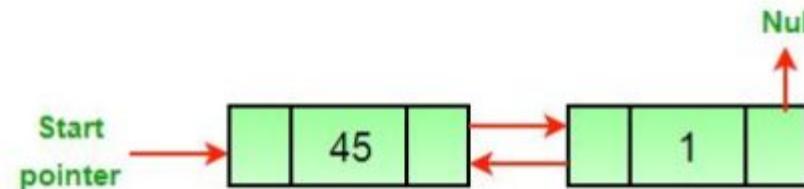
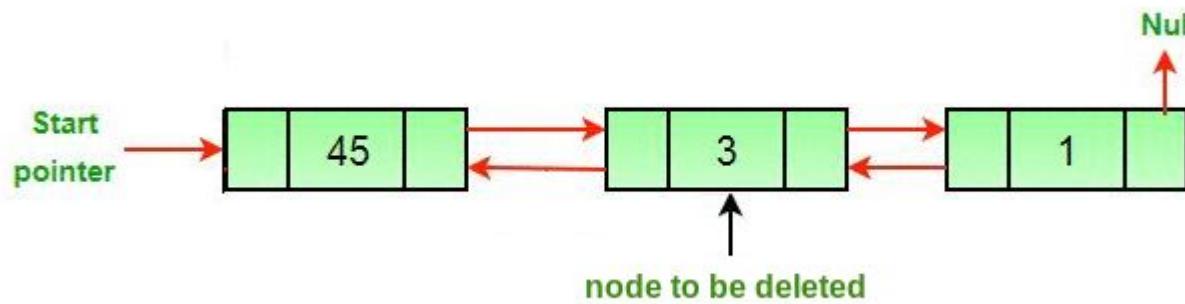
DELETE FROM MIDDLE



```
int delete_from_middle(int value)
{
    struct node *temp,*var,*t, *temp1;
    temp=head;  while(temp!=NULL)
    {
        if(temp->data == value)
        {
            if(temp->previous==NULL)      {
                free(temp);
                head=NULL;           last=NULL;
                return 0;
            }
            else
            {
                var=temp;
                temp=temp->next;
                temp1=temp->next;
            }
            printf("data deleted from list is
                    %d",value);
        }
        else
        {
            var->next=temp1;
            temp1->previous=var;
            free(temp);
            return 0;
        }
    }
}
```



Deletion of middle node



Search an Element in Doubly Circular Linked List

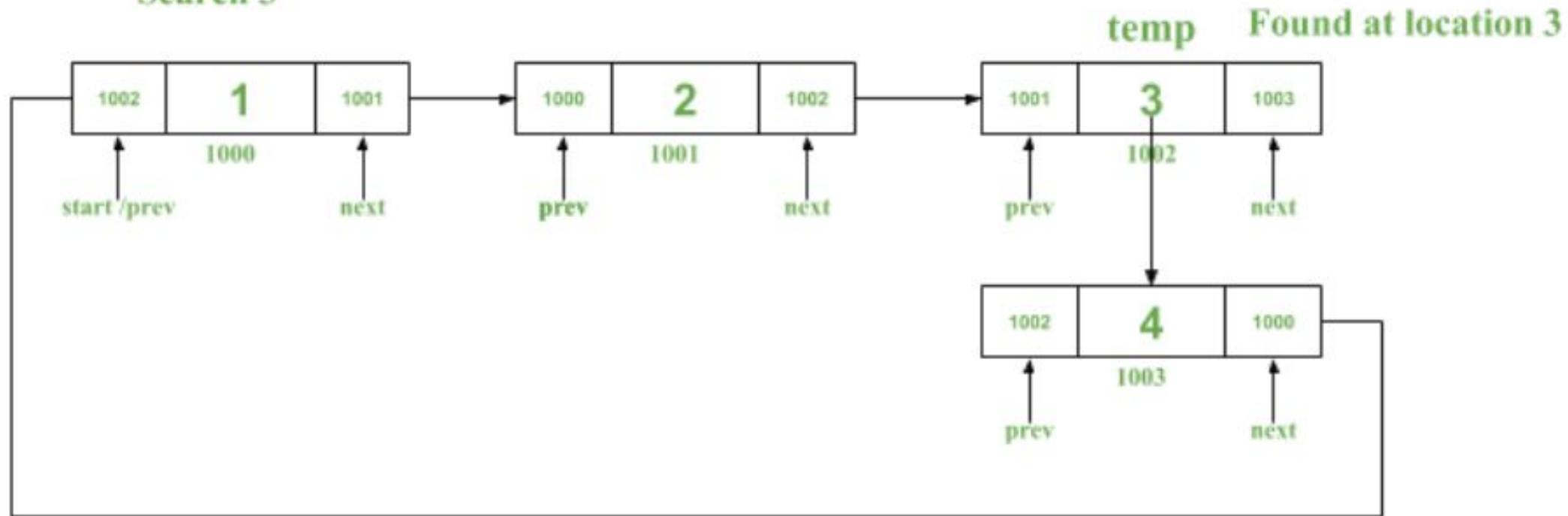


• Algorithm

- Declare a temp pointer, and initialize it to head of the list.
- Iterate the loop until temp reaches start address (last node in the list, as it is in a circular fashion), check for the n element, whether present or not.
- If it is present, raise a flag, increment count and break the loop.
- At the last, as the last node is not visited yet check for the n element if present do step 3.



Search 3





```
int searchList(struct Node* start, int search)
{
    // Declare the temp variable
    struct Node *temp = start;

    // Declare other control variable for the searching
    int count=0,flag=0,value;

    // If start is NULL return -1
    if(temp == NULL)
        return -1;

    else
    {
        // Move the temp pointer until temp->next doesn't move start address (Circular Fashion)
        while(temp->next != start)
        {
            count++;

            // If it is found raise the flag and break the loop
            if(temp->data == search)
            {
                flag = 1;
                count--;
                break;
            }

            // Increment temp pointer
            temp = temp->next;
        }
    }
}
```



References

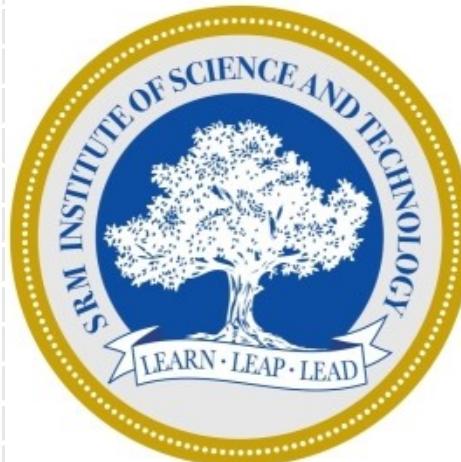
1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. <https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html>
4. http://www.btechsmartclass.com/data_structures
5. <https://www.geeksforgeeks.org/>



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE





SESSION 1





STACK ADT

- **Abstract Data Type (ADT)**
 - It is a mathematical model for **data types**
 - An **abstract data type** is **defined** by its behavior (semantics) from the point of view of a user, of the **data**, specifically in terms of possible values, possible operations on **data** of this **type**, and the behavior of these operations. Set of values and set of operations
- A stack is an ADT
 - It follows **Last In First Out (LIFO)** methodology to perform operations push, pop, etc.



What is STACK?



Diagram Reference : <http://www.firmcodes.com/write-a-c-program-to-implement-a-stack-using-an-array-and-linked-list/>

- Stack works on “Last in First out” or “First in Last Out”, principle.
- Plates placed one over another
- The plate at the top is removed first & the bottommost plate kept longest period of time.
- So, it follows LIFO/FILO order.



Stack Example

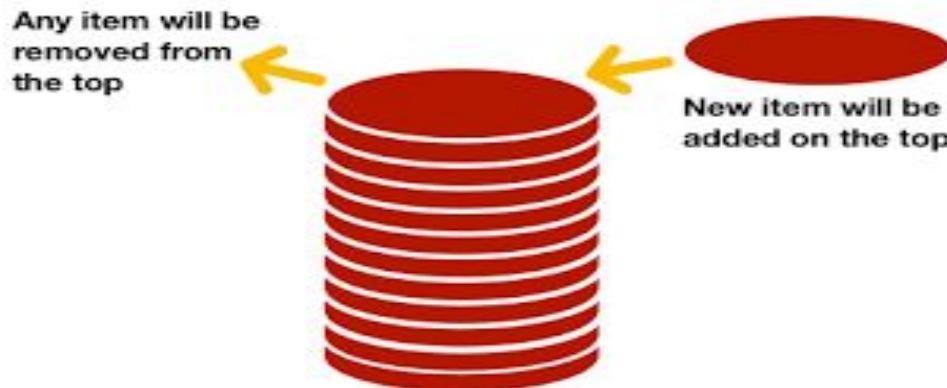


Diagram Reference: <https://www.codesdope.com/course/data-structures-stacks/>

- Always new items are added at top of the stack and also removed from the top of the stack only.
- Entry and Exit at same point



STACK – Data Structure

- Stack is a **Linear Data Structure**
- Follows a particular order to perform the operations.
- The order are either
 - LIFO(Last In First Out)
 - OR
 - FILO(First In Last Out).



STACK - Operations

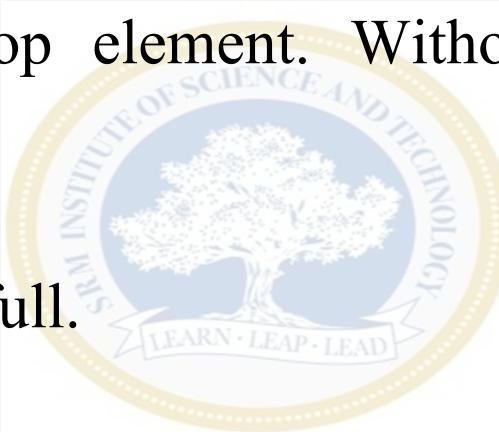
TWO PRIMARY OPERATIONS

- **push**
 - Pushing (storing) an element on the stack
 - If the stack is full, Overflow condition is enabled.
- **Pop**
 - Removing (accessing) an element from the stack.
 - The elements are popped in the decreasing order.
 - If the stack is empty, Underflow condition enabled.



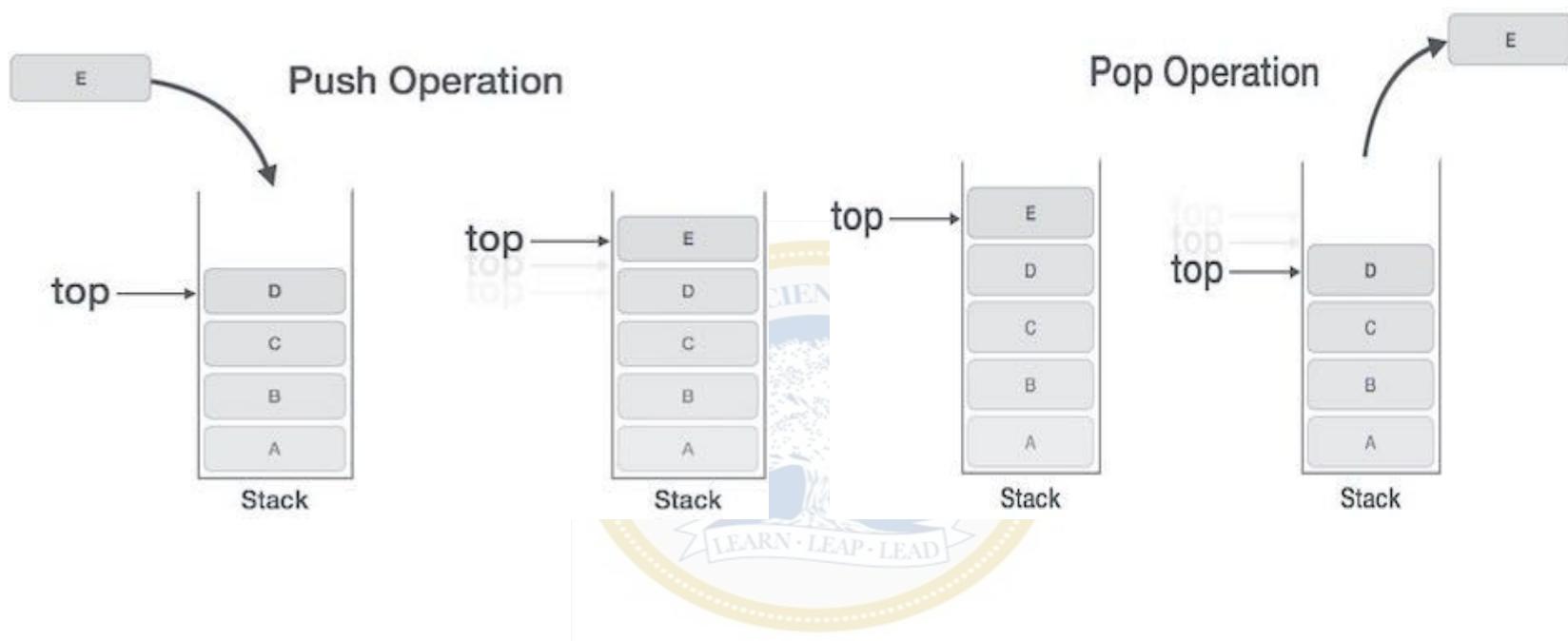
STACK – Additional Functionality

- For effective utilization of the stack, Status of the stack should be checked. For that additional functionality is of the stack is given below
- **Peek or Top**
 - Accessing the top element. Without removing the top element.
- **isFull**
 - check if stack is full.
- **isEmpty**
 - check if stack is empty.





Stack Operation Example





Stack Implementation - Array





Stack - Array

- One dimensional array is enough to implement the stack
- Array size always fixed
- Easy to implement
- Create fixed size one dimensional array
 - insert or delete the elements into the array using **LIFO principle** using the variable 'top'



Stack - *top*

- About *top*
 - Initial value of the top is -1
 - To insert a value into the stack, increment the top value by one and then insert
 - To delete a value from the stack, delete the top value and decrement the top value by one



Steps to create an empty stack

1. Declare the functions like push, pop, display etc. need to implement the stack.
2. Declare one dimensional array with fixed size
3. Declare a integer variable 'top' and initialize it with '-1'.
(int top = -1)





push(Value)

Inserting value into the stack

- push() is a function used to insert a new element into stack at **top** position.
- Push function takes one integer value as parameter
 1. Check whether **stack** is **FULL** based on (**top == SIZE-1**)
 2. If stack is **FULL**, then display "**Stack is FULL Not able to Insert element**" and terminate the function.
 3. If stack is **NOT FULL**, then increment **top** value by one (**top=top+1**) and set (**stack[top] = value**)



pop() – Delete a value from the Stack

- pop() is a function used to delete an element from the stack from **top** position
- Pop function does not take any value as parameter
 1. Check whether **stack** is **EMPTY** using (**top == -1**)
 2. If stack is **EMPTY**, then display "**Stack is EMPTY No elements to delete**" and terminate the function
 3. If stack is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top=top-1**)



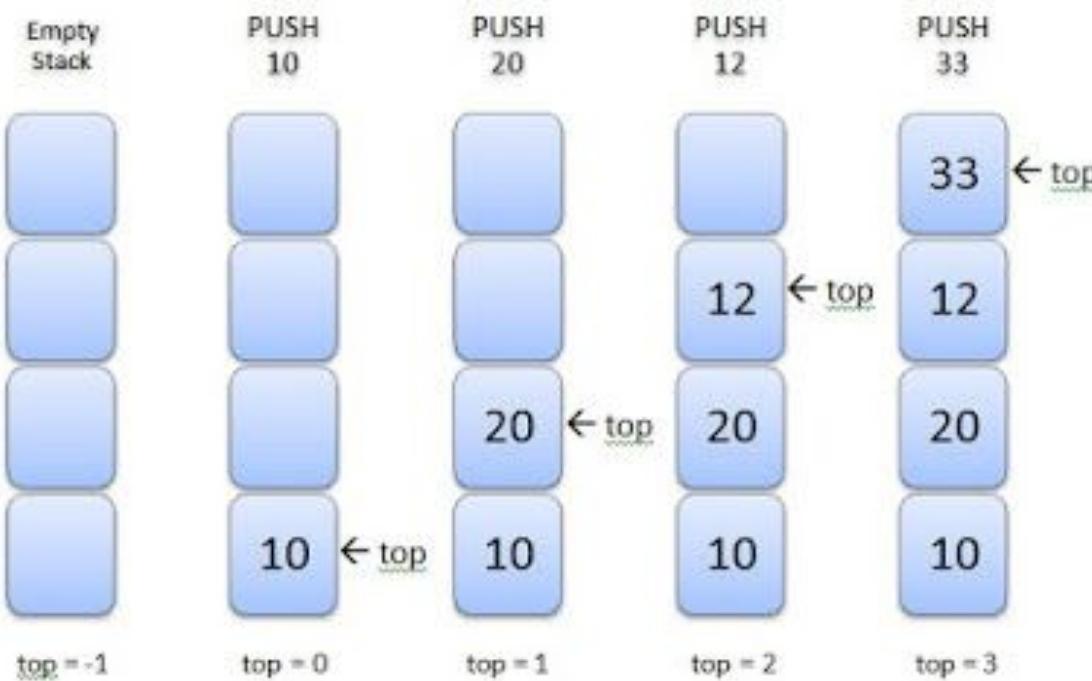
display()

Displays the elements of a Stack

- **display() – function used to Display the elements of a Stack**
 1. Check whether stack is **EMPTY** based on (**top == -1**)
 2. If stack is **EMPTY**, then display "**Stack is EMPTY**" and terminate the function
 3. If stack is **NOT EMPTY**, display the value in decreasing order of array



Stack Push Example



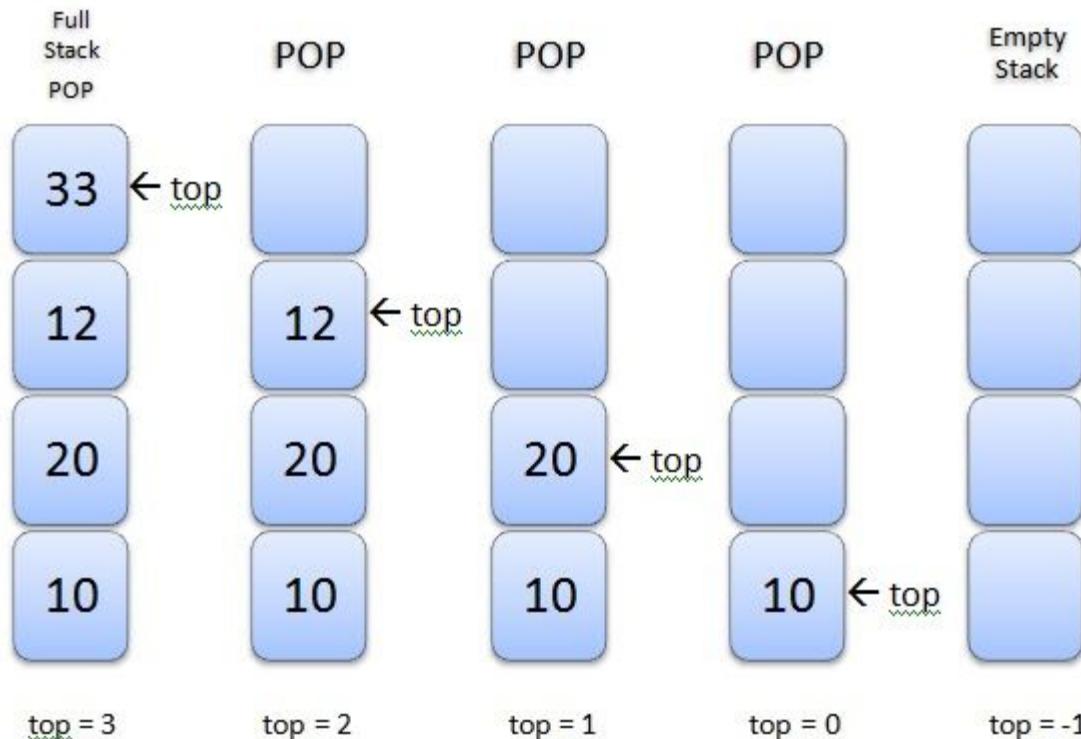
PUSH operation on Stack with the position of top pointer

'top' is initialized to -1. Each time an element is pushed, then $\text{top} = \text{top}+1$.

Diagram reference :<http://www.exploredatabase.com/2018/01/stack-abstract-data-type-data-structure.html>



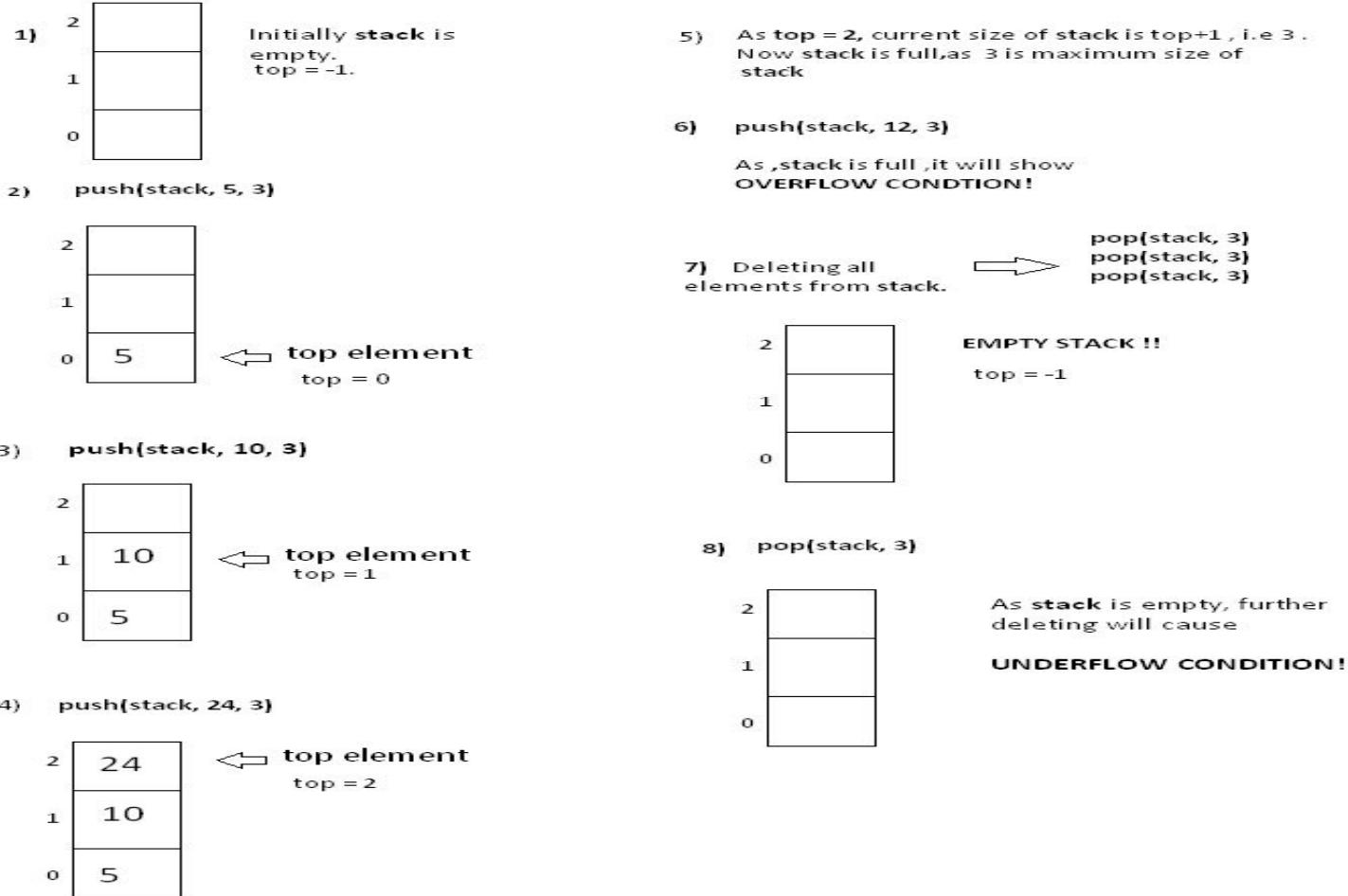
Stack Pop Example



POP operation on Stack with the position of top pointer
Each time an element is popped, then $\text{top} = \text{top} - 1$.



Another Example of Stack Push & Pop





Cons Stack - Array

- The size of the array should be mentioned at the beginning of the implementation
- If more memory needed or less memory needed that time this array implementation is not useful

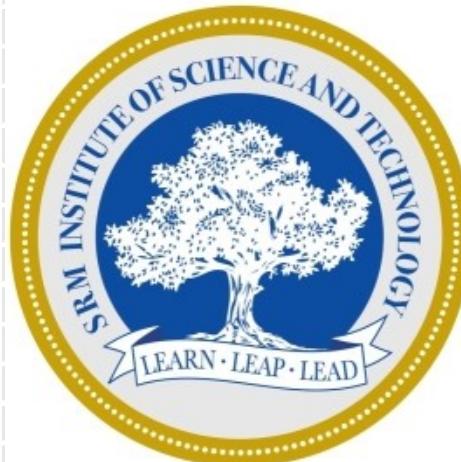




SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE





SESSION 2





Stack Implementation – Linked List





Pros Stack – Linked List

- A stack data structure can be implemented by using a linked list
- The stack implemented using linked list can work for the variable size of data. So, there is no need to fix the size at the beginning of the implementation





Stack – Linked list

- Dynamic Memory allocation of Linked list is followed
- The nodes are scattered and non-contiguously in the memory
- Each node contains a pointer to its immediate successor node in the stack
- Stack is said to be overflowed if the space left in the memory heap is not enough to create a node



Stack – Linked list

- In linked list implementation of a stack, every new element is inserted as '**top**' element.
- That means every newly inserted element is pointed by '**top**'.
- To remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list.
- The **next** field of the **First** element must be always **NULL**.



Example Stack – Linked List

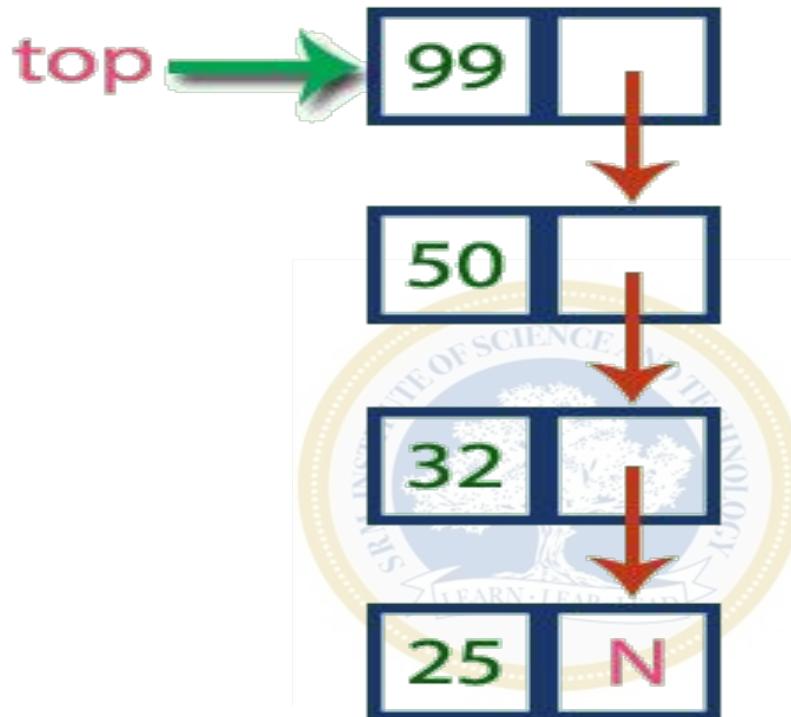
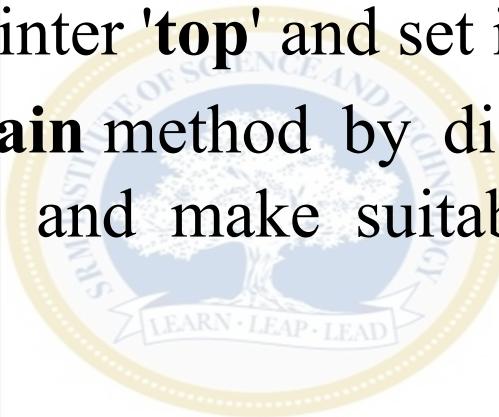


Diagram Reference :http://www.btechsmartclass.com/data_structures/stack-using-linked-list.html



Create Node – Linked List

1. Include all the **header files** which are used in the program. And declare all the **user defined functions**
2. Define a '**Node**' structure with two members **data** and **next**
3. Define a **Node** pointer '**top**' and set it to **NULL**
4. Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method

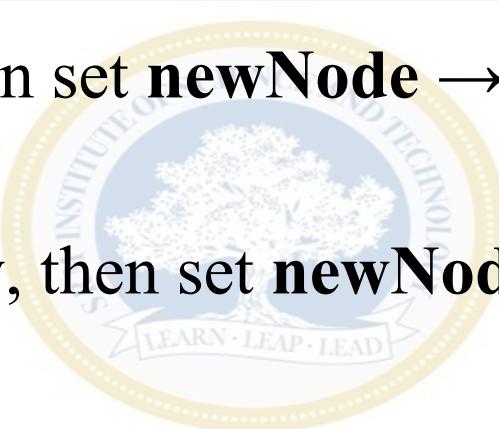




push(value)

Inserting an element into the Stack

1. Create a **newNode** with given value
2. Check whether stack is Empty (**top == NULL**)
3. If it is **Empty**, then set **newNode → next = NULL**
4. If it is **Not Empty**, then set **newNode → next = top**
5. Finally, set **top = newNode**

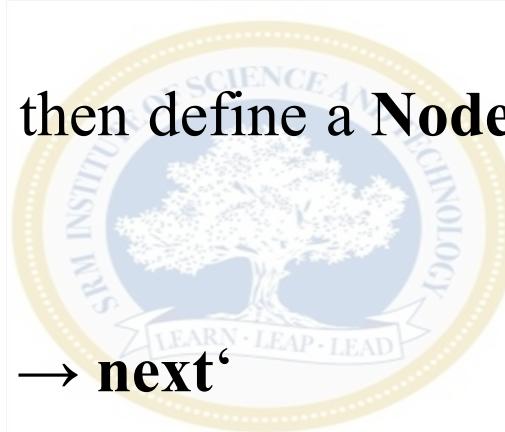




pop()

Deleting an Element from a Stack

1. Check whether **stack** is Empty (**top == NULL**)
2. If it is **Empty**, then display "**Stack is Empty!!! Deletion is not possible!!!**" and terminate the function
3. If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'
4. Then set '**top = top → next**'
5. Finally, delete '**temp**'. (**free(temp)**)





display()

Displaying stack of elements

1. Check whether stack is **Empty** (**top == NULL**)
2. If it is **Empty**, then display '**Stack is Empty!!!**' and terminate the function
3. If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top**
4. Display '**temp → data --->**' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next != NULL**)
5. Finally! Display '**temp → data ---> NULL**'



Stack - Applications

1. Infix to Postfix Conversion
2. Postfix Evaluation
3. Balancing Symbols
4. Nested Functions
5. Tower of Hanoi





Infix to Postfix Conversion





Introduction

- Expression conversion is the most important application of stacks
- Given an infix expression can be converted to both prefix and postfix notations
- Based on the Computer Architecture either Infix to Postfix or Infix to Prefix conversion is followed





What is Infix, Postfix & Prefix?

- **Infix Expression :** The operator appears in-between every pair of operands. **operand1 operator operand2 (a+b)**
- **Postfix expression:** The operator appears in the expression after the operands. **operand1 operand2 operator (ab+)**
- **Prefix expression:** The operator appears in the expression before the operands. **operator operand1 operand2 (+ab)**



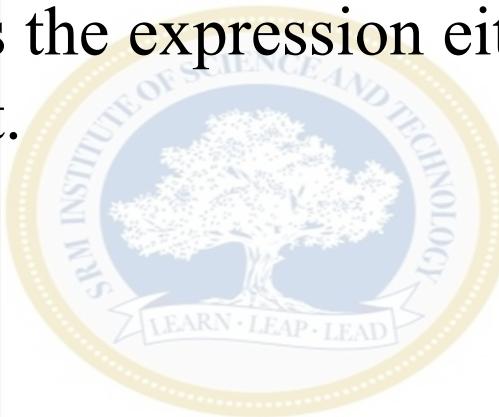
Example – Infix, Prefix & Postfix

Infix	Prefix	Postfix
$(A + B) / D$	$/ + A B D$	$A B + D /$
$(A + B) / (D + E)$	$/ + A B + D E$	$A B + D E + /$
$(A - B / C + E)/(A + B)$	$/ + - A / B C E + A B$	$A B C / - E + A B + /$
$B ^ 2 - 4 * A * C$	$- ^ B 2 * * 4 A C$	$B 2 ^ 4 A * C * -$



Why postfix representation of the expression?

- Infix expressions are readable and solvable by humans because of easily distinguishable order of operators, but compiler doesn't have integrated order of operators.
- The compiler scans the expression either from left to right or from right to left.





Why postfix representation of the expression?

- Consider the below expression: a op1 b op2 c op3 d
If op1 = +, op2 = *, op3 = +

a + b *c +d

- The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.



Why postfix representation of the expression?

- The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.
- The corresponding expression in postfix form is:
 abc^*+d+ .
- The postfix expressions can be evaluated easily in a single scan using a stack.



ALGORITHM

Step 1 : Scan the Infix Expression from left to right.

Step 2 : If the scanned character is an operand, append it with final Infix to Postfix string.

Step 3 : Else,

Step 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a ‘(’), push it on stack.

Step 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Step 4 : If the scanned character is an ‘(‘ , push it to the stack.

Step 5 : If the scanned character is an ‘)’ , pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.

Step 6 : Repeat steps 2-6 until infix expression is scanned.

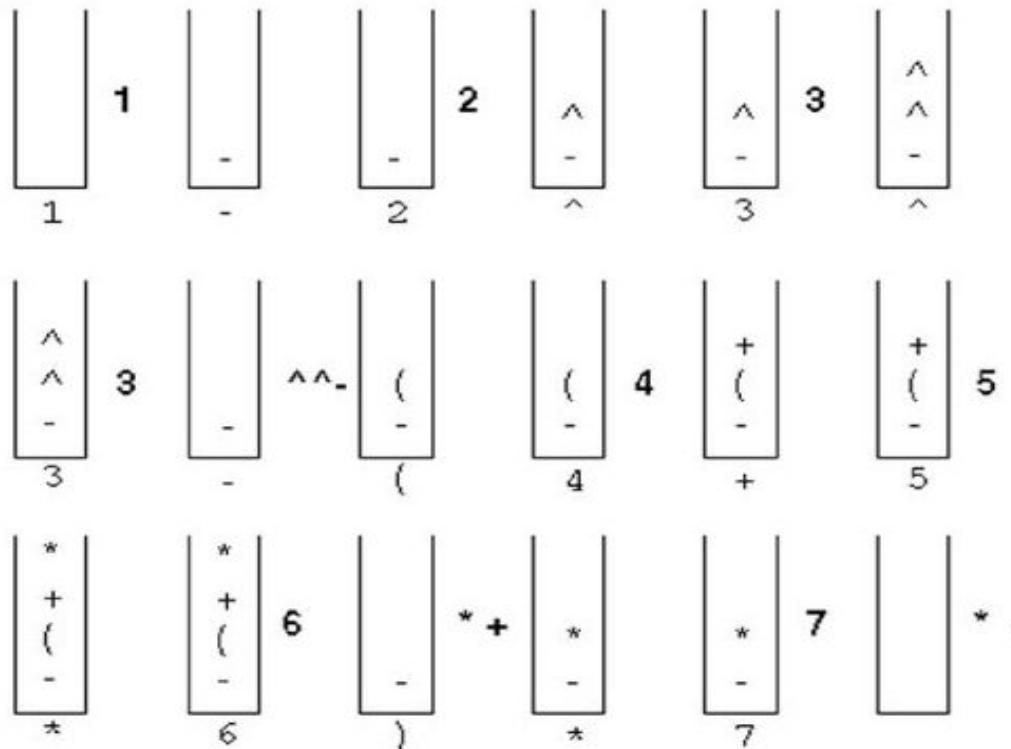
Step 7 : Print the output

Step 8 : Pop and output from the stack until it is not empty.



Example

Prefix: 1 - 2 ^ 3 ^ 3 - (4 + 5 * 6) * 7



Postfix: 1 2 3 3 ^ ^ - 4 5 6 * + 7 * -



Exercise Problems to Solve

Infix to Postfix conversion

1. $A+B-C$
2. $A+B*C$
3. $(A+B)*C$
4. $(A+B)*(C-D)$
5. $((A+B)*C-(D-E))\%(F+G)$
6. $A*(B+C/D)$
7. $((A*B)+(C/D))$
8. $((A*(B+C))/D)$
9. $(A*(B+(C/D)))$
10. $(2+((3+4)*(5*6)))$
11. $B^2 - 4 * A * C$
12. $(A - B / C + E)/(A + B)$

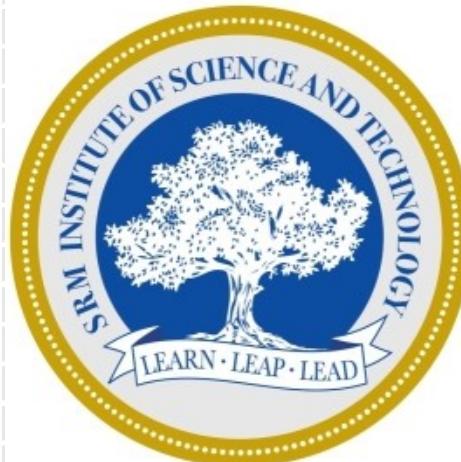




SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE





SESSION 3





Postfix Expression Evaluation





Postfix Expression

- A postfix expression is a collection of operators and operands in which the operator is placed after the operands.



Image Reference : http://www.btechsmartclass.com/data_structures/expressions.html



Postfix Expression Evaluation using Stack





Algorithm

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.



Example 1

Infix Expression $(5 + 3) * (8 - 2)$

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression	
Initially	Stack is Empty	Nothing	
5	push(5)	Nothing	$\begin{array}{l} \text{value1 = pop(); // 2} \\ \text{value2 = pop(); // 8} \\ \text{result = value2 - value1} \\ \text{push(result)} \end{array}$ 6 8 $(8 - 2)$ $(5 + 3), (8 - 2)$
3	push(3)	Nothing	$\begin{array}{l} \text{value1 = pop(); // 6} \\ \text{value2 = pop(); // 8} \\ \text{result = value2 * value1} \\ \text{push(result)} \end{array}$ 48 $(6 * 8)$ $(5 + 3) * (8 - 2)$
+	$\begin{array}{l} \text{value1 = pop(); // 3} \\ \text{value2 = pop(); // 5} \\ \text{result = value2 + value1} \\ \text{push(result)} \end{array}$	$(5 + 3)$	$\$$ End of Expression
8	push(8)	$(5 + 3)$	$\begin{array}{l} \text{result = pop()} \end{array}$
2	push(2)	$(5 + 3)$	Display (result) 48 As final result

Infix Expression $(5 + 3) * (8 - 2) = 48$

Postfix Expression 5 3 + 8 2 - * value is **48**



Example 2

Postfix Expression: 1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -

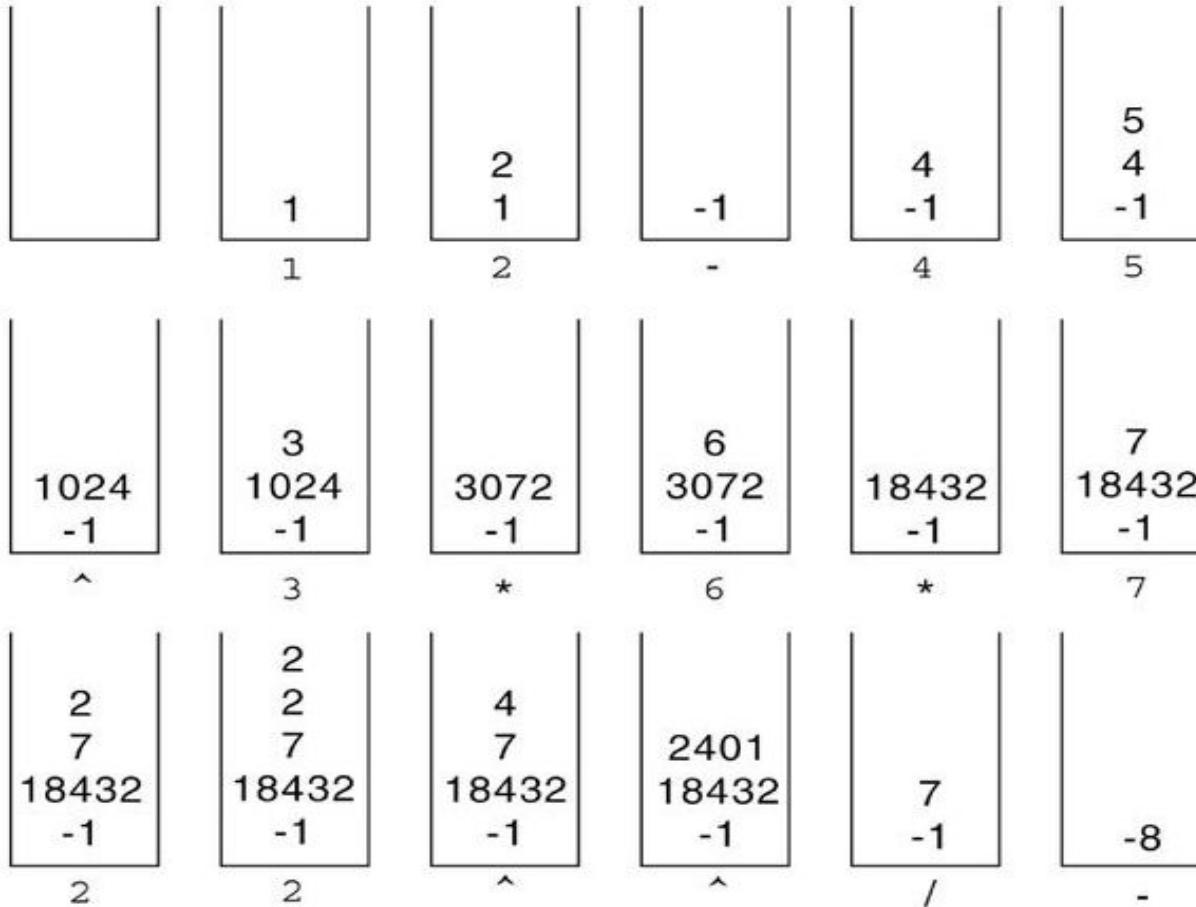


Image from Data Structures by Mark Allen Weiss book



Balancing Symbols





Introduction

- Stacks can be used to check if the given expression has balanced symbols or not.
- The algorithm is very much useful in compilers.
- Each time parser reads one character at a time.
 - If the character is an opening delimiter like ‘(‘ , ‘{‘ or ‘[‘ then it is PUSHED in to the stack.
 - When a closing delimiter is encountered like ‘)’ , ‘}’ or ‘]’ is encountered, the stack is popped.
 - The opening and closing delimiter are then compared.
 - If they match, the parsing of the string continues.
 - If they do not match, the parser indicates that there is an error on the line.



Balancing Symbols - Algorithm

- Create a stack
- while (end of input is not reached) {
 - If the character read is not a symbol to be balanced, ignore it.
 - If the character is an opening delimiter like (, { or [, PUSH it into the stack.
 - If it is a closing symbol like) , } ,] , then if the stack is empty report an error, otherwise POP the stack.
 - If the symbol POP-ed is not the corresponding delimiter, report an error.
- At the end of the input, if the stack is not empty report an error.



Example 1

Expression	Valid?	Description
$(A+B) + (C-D)$	Yes	The expression is having balanced symbol
$((A+B) + (C-D)$	No	One closing brace is missing.
$((A+B) + [C-D])$	Yes	Opening and closing braces correspond
$((A+B) + [C-D]$	No	The last brace does not correspond with the first opening brace.



Example 2

Eg: $[a+(b*c)+(d-e)]$

[Push [
[(Push (
[) and (matches, Pop (
[{	Push {
[{ (Push (
[{	matches, pop (
[Matches, pop {
	Matches, pop [

Thus, parenthesis match here

Image Reference : <https://giunub.com/nimpatel/Parenthesis-matching-with-Reduce-reduce-method>



References

1. https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm
2. <https://www.codesdope.com/course/data-structures-stacks/>
3. <http://www.firmcodes.com/write-a-c-program-to-implement-a-stack-using-an-array-and-linked-list/>
4. http://www.btechsmartclass.com/data_structures/stack-using-linked-list.html
5. <http://www.exploredatabase.com/2018/01/stack-abstract-data-type-data-structure.html>
6. <https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/>
7. <https://www.hackerearth.com/practice/notes/stacks-and-queues/>
8. <https://github.com/niinpatel/Parenthesis-Matching-with-Reduce-Method>
9. <https://www.studytonight.com/data-structures/stack-data-structure>
10. <https://www.programming9.com/programs/c-programs/230-c-program-to-convert-infix-to-postfix-expression-using-stack>



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE





SESSION 6





Applications of Stack: Function Call and Return

- 2 types of Memory
 - Stack
 - Heap
- Stack memory stores the data (variables) related to each function.
- Why is it called stack memory?
 - The last function to be called is the first to return (LIFO)
 - The data related to a function are pushed into the stack when a function is called and popped when the function returns



Nested Function Calls

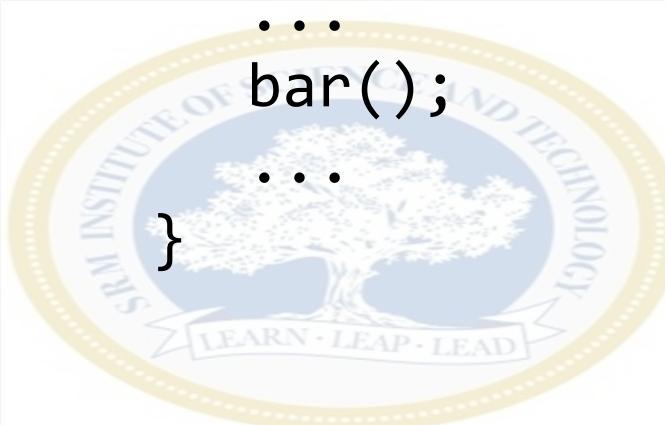
- Consider the code snippet below:

```
main()
{
    ...
    foo();
    bar();
}
```

```
foo()
{
```

```
    ...
    bar();
    ...
}
```

```
bar()
{
    ...
}
```





Nested Function Calls and Returns in Stack Memory

- Stack memory when the code executes:

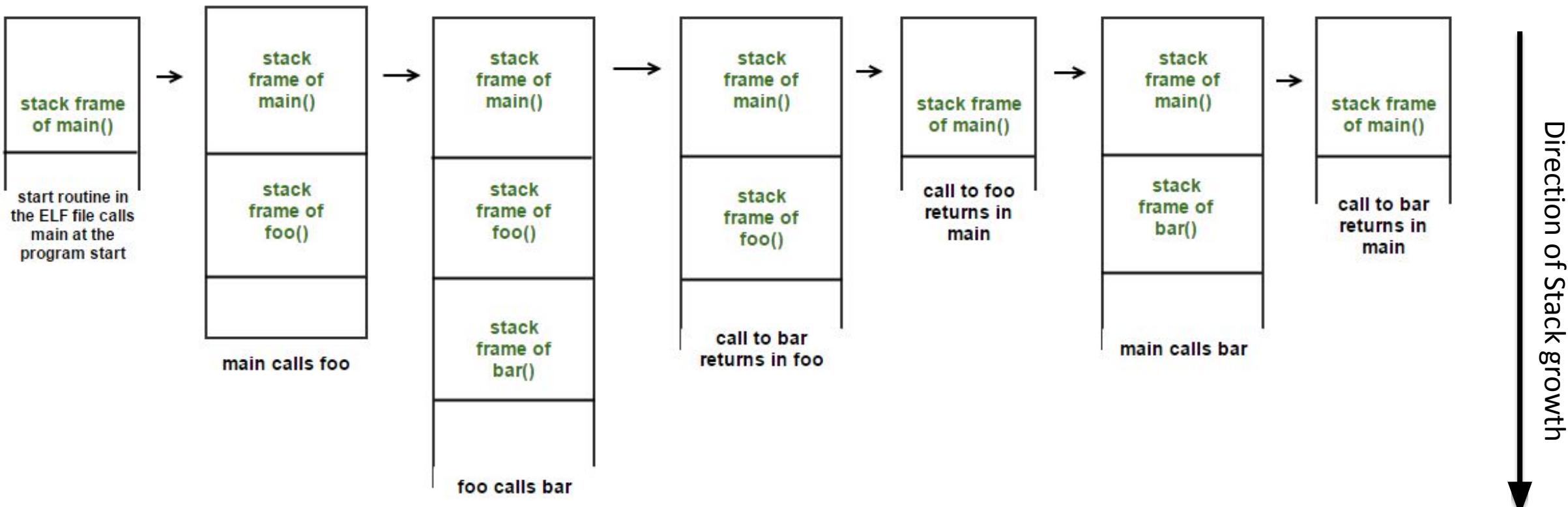


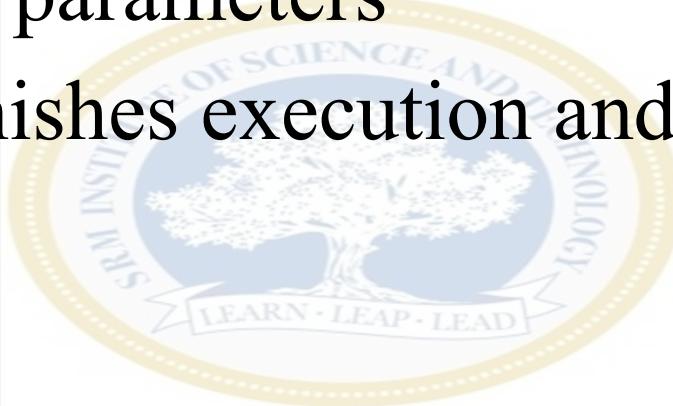
Image Source:

<https://loonytek.com/2015/04/28/call-stack-internals-part-1/>



Function Call and Return in Stack Memory

- Each call to a function pushes the function's activation record (or stack frame) into the stack memory
- Activation record mainly consists of: local variables of the function and function parameters
- When the function finishes execution and returns, its activation record is popped





Recursion

- A recursive function is a function that calls itself during its execution.
- Each call to a recursive function pushes a new activation record (a.k.a stack frame) into the stack memory.
- Each new activation record will consist of freshly created local variables and parameters for that specific function call.
- So, even though the same function is called multiple times, a new memory space will be created for each call in the stack memory.



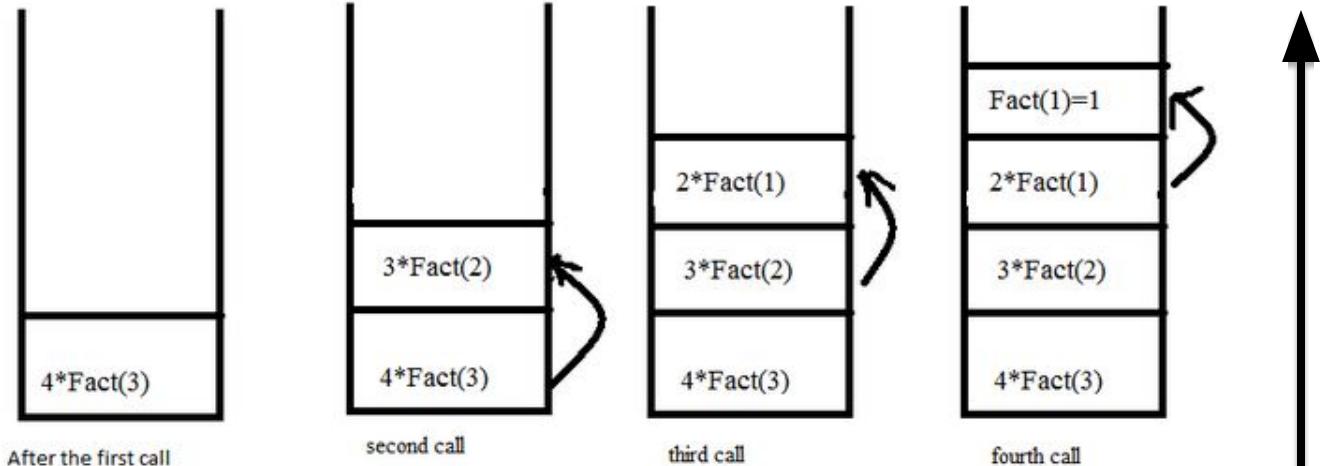
Recursion Handling by Stack Memory

When function call happens previous variables gets stored in stack

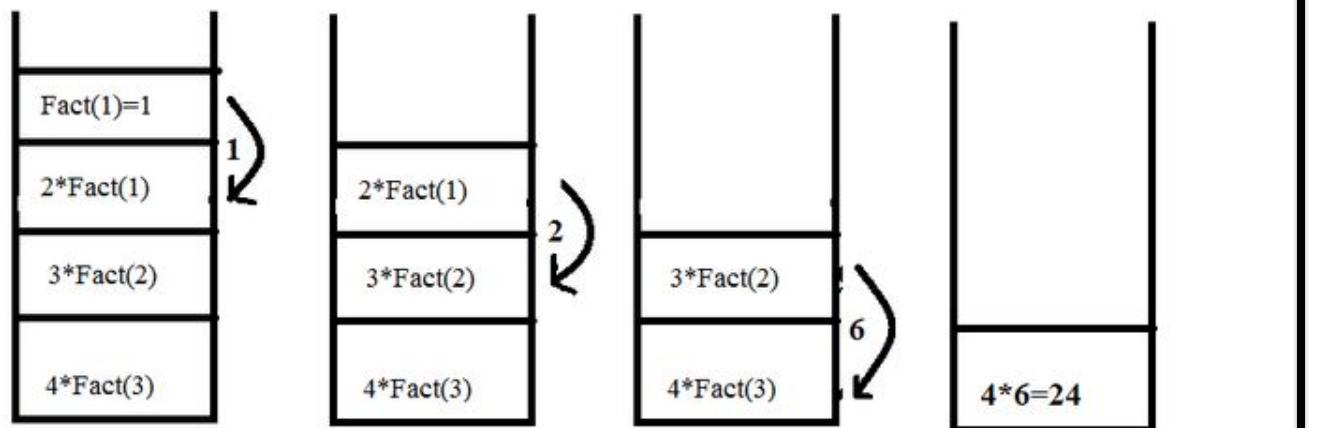
- Factorial program

```
int fact( int n)
{
    if (n==1)
        return 1;
    else
        return (n*fact(n-1));
}

fact(4);
```



Returning values from base case to caller function





SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE





Session 7





Applications of Recursion: Towers of Hanoi

Towers of Hanoi Problem:

There are 3 pegs and n disks. All the n disks are stacked in 1 peg in the order of their sizes with the largest disk at the bottom and smallest on top. All the disks have to be transferred to another peg.

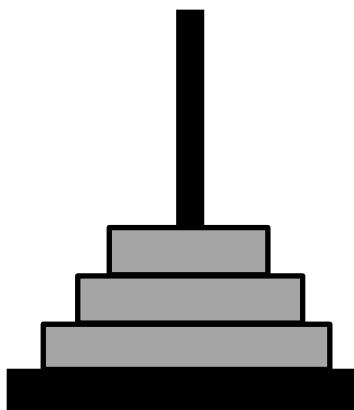
Rules for transfer:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

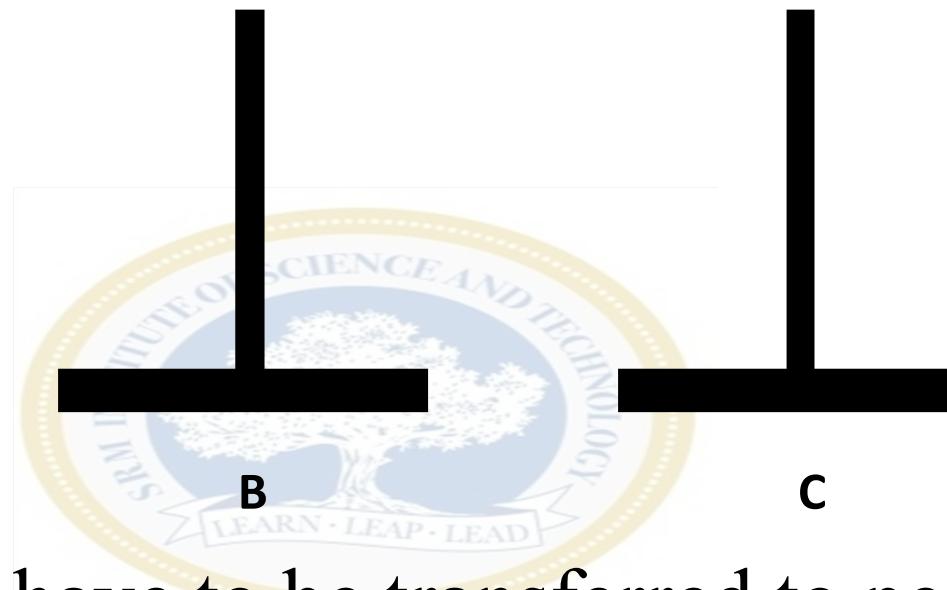


Towers of Hanoi

- Initial Position:



A



B

C

- All disks from peg A have to be transferred to peg C



Towers of Hanoi Solution for 3 disks

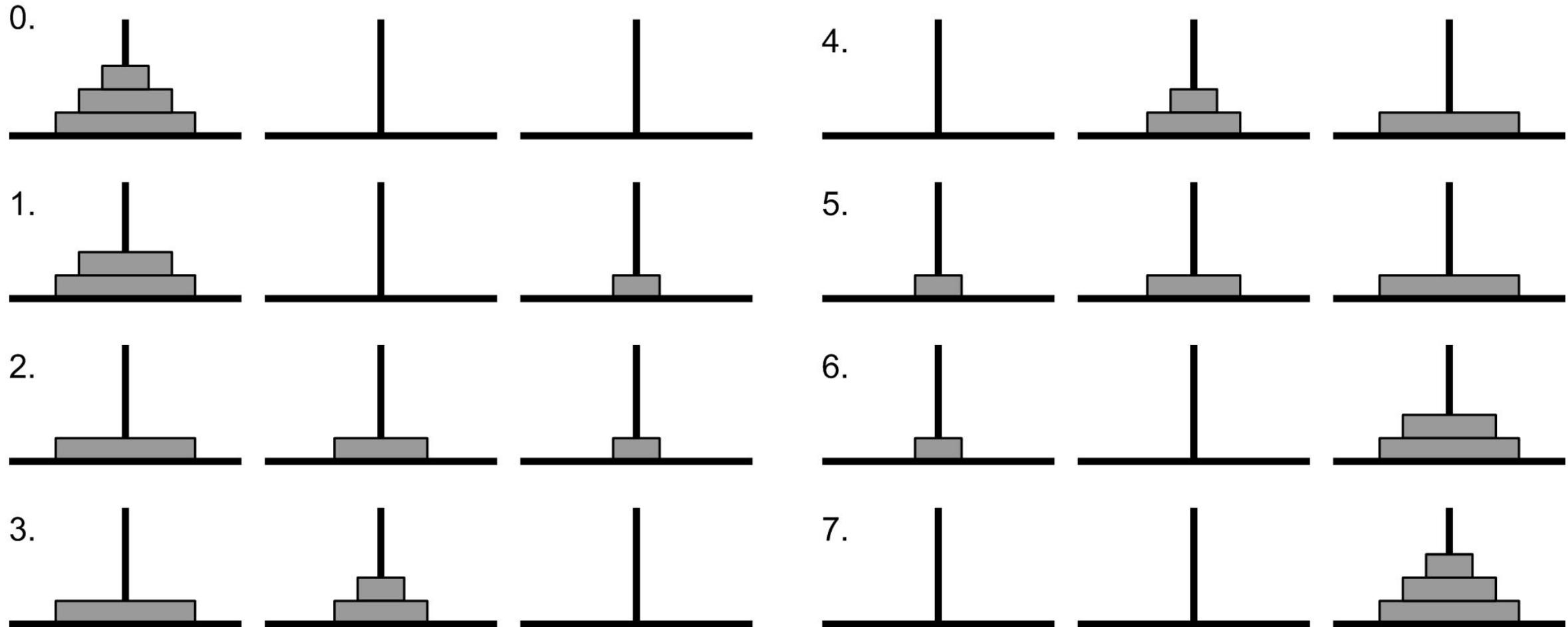


Image Source: <https://www.includehelp.com/data-structure-tutorial/tower-of-hanoi-using-recursion.aspx>



Towers of Hanoi – Recursive Solution

```
/* N = Number of disks  
   Beg, Aux, End are the pegs */
```

```
Tower(N, Beg, Aux, End)
```

```
Begin
```

```
    if N = 1 then
```

```
        Print: Beg --> End;
```

```
    else
```

```
        Call Tower(N-1, Beg, End, Aux);
```

```
        Print: Beg --> End;
```

```
        Call Tower(N-1, Aux, Beg, End);
```

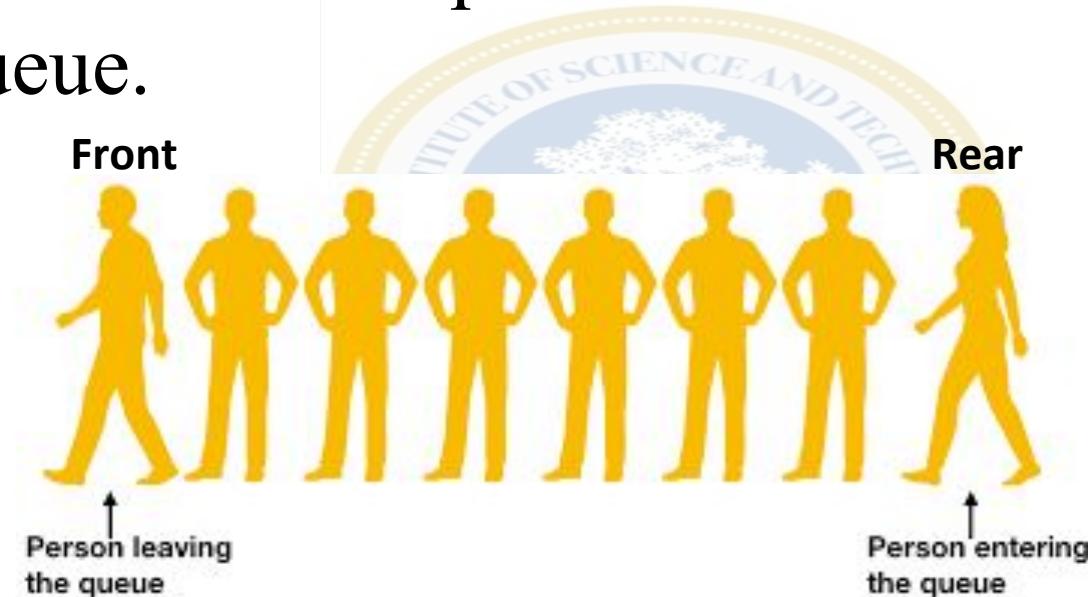
```
    endif
```

```
End
```



The Queue ADT

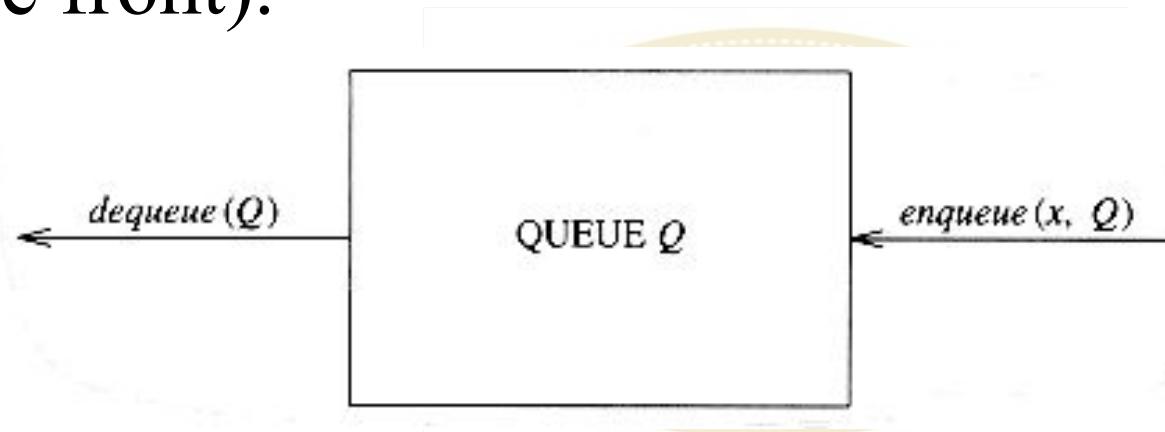
- With a queue, insertion is done at one end whereas deletion is performed at the other end.
- It is a **First In First Out (FIFO)** Structure. That is, the first element to enter into the queue will be the first element to get out of the queue.





The basic operations on a Queue

- Enqueue: inserts an element at the end of the list (called the rear)
- Dequeue: deletes (and returns) the element at the start of the list (called the front).



- Peek: Get the front element



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE





SESSION 8





Queue Representations

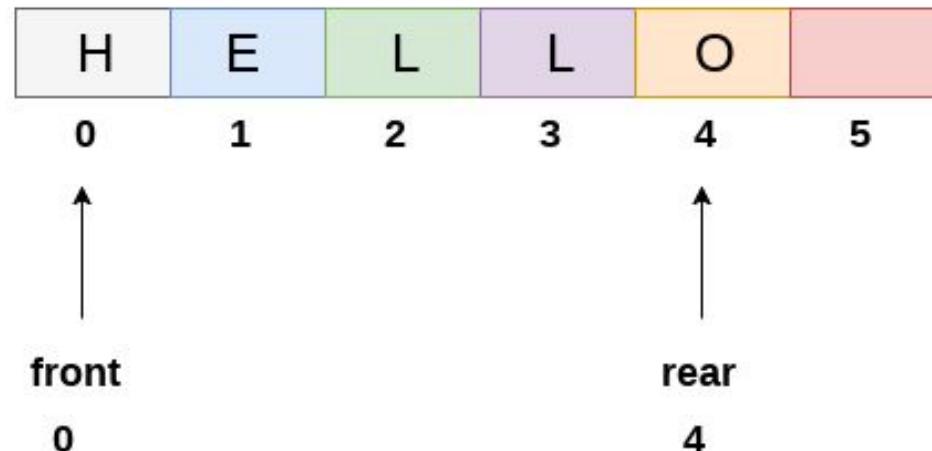
- Queues can be represented using:
 - Arrays
 - Linked Lists





Array Implementation of Queues

- Queue can be implemented using a one-dimensional array.
- We need to keep track of 2 variables: *front* and *rear*
- *front* is the index in which the first element is present
- *rear* is the index in which the last element is present

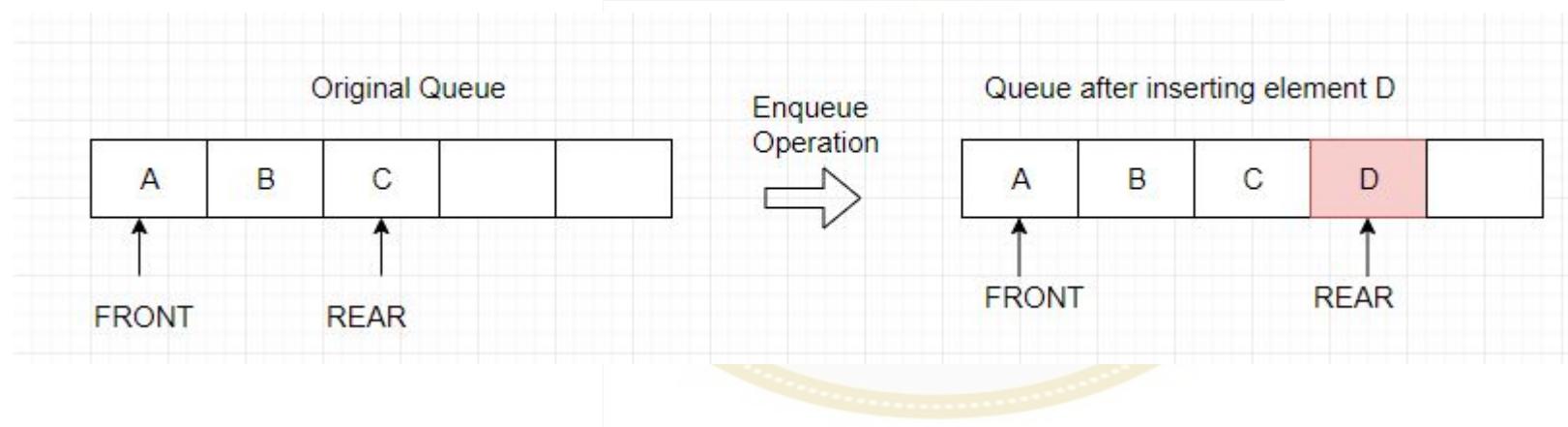




Enqueue in Array Implementation of Queue

```
//Basic Idea  
enqueue(num):  
    rear++  
    queue[rear]=num  
    return
```

Time Complexity: O(1)

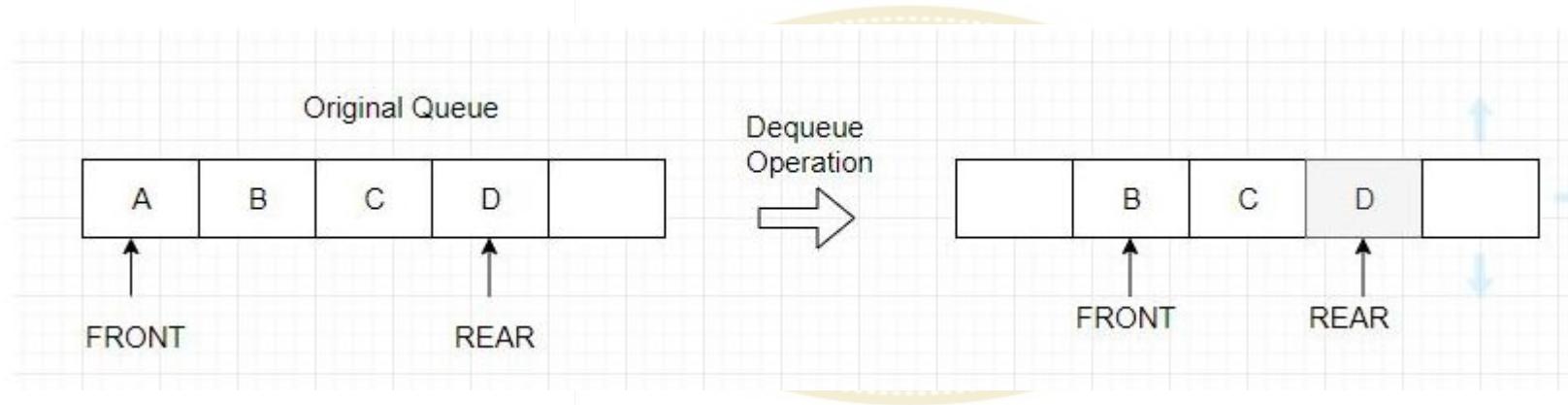




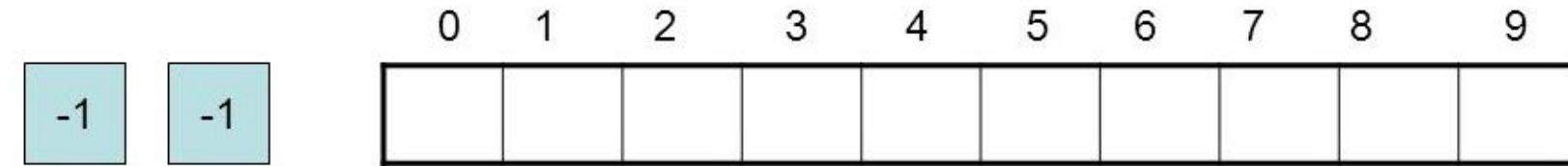
Dequeue in Array Implementation of Queue

```
//Basic Idea  
dequeue():  
    num = queue[front]  
    front++  
    return num
```

Time Complexity: $O(1)$
 $O(n)$ if elements are shifted

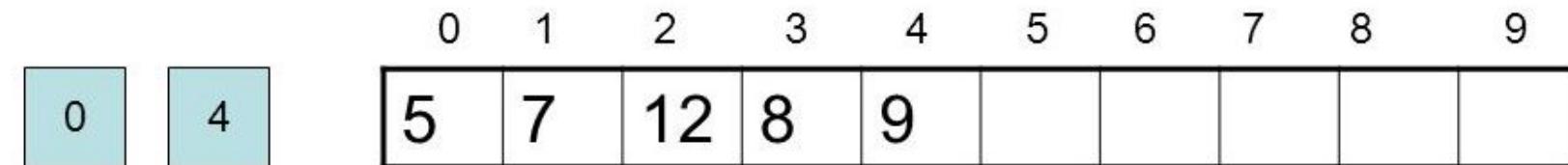


Enqueue and Dequeue in Queue



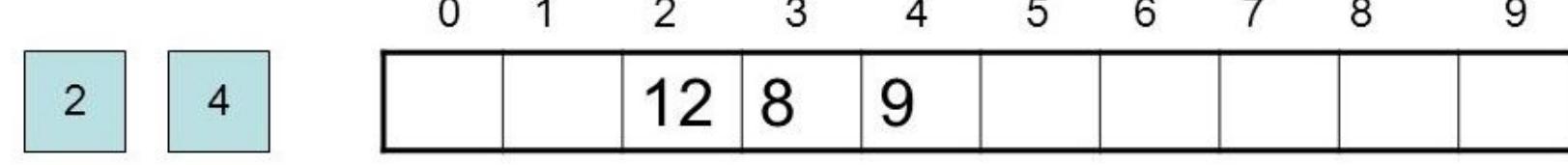
front rear

Representation of queue in memory



front rear

Representation of queue in memory



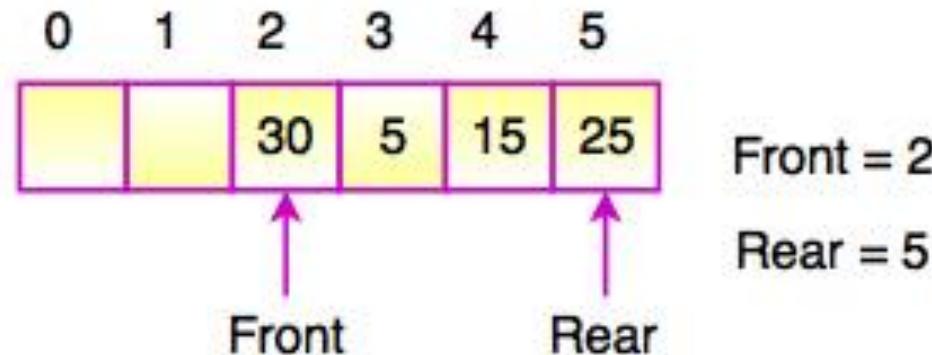
front rear

Representation of queue in memory



Disadvantage of Array Implementation of Queue

- Multiple enqueue and dequeue operations may lead to situation like this:



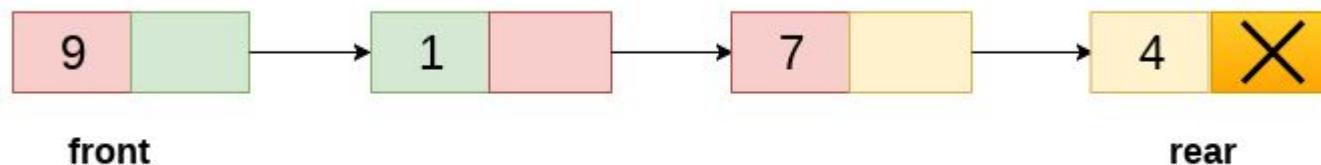
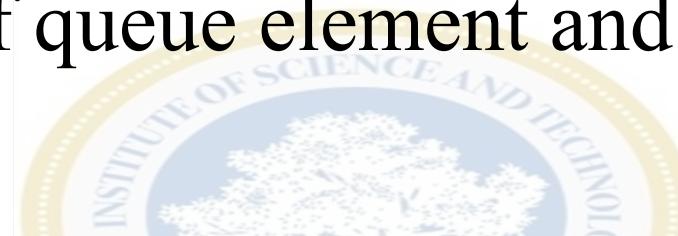
<https://www.tutorialride.com/data-structures/queue-in-data-structure.htm>

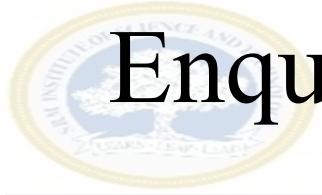
- No more insertion is possible (since rear cannot be incremented anymore) even though 2 spaces are free.
- Workaround: Circular Queue, Shifting elements after each dequeue



Linked List Implementation of Queue

- In the linked list implementation of queue, two pointers are used, namely, *front* and *rear*
- New nodes are created when an element has to be inserted.
- Each node consists of queue element and a pointer to the next node



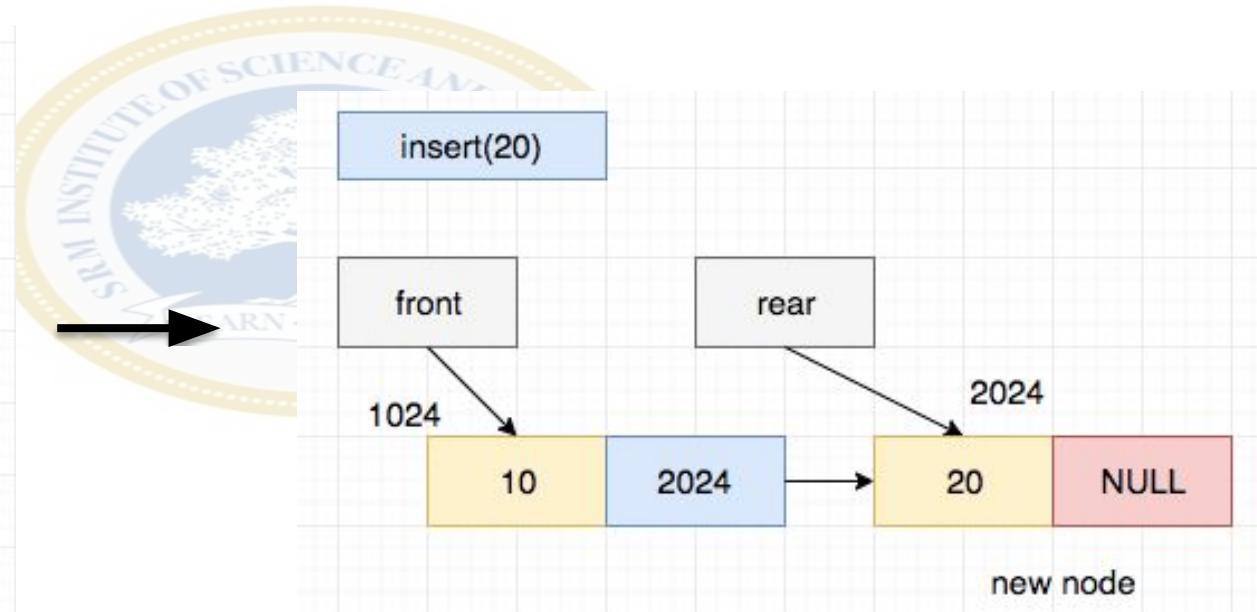
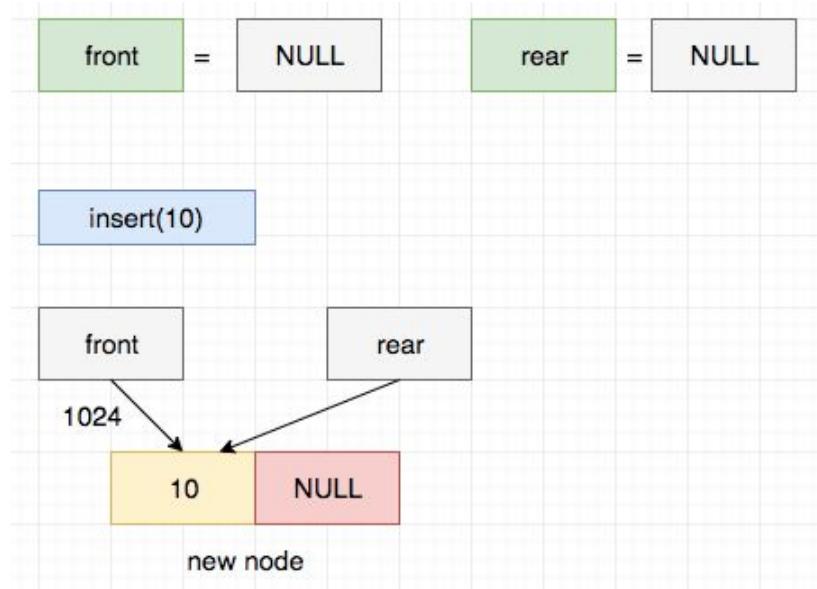


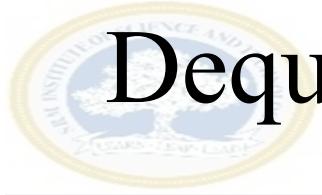
Enqueue Operation in Linked List Implementation of Queue

- Enqueue basic code:

```
ptr -> data = item;  
rear -> next = ptr;  
rear = ptr;  
rear -> next = NULL;
```

Time Complexity: O(1)





Dequeue Operation in Linked List Implementation of Queue

- Dequeue basic code:

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

Time Complexity: O(1)

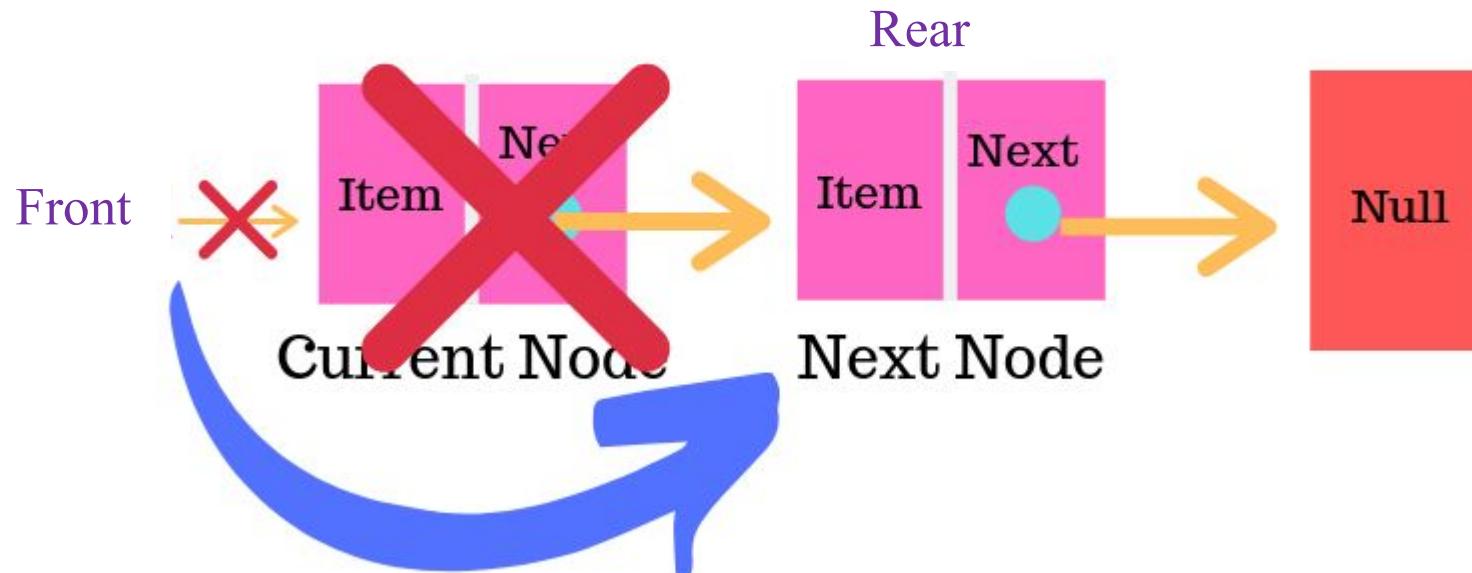


Image Source: <https://learnersbucket.com/tutorials/data-structures/implement-queue-using-linked-list/>



References

- Seymour Lipschutz, Data Structures, Schaum's Outlines, 2014.

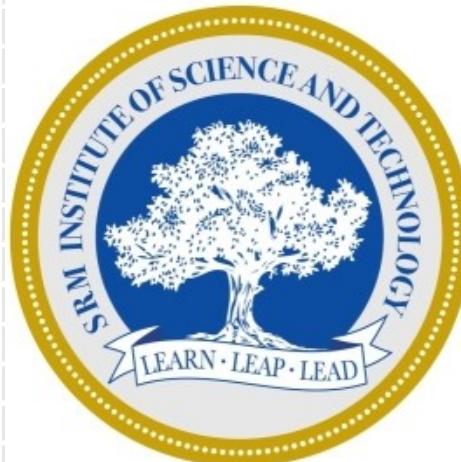




SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE



SESSION 11

Circular Queue

Definition

Circular Queue is a linear data structure in which first position of the queue is connected with last position of the queue.

In other words, The queue is considered as a circular queue when the positions 0 and MAX-1 are adjacent.

It is also referred as RING BUFFER.

Principle Used

- FIFO (First In First Out) is used for performing operation.

Operations Involved

- Enqueue
- Dequeue

Variables Used

- MAX- Number of entries in the array
- Front – is the index of front queue entry in an array
(Get the front item from queue)
- Rear – is the index of rear queue entry in an array.(Get the last item from queue)

Concepts of Circular Queue

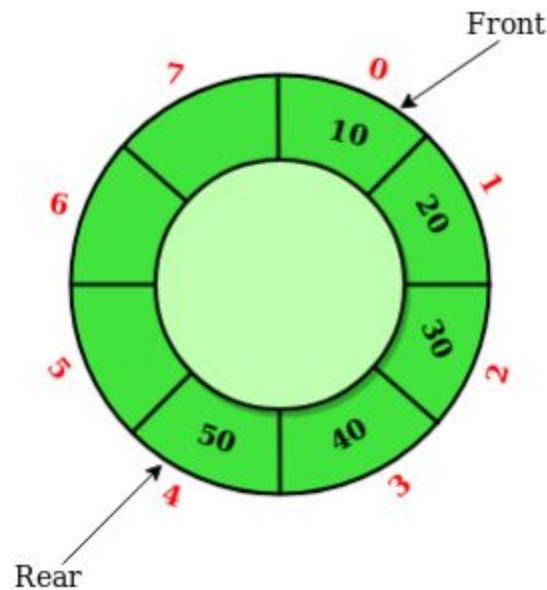
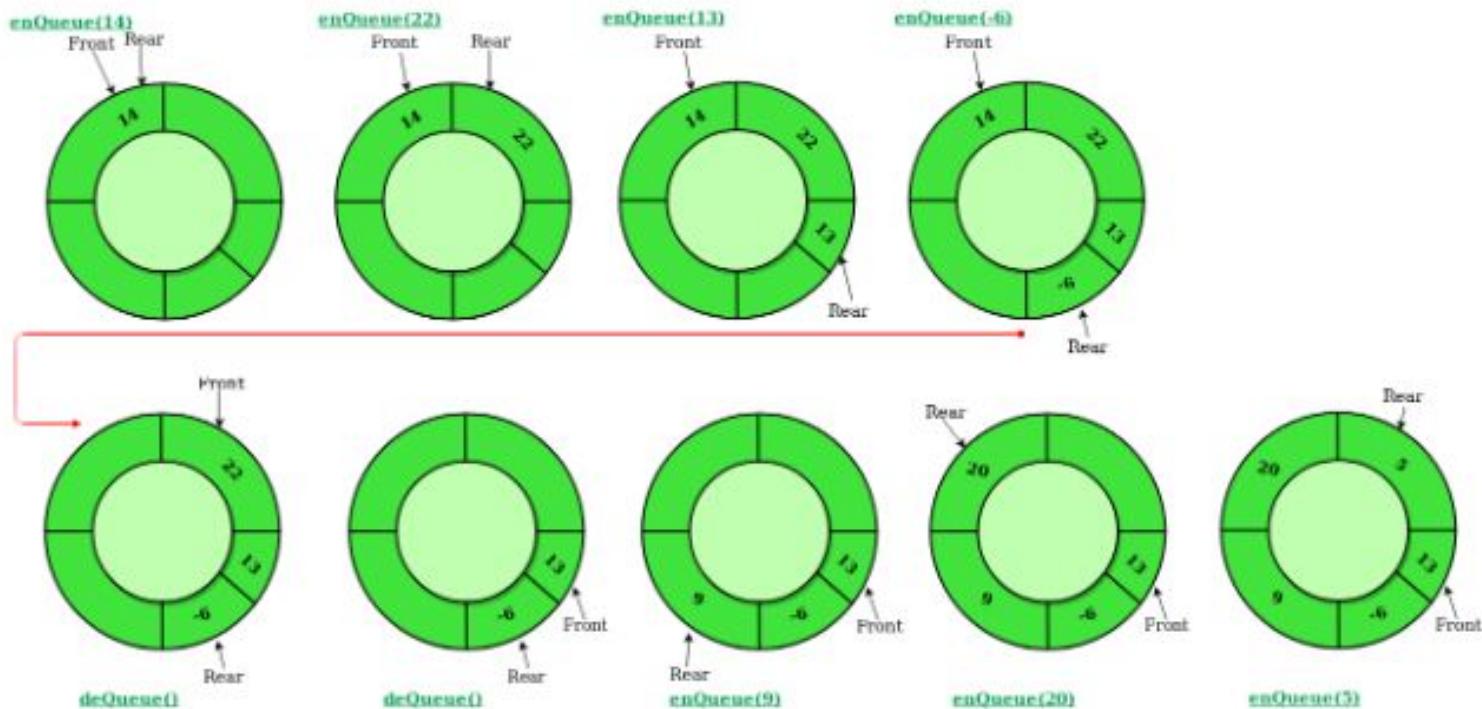


Illustration of Enqueue and Dequeue



Conditions used in Circular Queue

- Front must point to the first element.
- The queue will be empty if $\text{Front} = \text{Rear}$.
- When a new element is added the queue is incremented by value one($\text{Rear} = \text{Rear} + 1$).
- When an element is deleted from the queue the front is incremented by one ($\text{Front} = \text{Front} + 1$).

Steps Involved in Enqueue

Check whether queue is Full or not by using the following condition

$$((\text{rear} == \text{SIZE}-1 \ \&\& \ \text{front} == 0) \ || (\text{rear} == \text{front}-1))$$

If queue is full, display the queue is full else we can insert an element by incrementing rear pointer.

Steps Involved in Dequeue

1. Check whether queue is Empty or not by using the following condition
 if(front== -1).
2. If it is empty then display Queue is empty, else we can delete the element.

Difference between Queue and Circular Queue

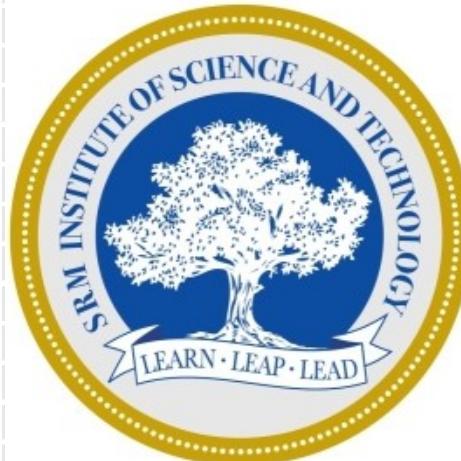
- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.
- The unused memory locations in the case of ordinary queues can be utilized in circular queues.



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE



SESSION 12

Circular Queue Example-ATM

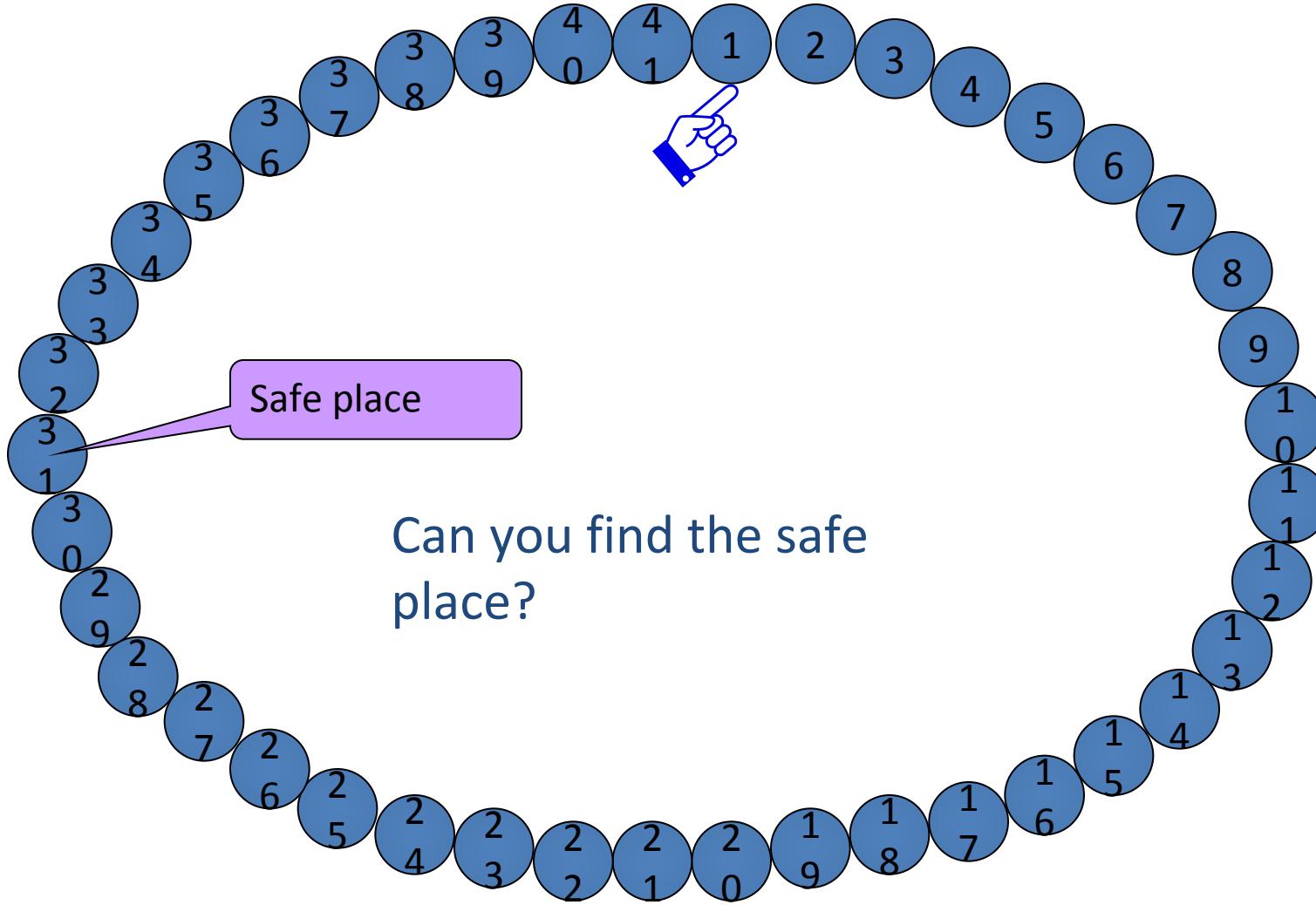
ATM is the best example for the circular queue. It does the following using circular queue

1. ATM sends request over private network to central server
2. Each request takes some amount of time to process.
3. More request may arrive while one is being processed.
4. Server stores requests in queue if they arrive while it is busy.
5. Queue processing time is negligible compared to request processing time.

Josephus problem

Flavius Josephus is a Jewish historian living in the 1st century. According to his account, he and his 40 comrade soldiers were trapped in a cave, surrounded by Romans. They chose suicide over capture and decided that they would form a circle and start killing themselves using a step of three. As Josephus did not want to die, he was able to find the safe place, and stayed alive with his comrade, later joining the Romans who captured them.

Can you find the safe place?



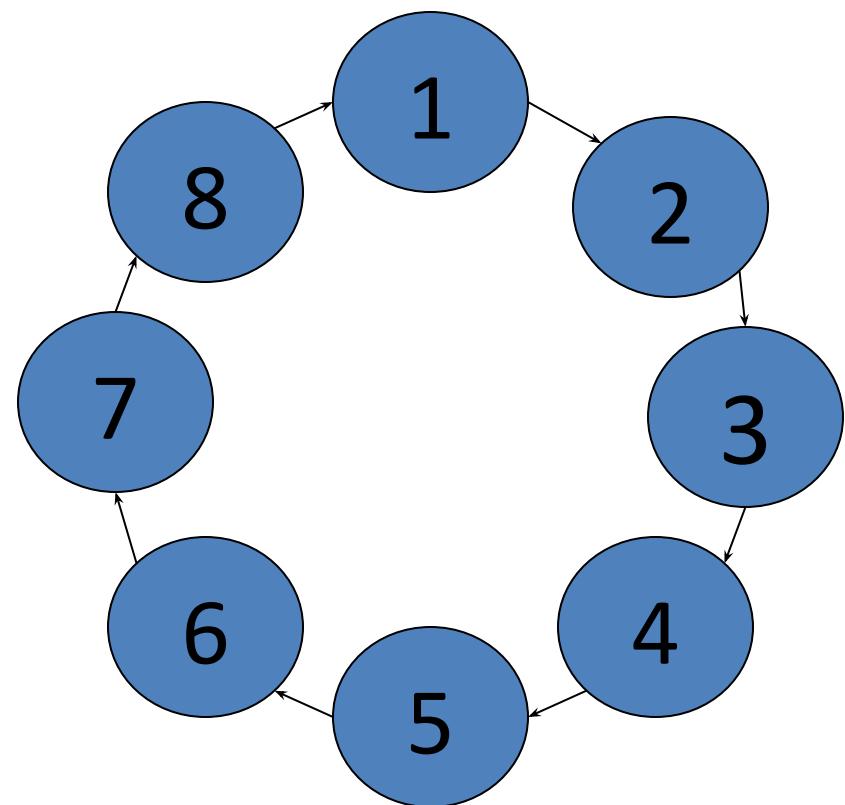
The first algorithm: Simulation

- We can find $f(n)$ using simulation.
 - Simulation is a process to imitate the real objects, states of affairs, or process.
 - We do not need to “kill” anyone to find $f(n)$.
- The simulation needs
 - (1) a **model** to represents “n people in a circle”
 - (2) a way to **simulate** “kill every 2nd person”
 - (3) knowing when to stop

Model n people in a circle

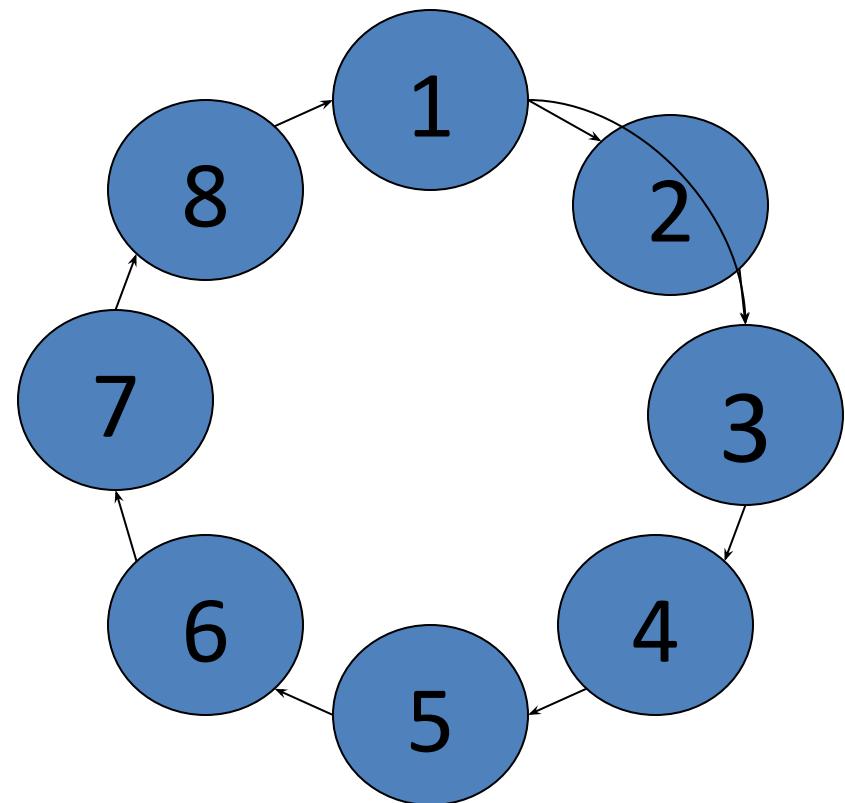
- We can use “data structure” to model it.
- This is called a “circular linked list”.
 - Each node is of some **“struct”** data type
 - Each link is a **“pointer”**

```
struct node {  
    int ID;  
    struct node *next;  
}
```



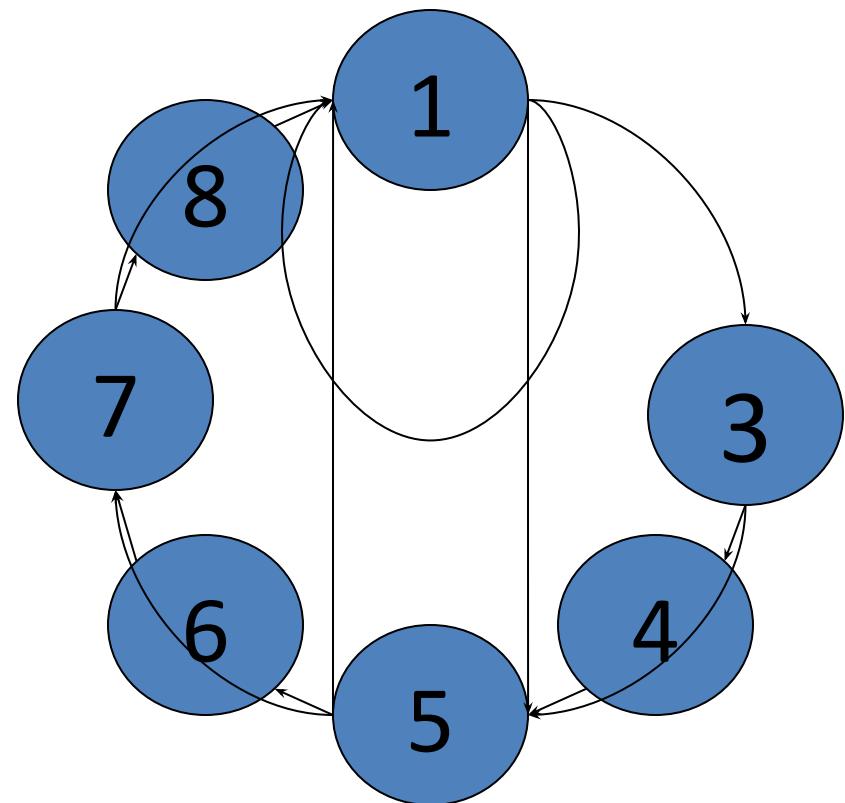
Kill every 2nd person

- Remove every 2nd node in the circular liked list.
 - You need to maintain the circular linked structure after removing node 2
 - The process can continue until ...



Knowing when to stop

- Stop when there is only one node left
 - How to know that?
 - When the `*next` is pointing to itself
 - It's ID is $f(n)$
 - $f(8) = 1$

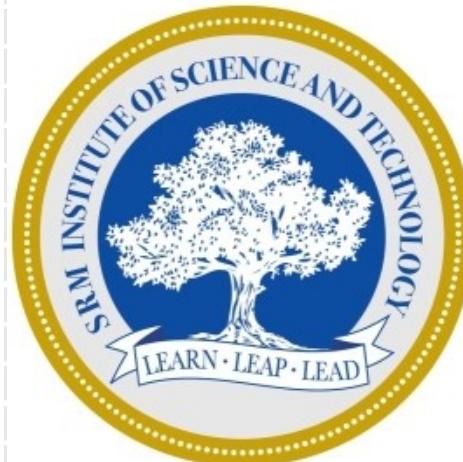




SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC201J – Data Structures and Algorithms

Unit III- STACK & QUEUE



SESSION 13

Priority Queue

- Priority Queue is an extension of a queue with following properties.
- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Priority Queue Example

Example:

X	Y	A	M	B	C	N	O	P	Z
1	1	2	3	2	2	3	3	3	1

Operations Involved

- **insert(item, priority):** Inserts an item with given priority.
- **getHighestPriority():** Returns the highest priority item.
- **pull highest priority element:** remove the element from the queue that has the *highest priority*, and return it.

Implementation of priority queue

- Circular array
- Multi-queue implementation
- Double Link List
- Heap Tree

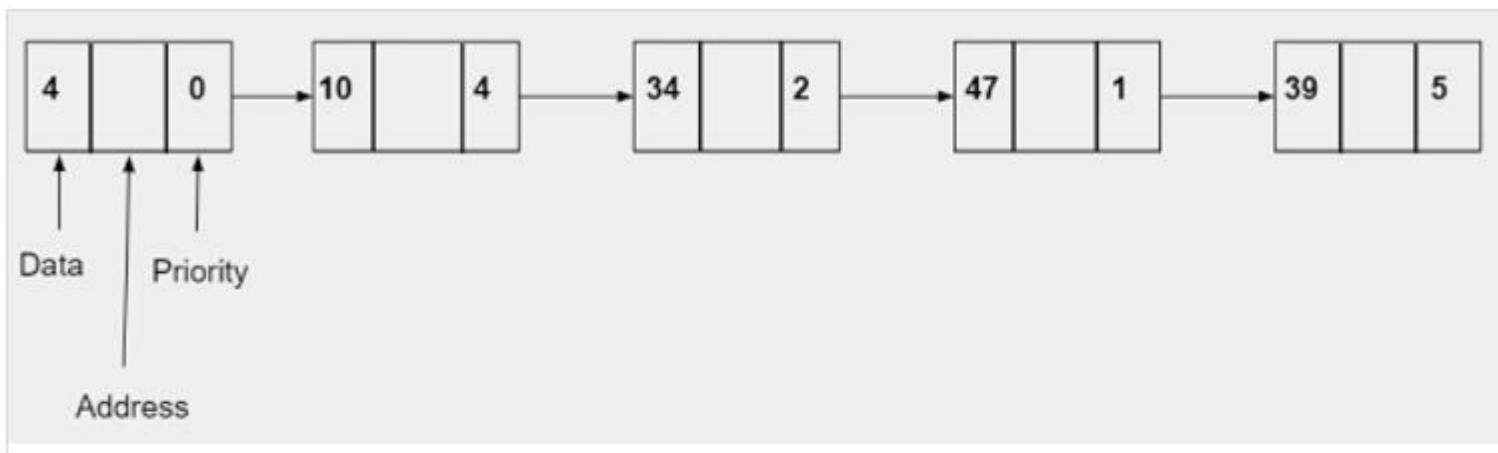
Node of linked list in priority queue

It comprises of three parts

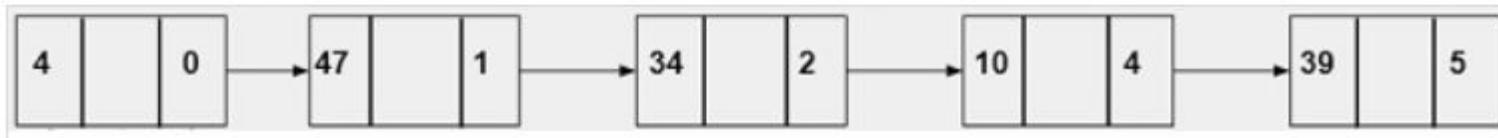
- **Data** – It will store the integer value.
- **Address** – It will store the address of a next node
- **Priority** –It will store the priority which is an integer value. It can range from 0-10 where 0 represents the highest priority and 10 represents the lowest priority.

Example

Input



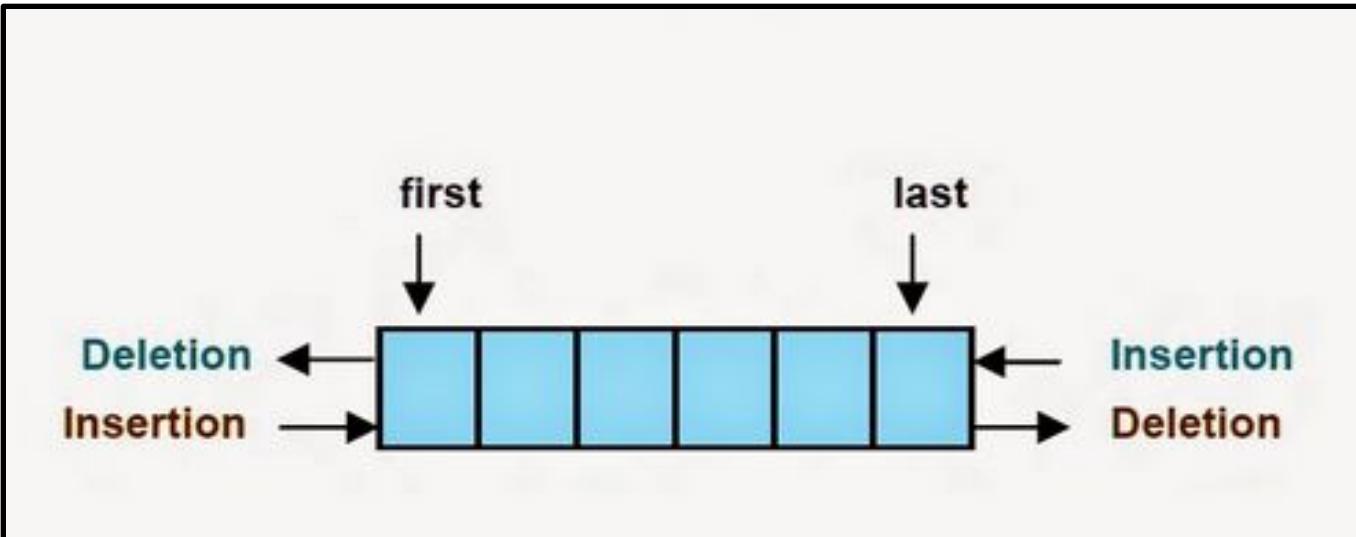
Output



Double Ended Queue

- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**).

Double Ended Queue

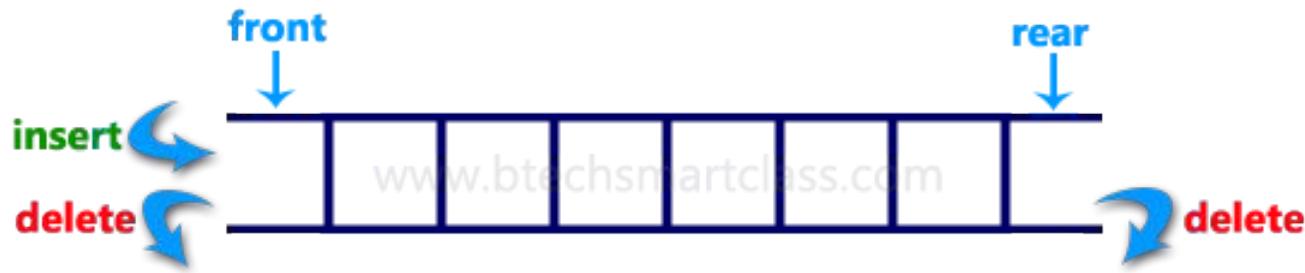


Double Ended Queue can be represented in TWO ways

- Input Restricted Double Ended Queue
- Output Restricted Double Ended Queue

Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Input Restricted Double Ended Queue

Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends





SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

TREES

UNIT - 4



Syllabus

- General Trees , Tree Terminologies
- Tree Representation , Tree Traversal
- Binary Tree Representation , Expression Trees
- Binary Tree Traversal , Threaded Binary Tree
- Binary Search Tree :Construction, Searching Insertion and Deletion
- AVL Trees: Rotations , Insertions
- B-Trees Constructions , B-Trees Search
- B-Trees Deletions, Splay Trees
- Red Black Trees , Red Black Trees Insertion



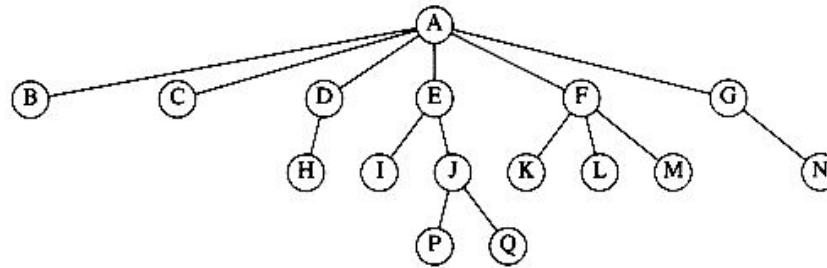
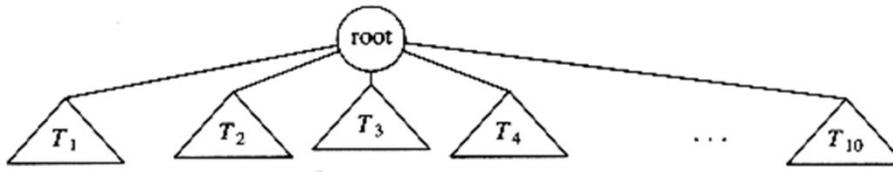
General Trees

- Linear access time for linked lists too high.
- Solution – group data into trees.
- Trees – non linear data structure
- Used to represent data contains a hierarchical relationship among elements example: family, record
- Worst Case time – $O(\log n)$
- Tree can be defined in many ways, eg: recursively

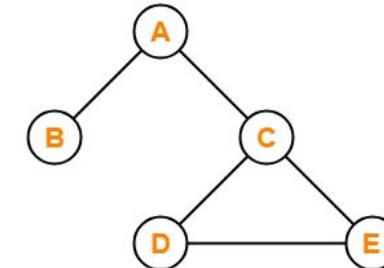


General Trees

- Tree consists of collection of nodes arranged in hierarchical pattern



Tree - Examples



Not a tree



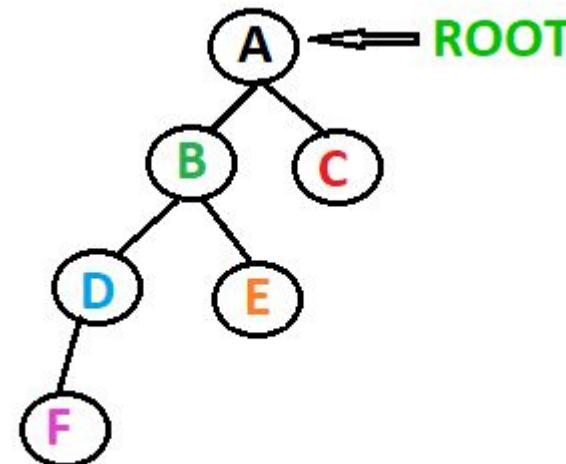
Tree Terminology

- Root
- Edge
- Parent
- Child
- Sibling
- Degree
- Internal node
- Leaf node
- Level
- Height
- Depth
- Subtree



Tree Terminology - Root

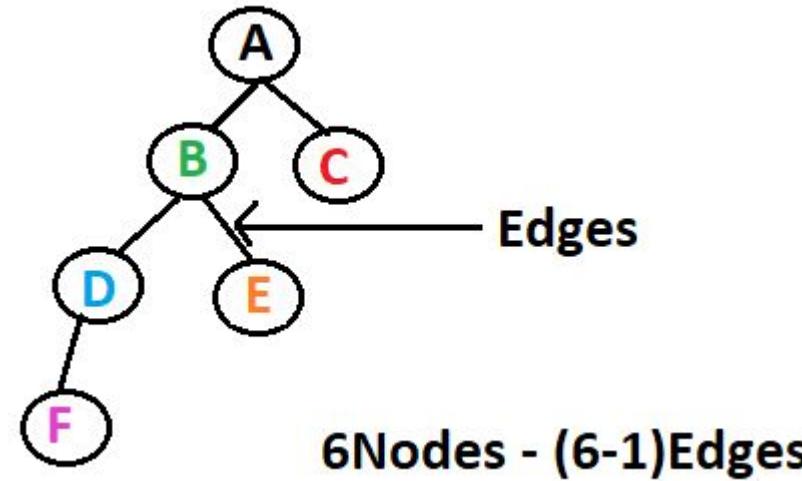
- The starting node of a tree is root node
- Only one root node





Tree Terminology - Edge

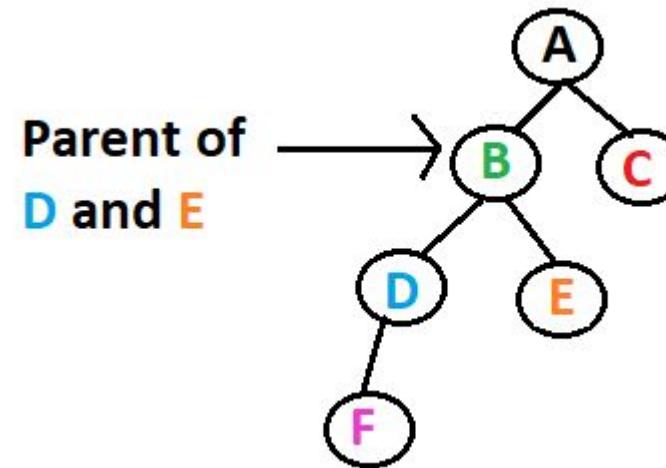
- Nodes are connected using the link called edges
- Tree with n nodes exactly have $(n-1)$ edges





Tree Terminology - Parent

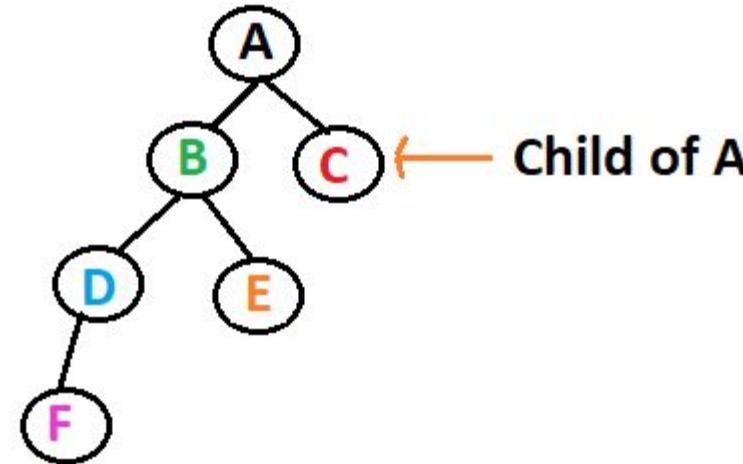
- Node that have children nodes or have branches connecting to other nodes
- Parental node have one or more children nodes





Tree Terminology - Child

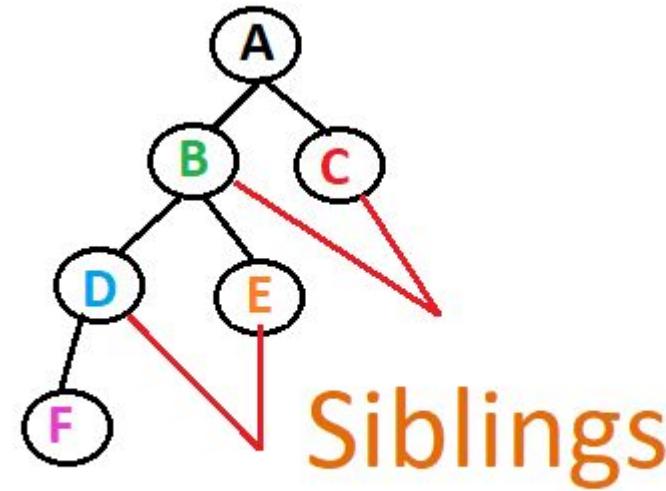
- Node that is descendant of any node is child
- All nodes except root node are child node





Tree Terminology - Siblings

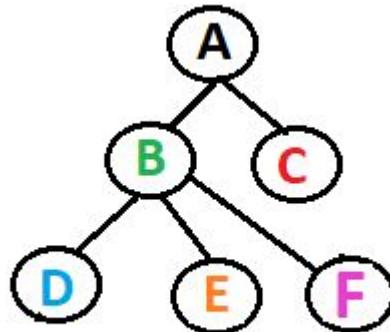
- Nodes with same parents are siblings





Tree Terminology - Degree

- Degree of node – number of children per node
- Degree of tree – highest degree of a node among all nodes in tree

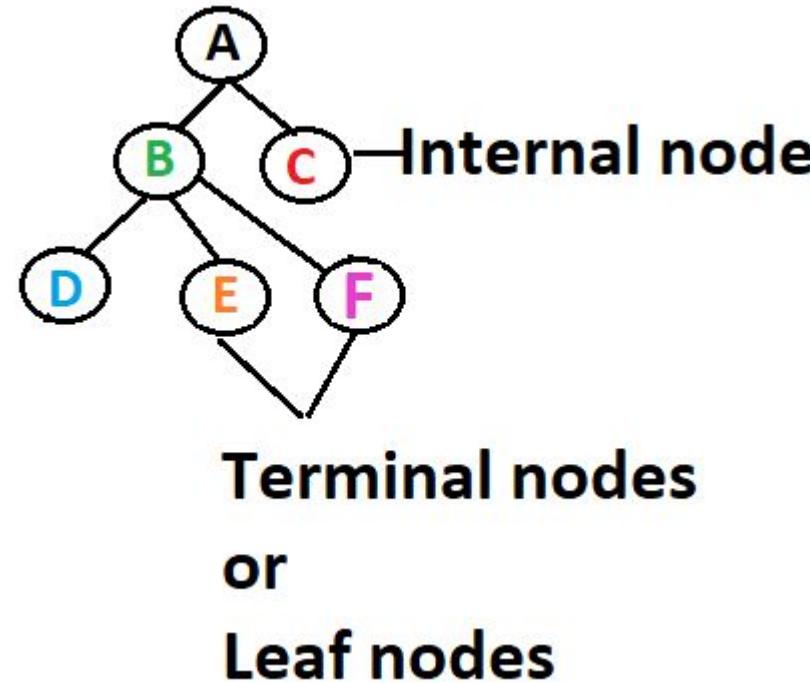


Degree A – 2
Degree B – 3
Degree C – 0
Degree D – 0
Degree E – 0
Degree F – 0
Degree of entire tree - 3



Tree Terminology – Internal Node

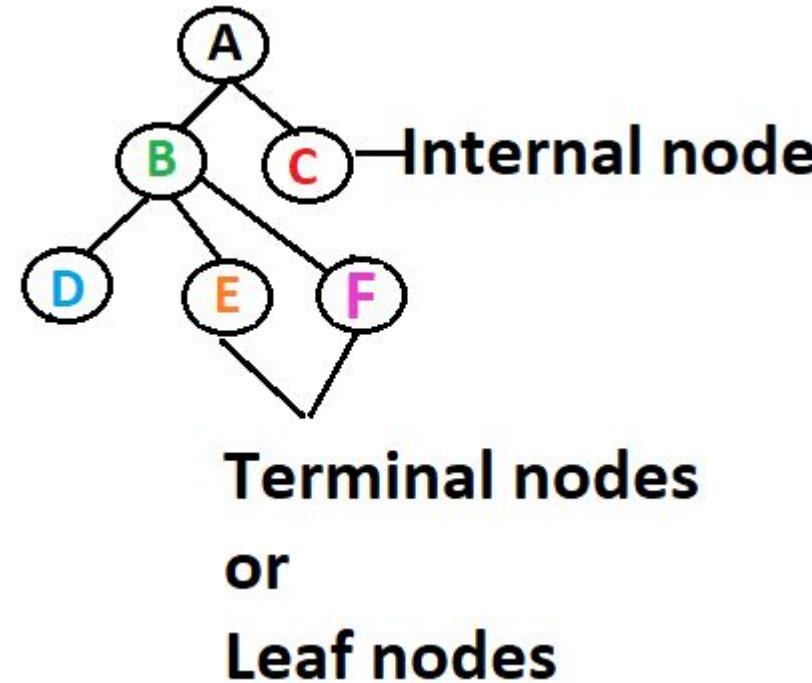
- Node with minimum one child – internal node
- Also known as **non – terminal nodes**





Tree Terminology – Leaf Node

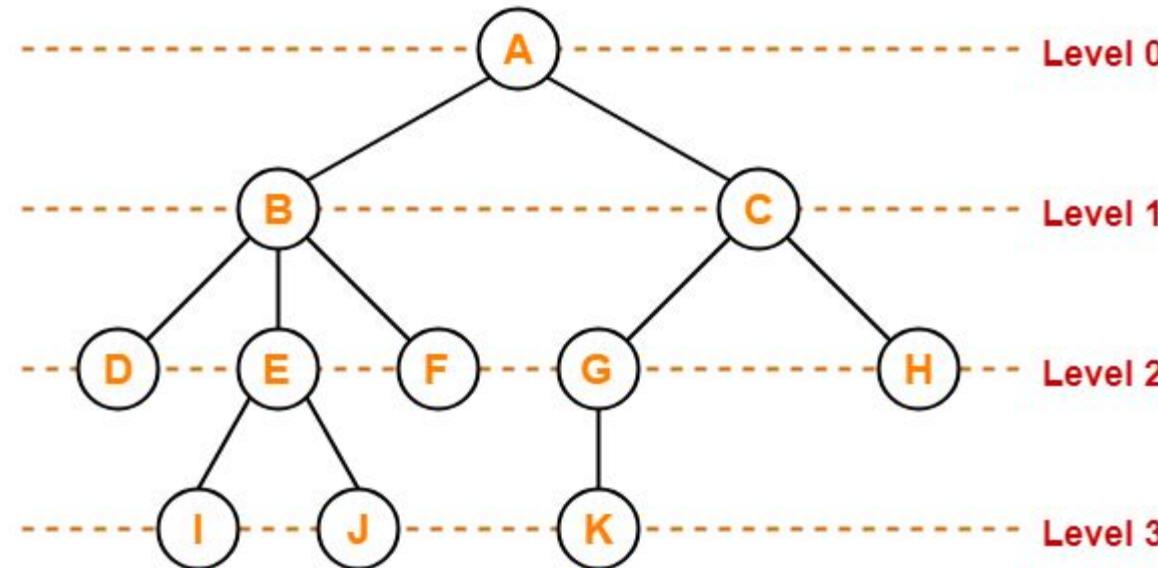
- Node with no child – Leaf node
- Also known as **External nodes or Terminal nodes**





Tree Terminology – Level

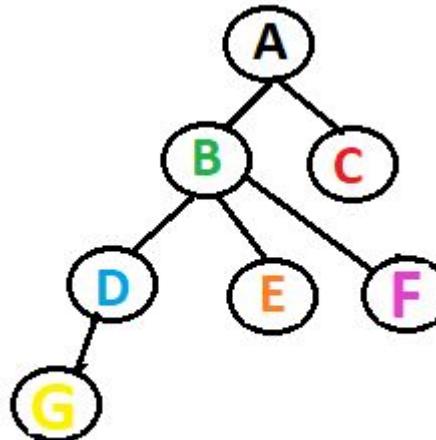
- Each step of tree is level number
- Starting with root as 0





Tree Terminology – Height

- Number of edges in longest path from the node to any leaf
- Height of any leaf node is 0
- Height of Tree = Height of Root node

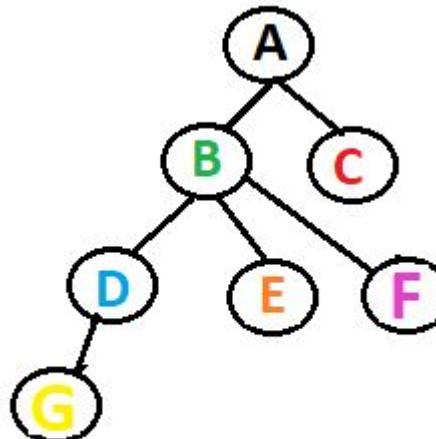


Height A – 3
Height B – 2
Height D – 1
Height C,G,E,F – 0
Height of tree - 3



Tree Terminology – Depth

- Number of edges from root node to particular node is Depth
- Depth of root node is 0
- Depth of Tree = Depth of longest path from root to leaf



Depth A – 0

Depth B ,C – 1

Depth D, E, F – 2

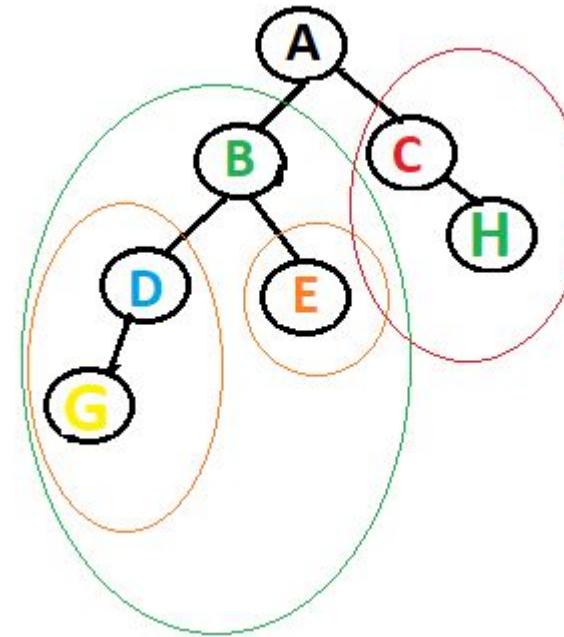
Depth G – 3

Depth of tree - 3



Tree Terminology – Subtree

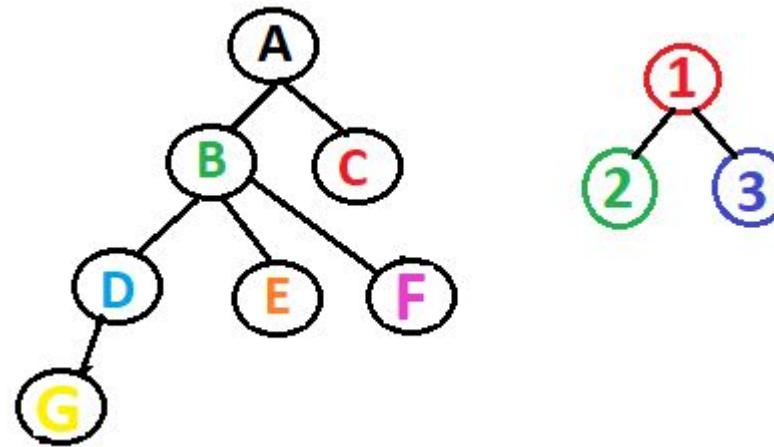
- Each child of a tree forms a subtree recursively





Tree Terminology – Forest

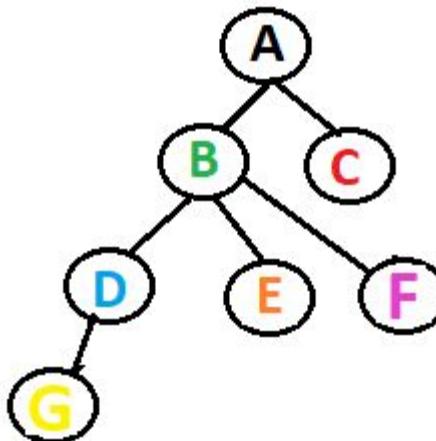
- Set of disjoint trees





Tree Terminology - Path

- A path from node a^1 to a^k is defined as a sequence of nodes a^1, a^2, \dots, a^k such that a^i is the parent of a^{i+1} for $1 \leq i < k$

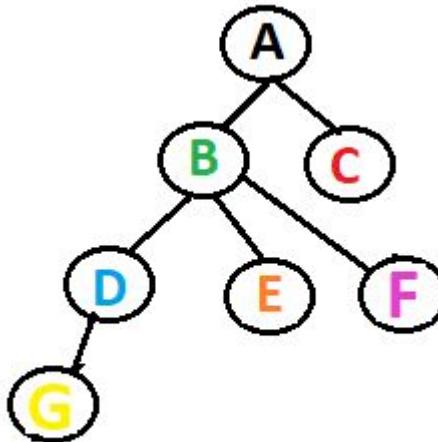


Path A-G: A,B,D,G



Tree Terminology – length of Path

- Number of edges in the path



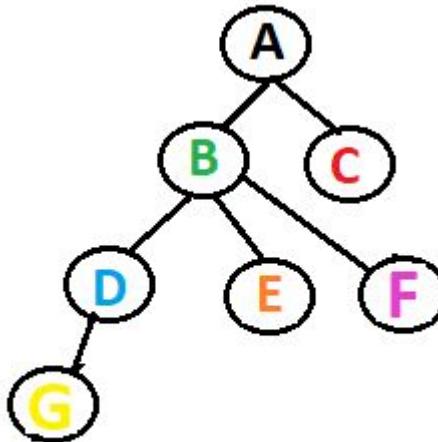
Path A-G: A – B – D – G

Length of path A-G : 3



Tree Terminology – Ancestor & Descendant

- If there is a path from A to B, then A is an ancestor of B and B is a descendant of A



Path A-G: A – B – D – G

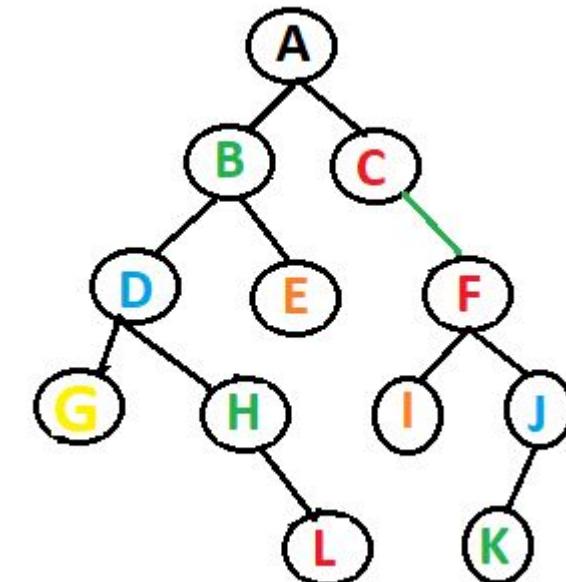
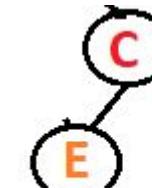
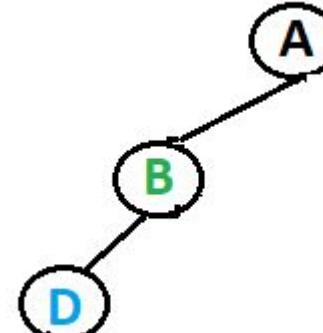
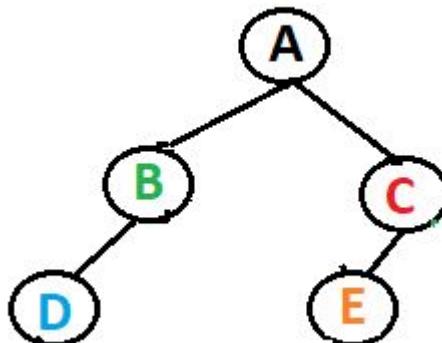
A – Ancestor for G, B, D...

G – Descendant of D,B,A



Binary Tree

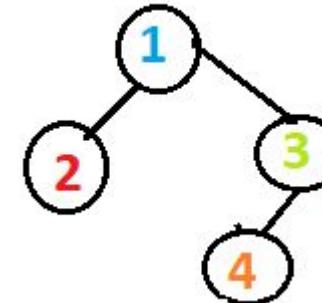
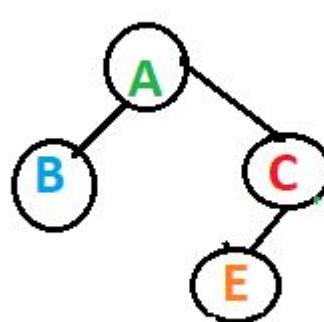
- A binary tree is a data structure specified as a set of so-called node elements. The topmost element in a binary tree is called the root node, and each node has 0, 1, or at most 2 kids.





Similar binary tree

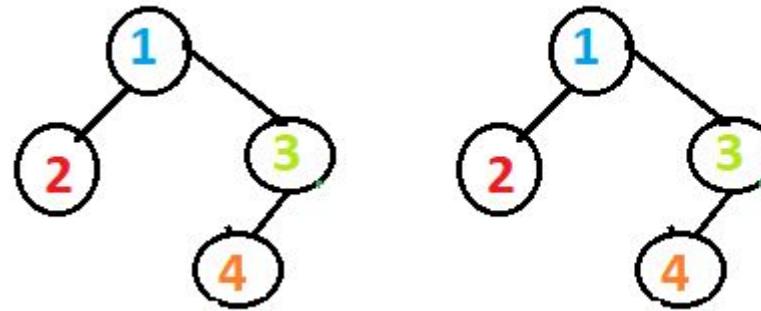
- Tree with same structure





Copies of Binary Tree

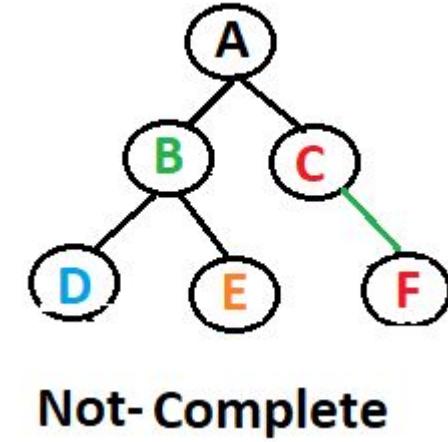
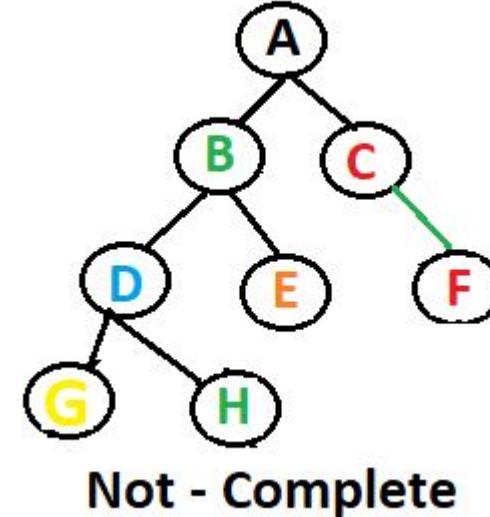
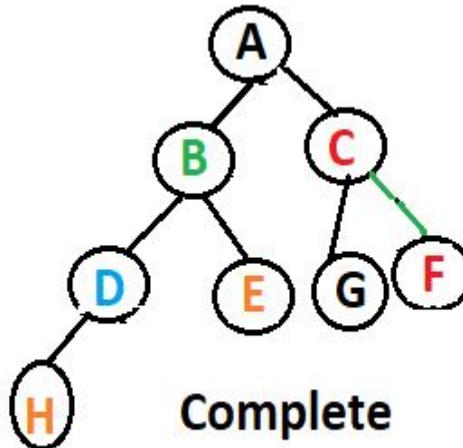
- Same structure and same data node





Complete Binary Tree

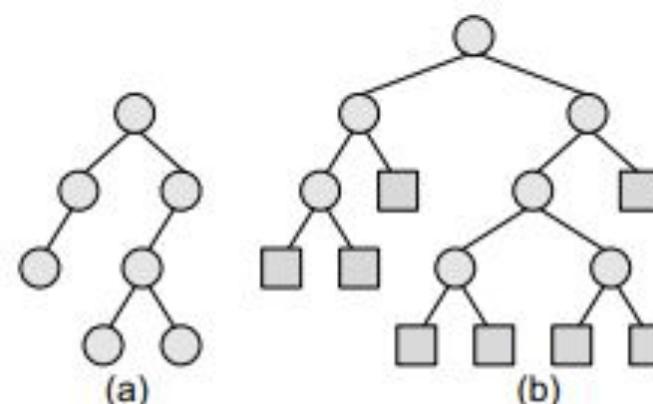
- Except last level, all the nodes need to completely filled
- All nodes filled from left to right





Extended Binary Tree

- Each node will have either two child or zero child
- Nodes with two children – Internal nodes
- Node with no child – External nodes

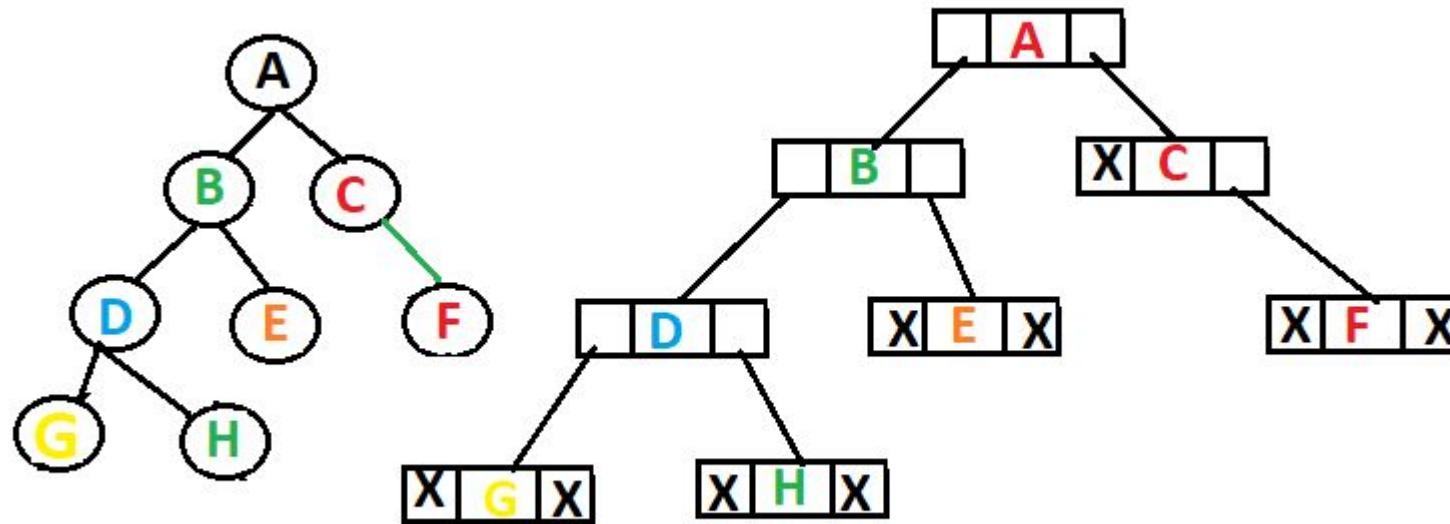


(a) Binary tree and (b) extended binary tree



Representation of Binary Tree

- Each node – three portion – Data portion, left child pointer, Right child pointer





Linked List Representation of Tree

Struct NODE

{

Struct NODE *leftchild;

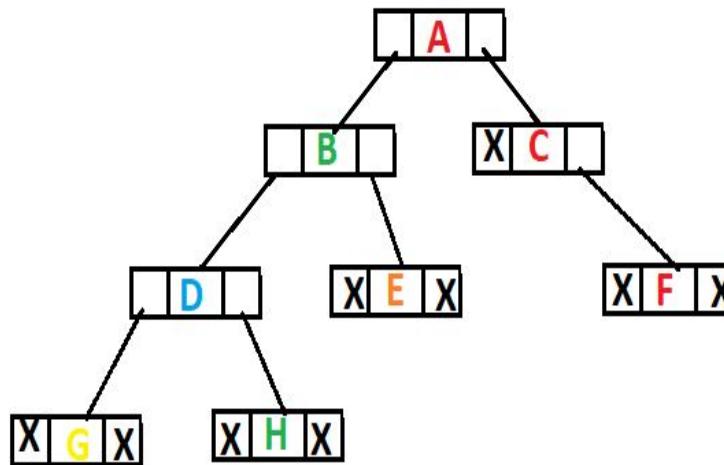
Int value;

Struct NODE *rightchild;

};



Linked List Representation of Tree



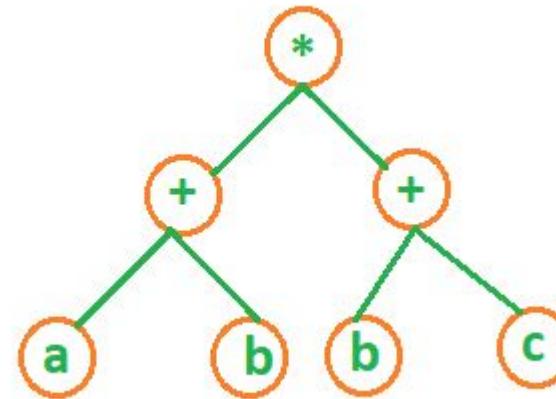
Root - 2

Index	Left child	Node	Right child
1	-1	G	-1
2	5	A	8
3			
4	-1	F	-1
5	10	B	12
6			
7			
8	-1	C	4
9			
10	1	D	14
11			
12	-1	E	-1
13			
14	-1	H	-1



Expression Tree

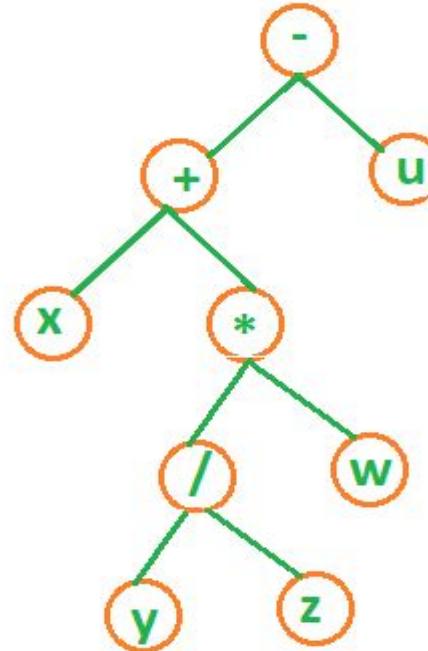
- Used to store algebraic expression
- Example 1: $\text{exp} = (\text{a}+\text{b}) * (\text{b}+\text{c})$





Expression Tree

- Used to store algebraic expression
- Example 1: $\text{exp} = \text{x} + \text{y} / \text{z} * \text{w} - \text{u}$
- Can be written as: $\text{exp} = ((\text{x} + ((\text{y} / \text{z}) * \text{w})) - \text{u})$





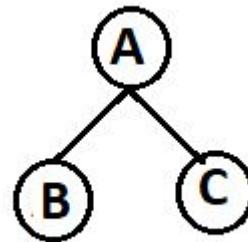
Binary Tree Traversal

- Traversal – visiting all node only once
- Based on the order of visiting :
 - In – order traversal
 - Pre – order traversal
 - Post – order traversal



In-order Traversal

- Traverse left node , visit root node, Traverse right node



B – A – C

Traverse Left node– B

No left child for B subtree

Visit root node B

No right child for B subtree

Visit root node A

Traverse Right node– C

No left child for C subtree

Visit root node C

No right child for C subtree



In-order Traversal

Algorithm: Inorder(Tree)

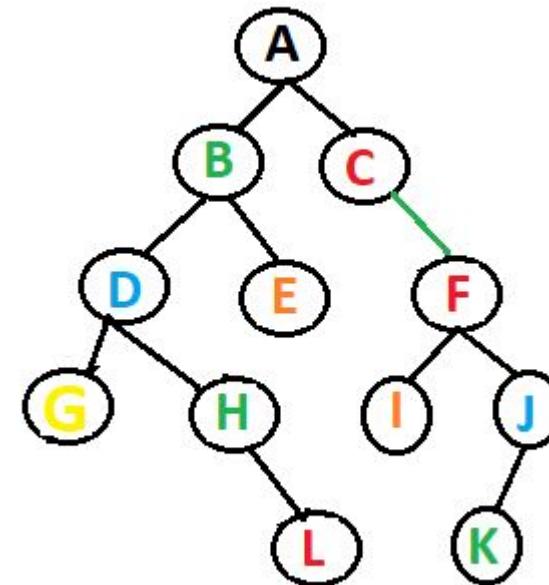
1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End



In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

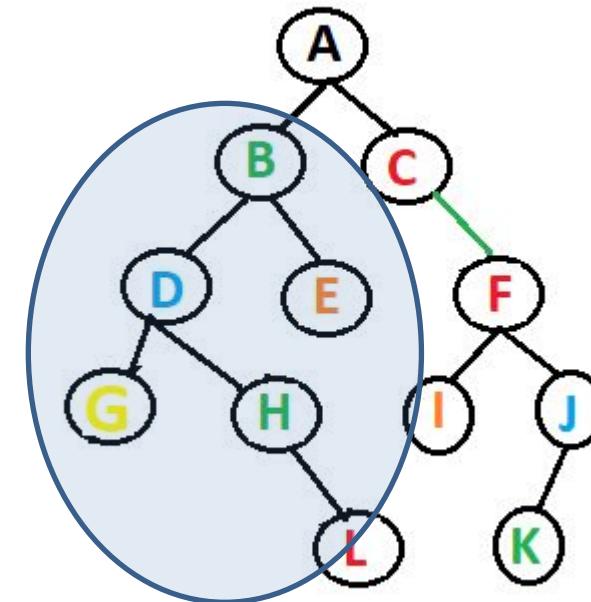




In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. **Inorder(Tree->left)**
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

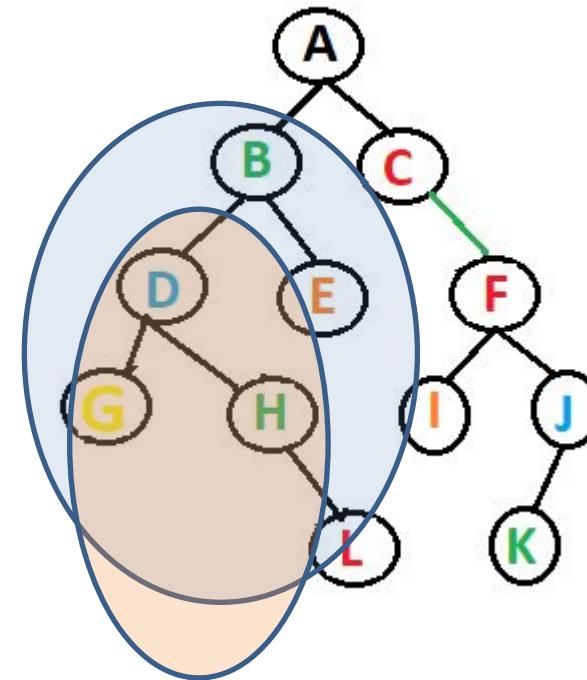




In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. **Inorder(Tree->left)**
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

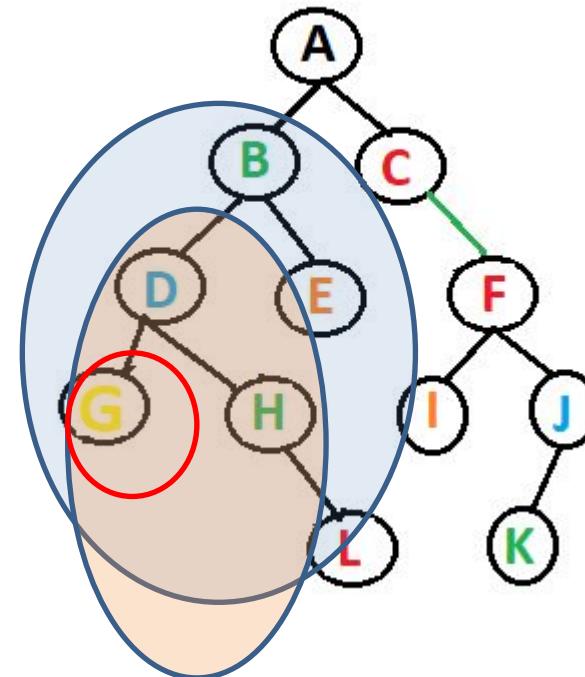




In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End



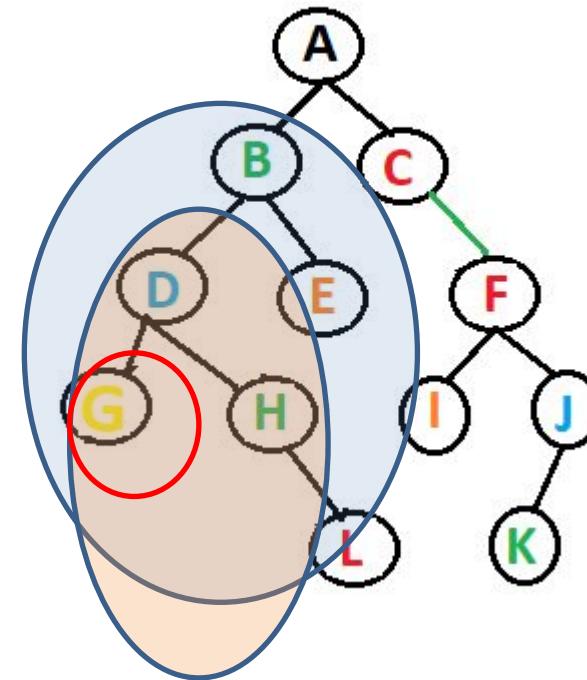


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G,



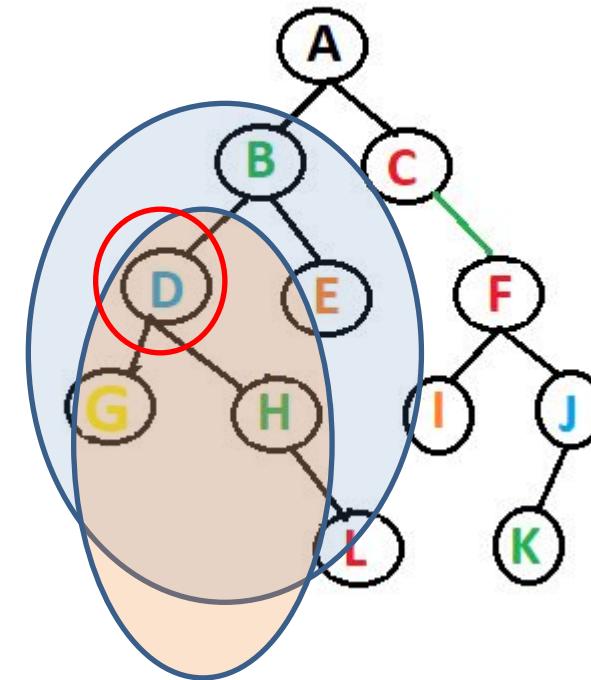


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D,



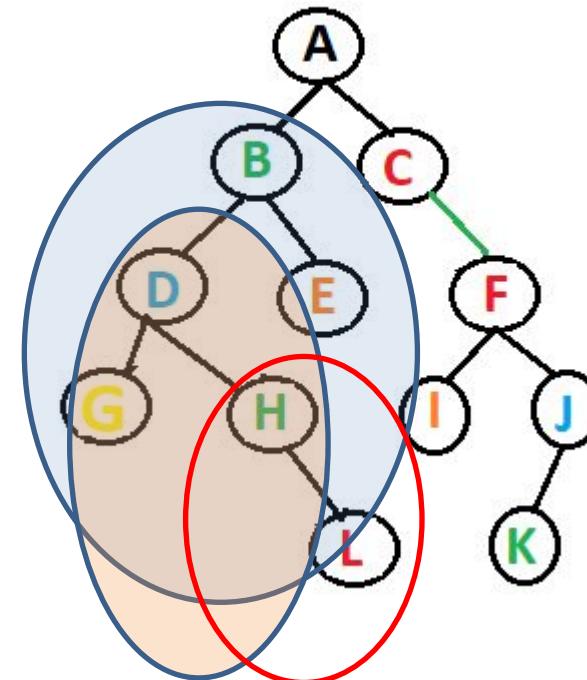


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D,



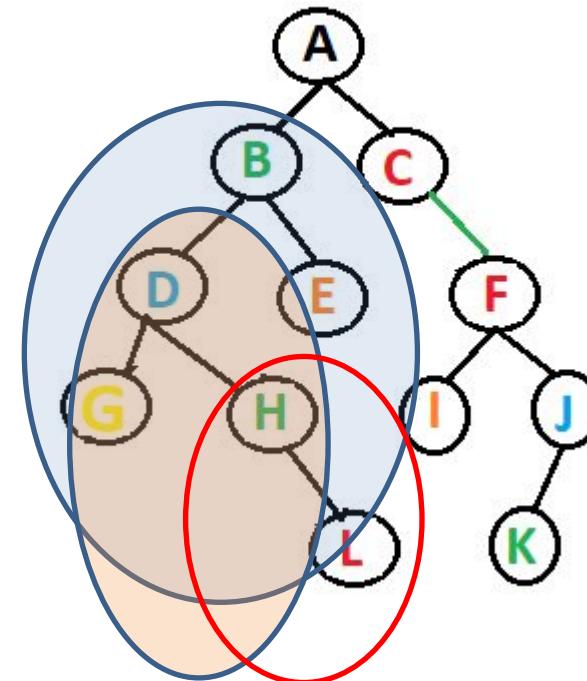


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H,



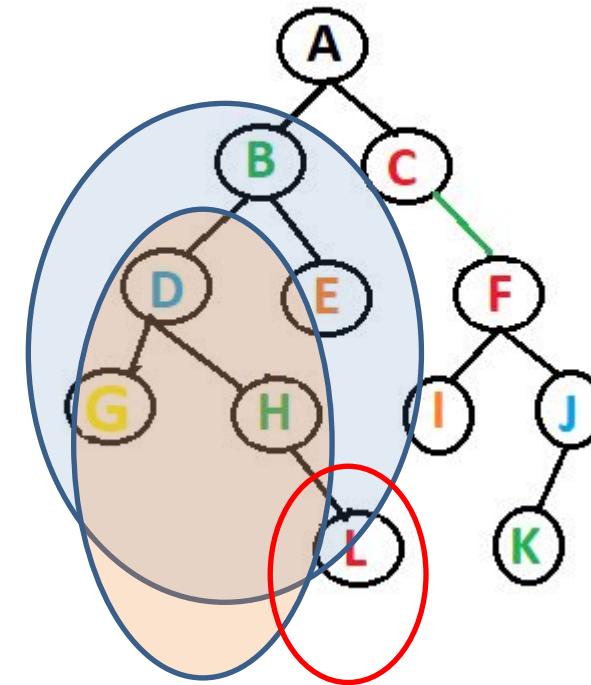


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L,



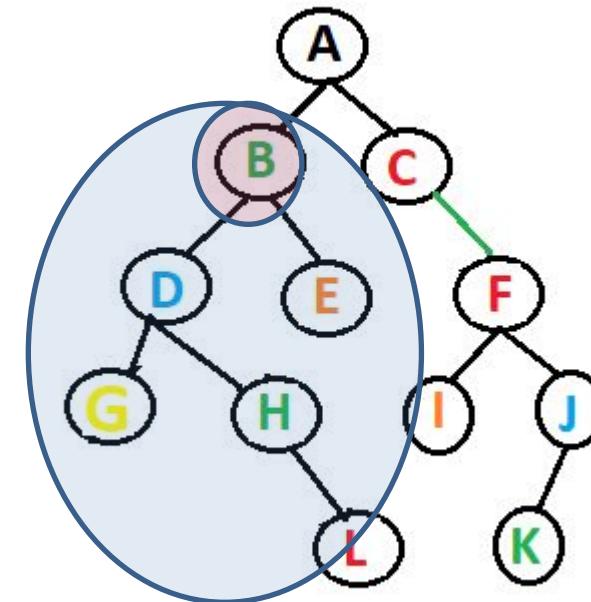


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B



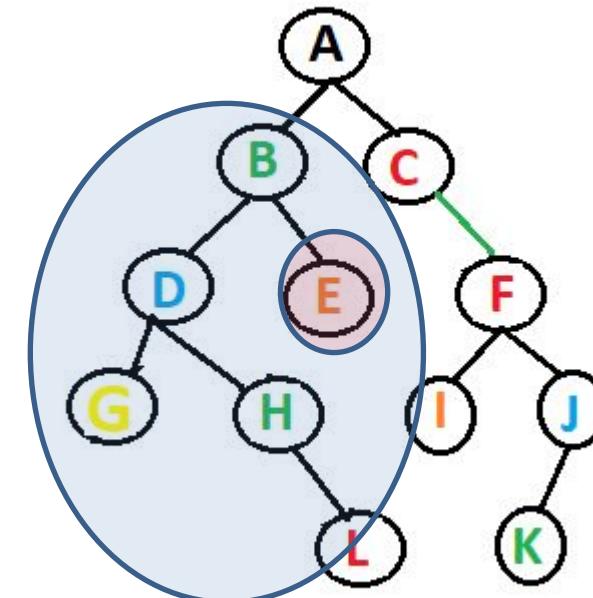


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E



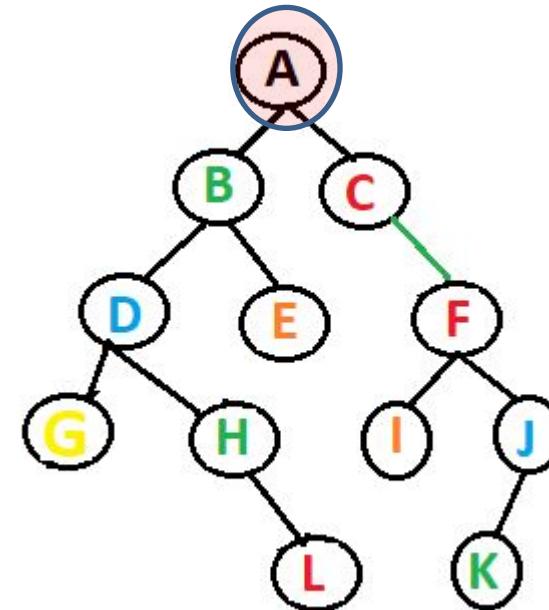


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A,



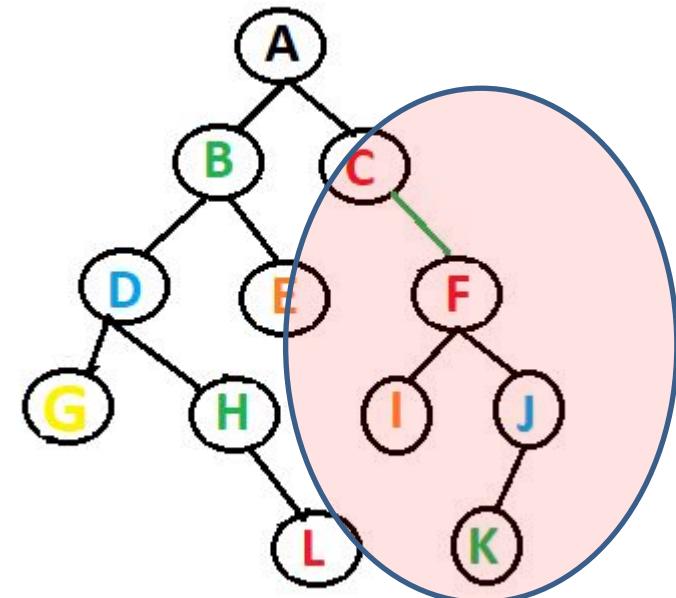


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A,



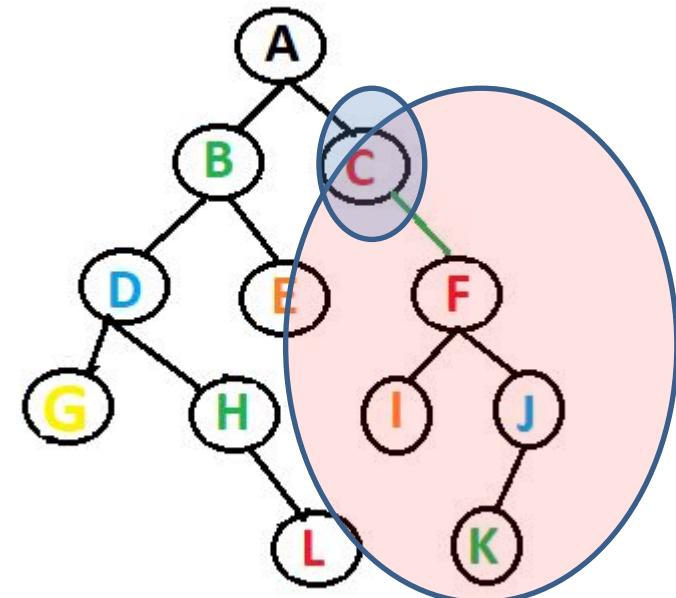


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C



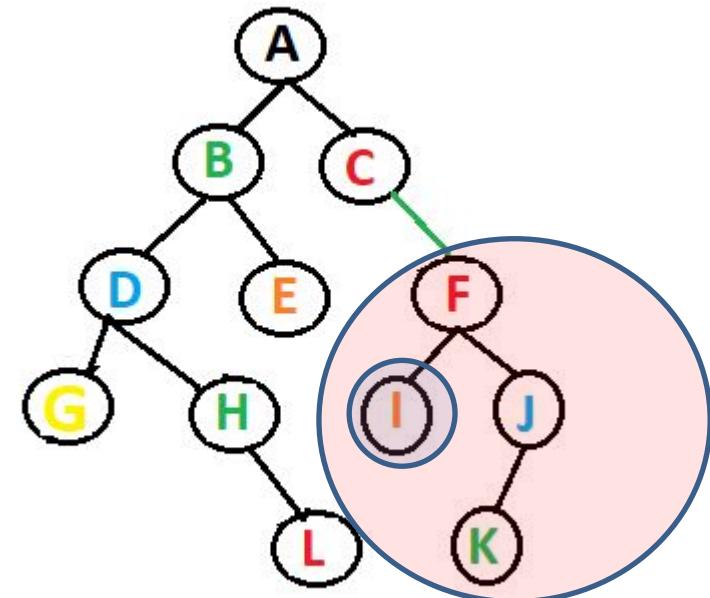


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. **Inorder(Tree->left)**
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,



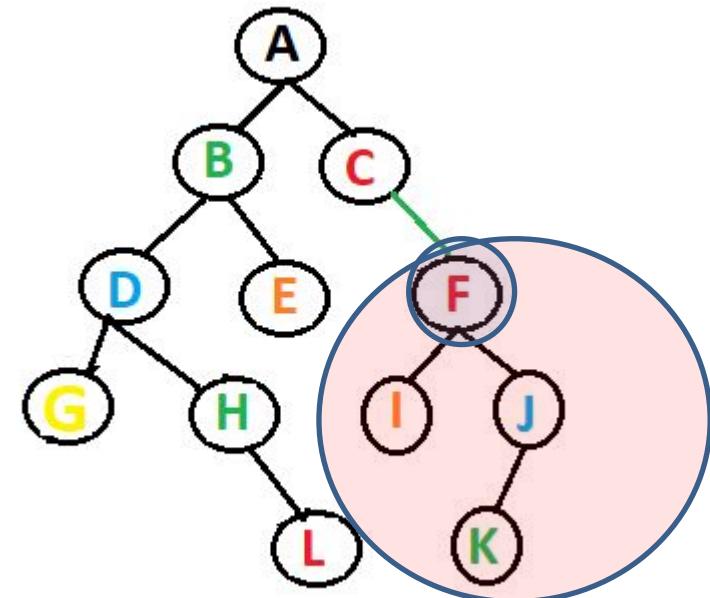


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,
F,



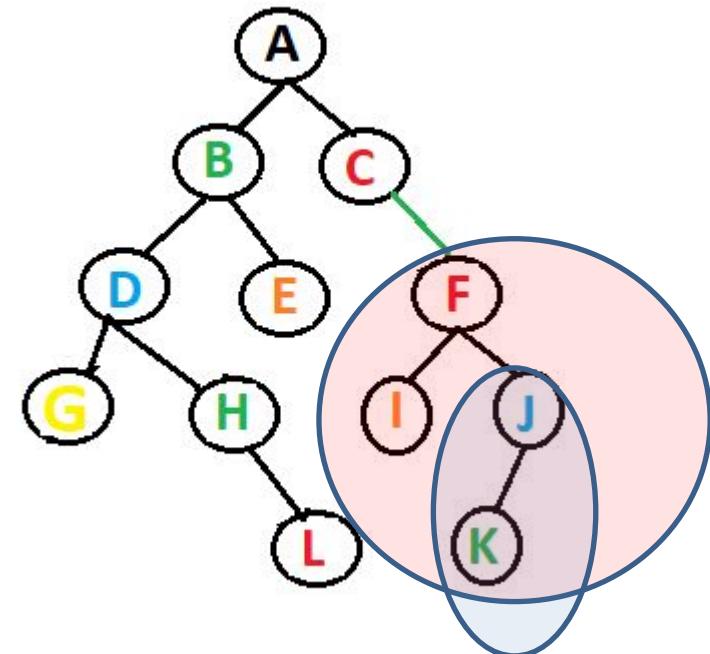


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,
F,



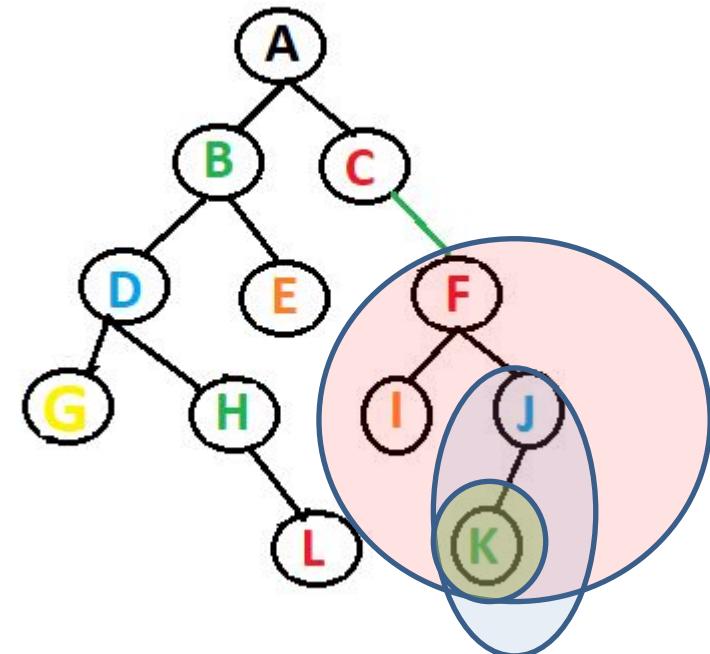


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. **Inorder(Tree->left)**
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,
F, K,



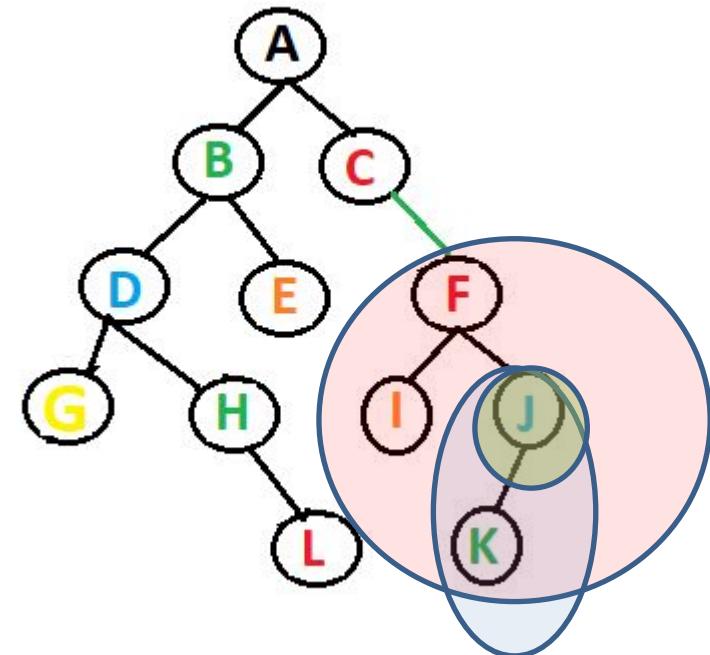


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,
F, K, J



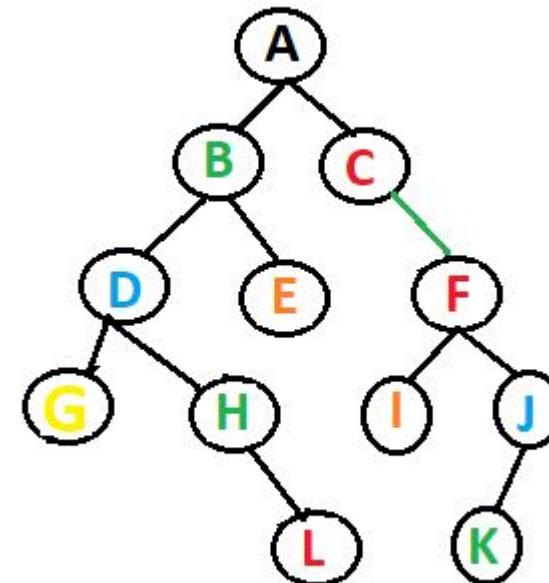


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

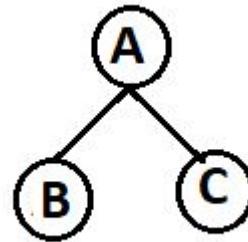
**Result : G, D, H, L, B, E, A, C, I
F, K, J**





Pre – order Traversal

- Visit root node, Traverse left node , Traverse right node



A – B – C

Visit root node A

Traverse Left node– B

Visit root node B

No left child for B subtree

No right child for B subtree

Traverse Right node– C

Visit root node C

No left child for C subtree

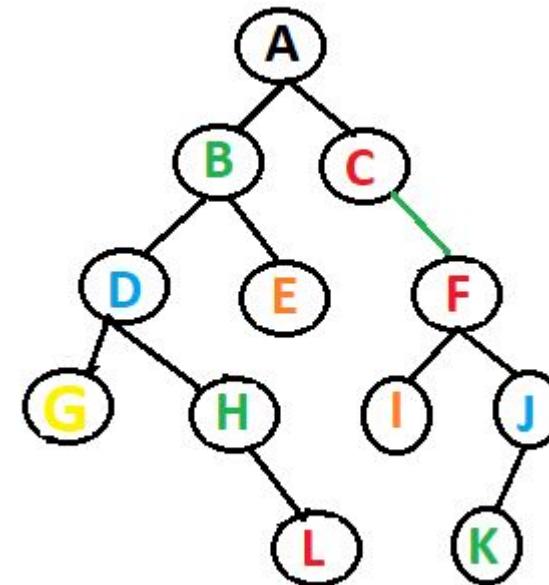
No right child for C subtree



Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End



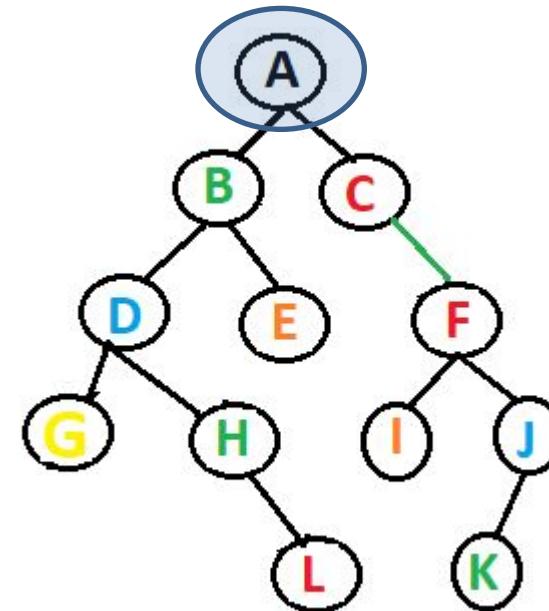


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A,



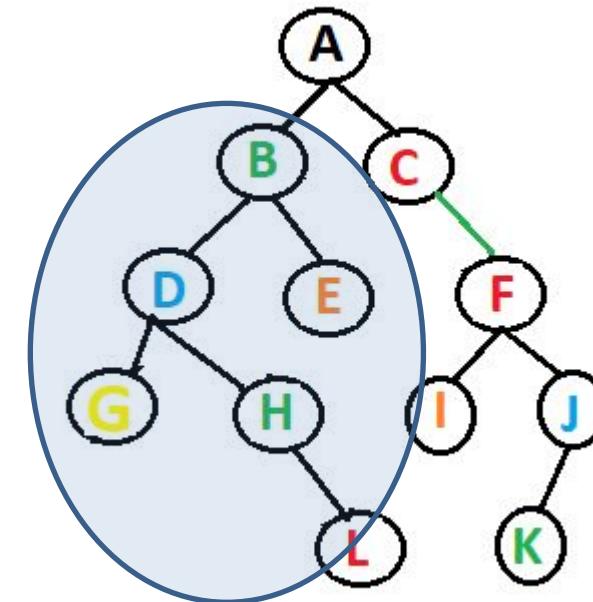


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A,



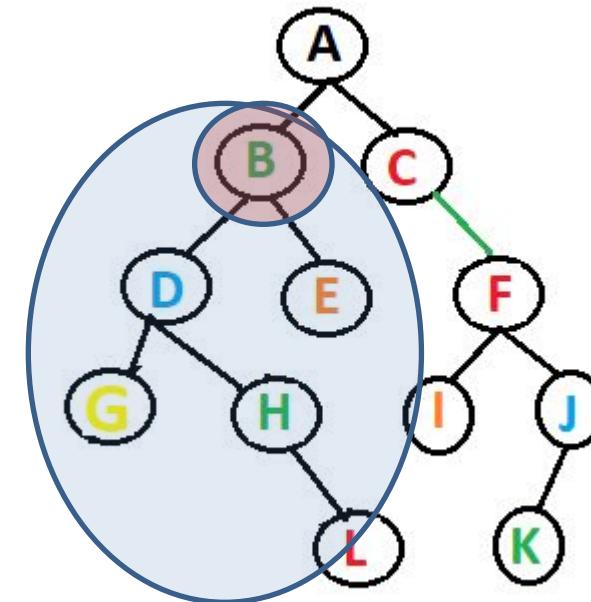


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B,



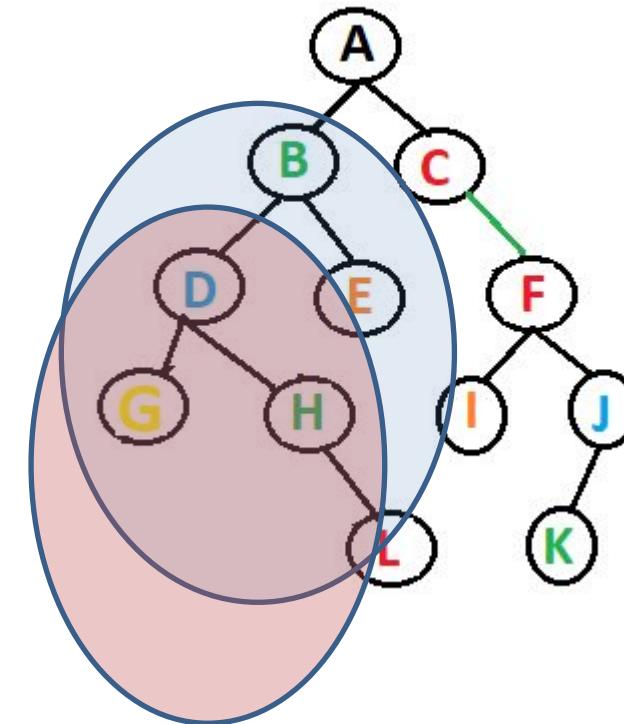


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B,



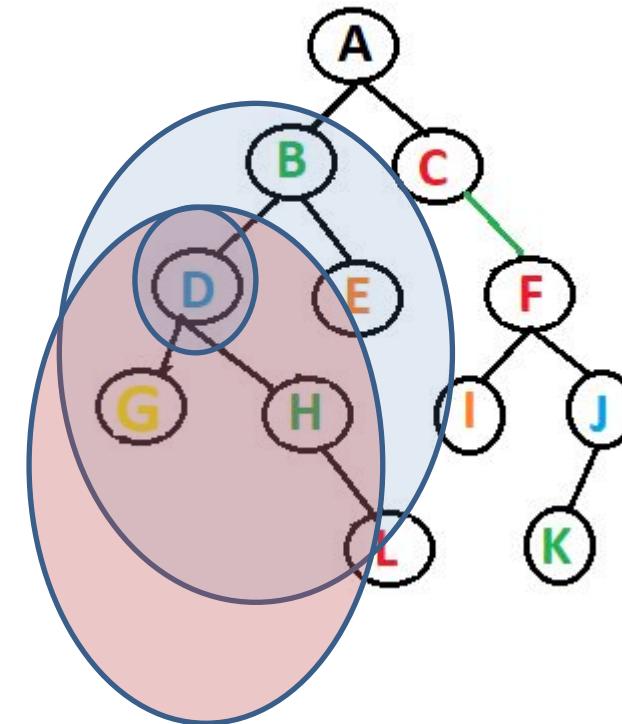


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D,



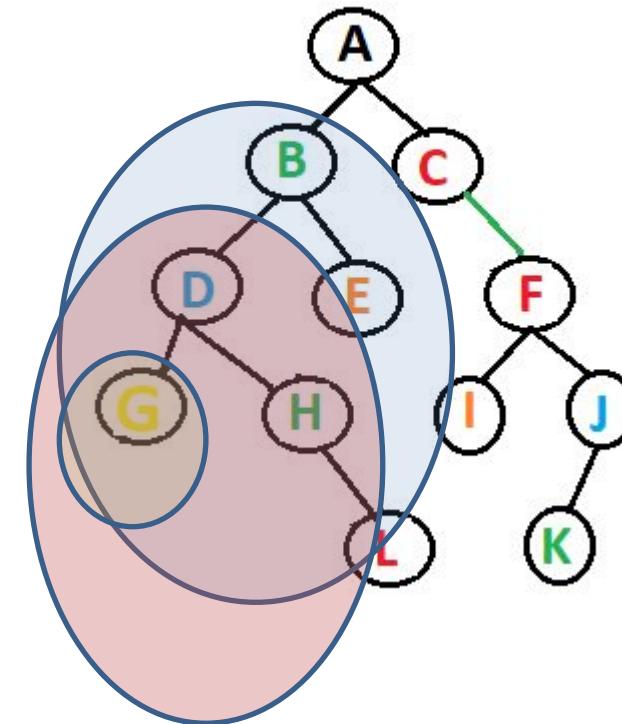


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D,G,



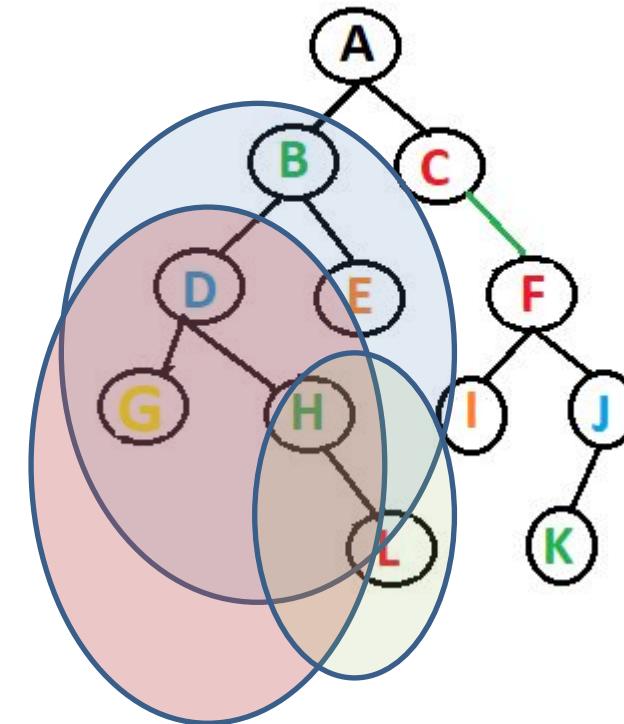


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G,



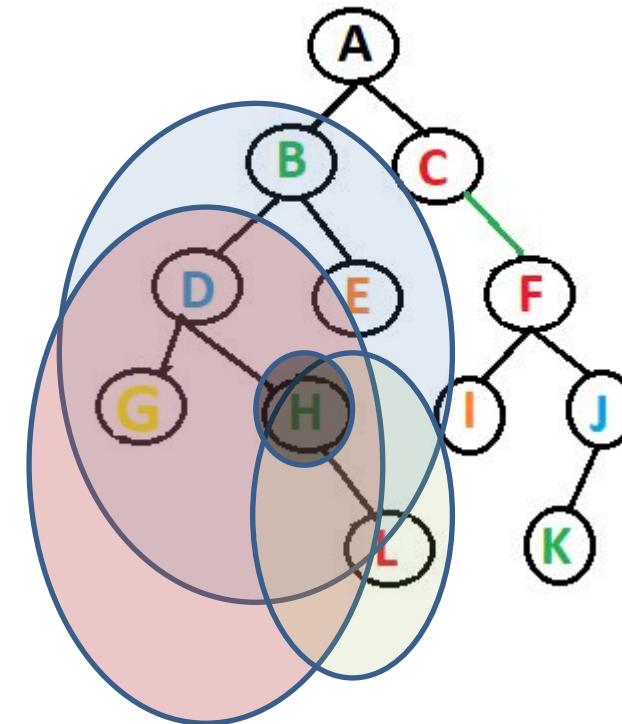


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H,



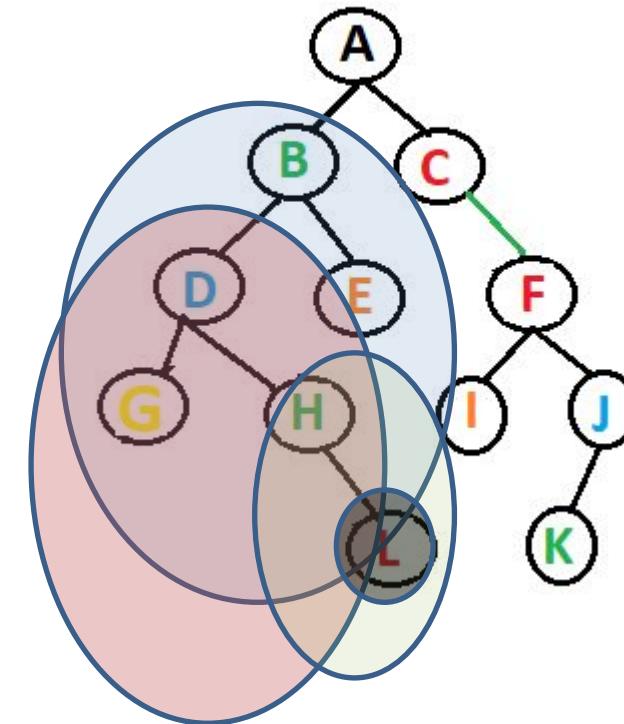


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L,



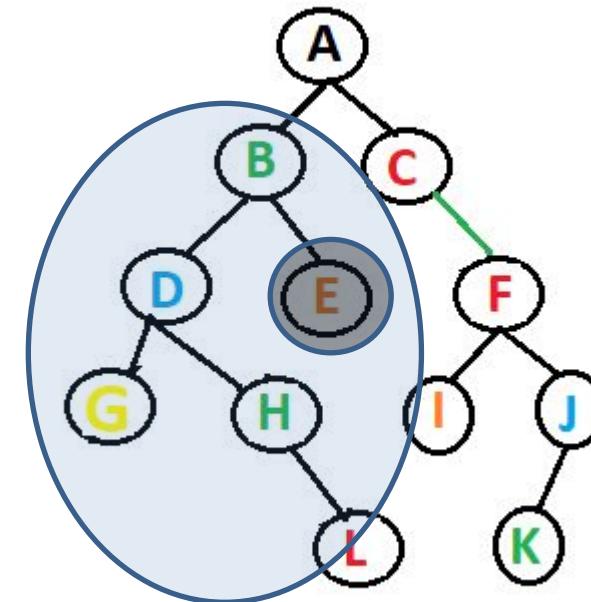


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E,



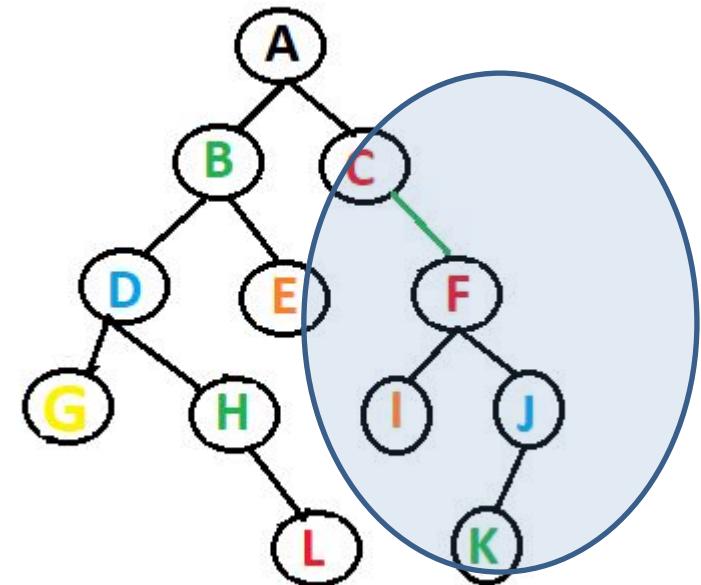


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E,



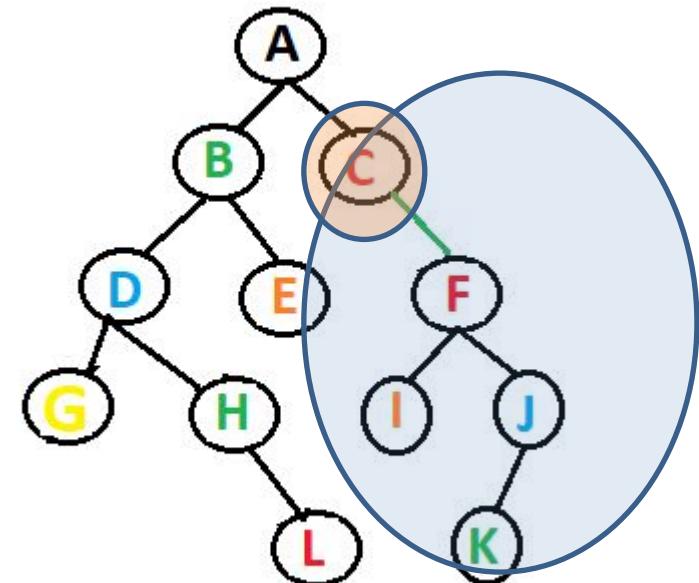


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C,



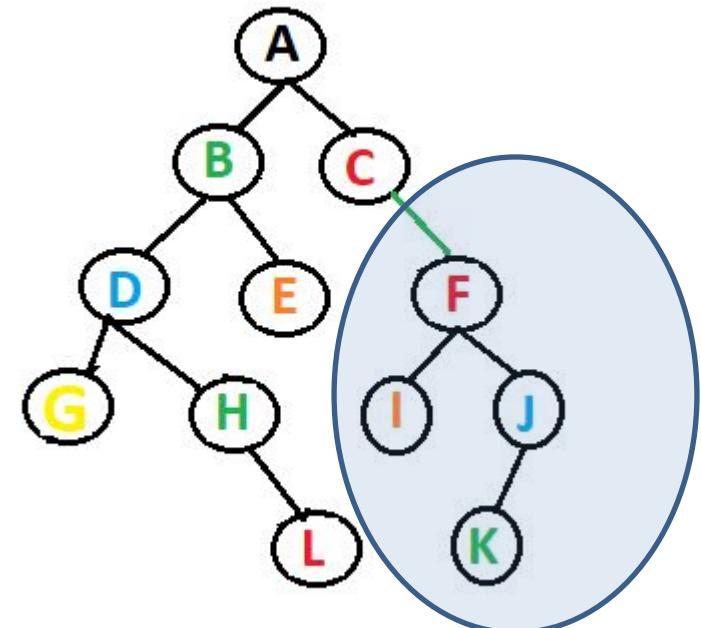


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C,



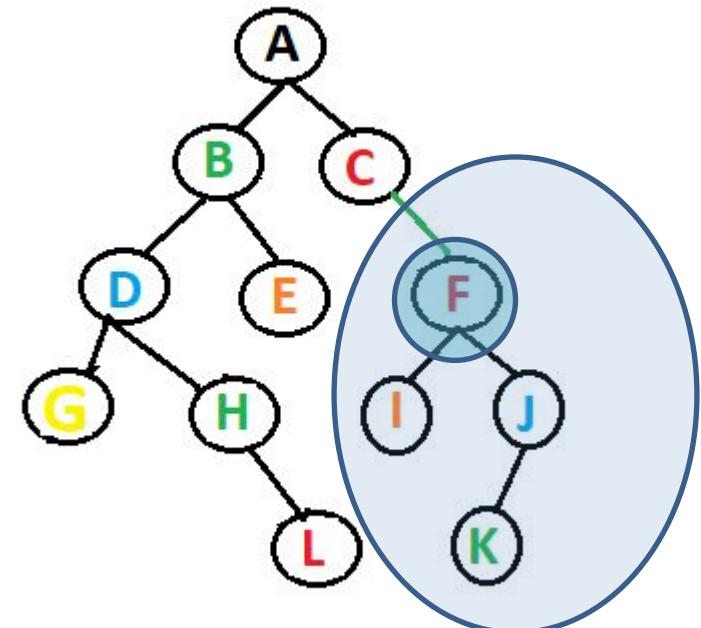


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C, F,



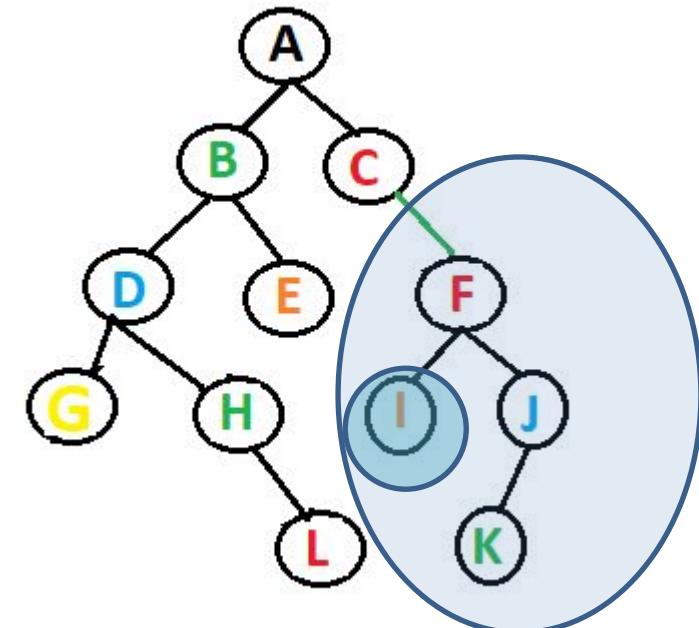


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. **Preorder(Tree->left)**
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C, F,
I,



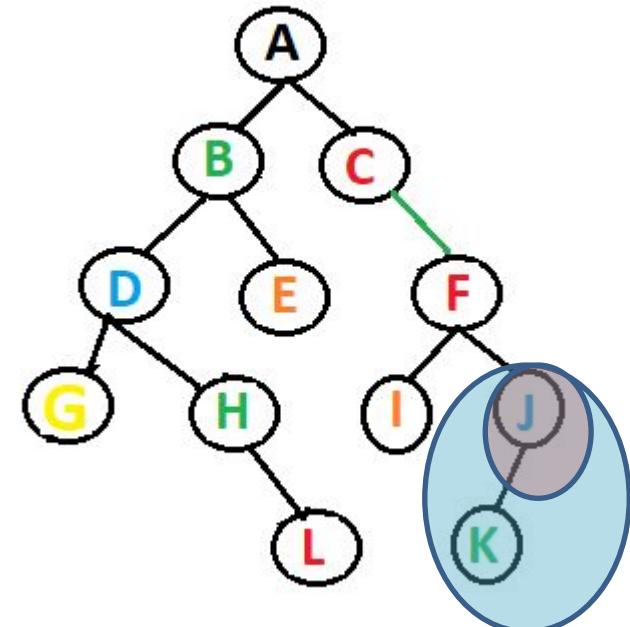


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. **Preorder(Tree->right)**
5. End

Result: A, B, D, G, H, L, E, C, F,
I, J,



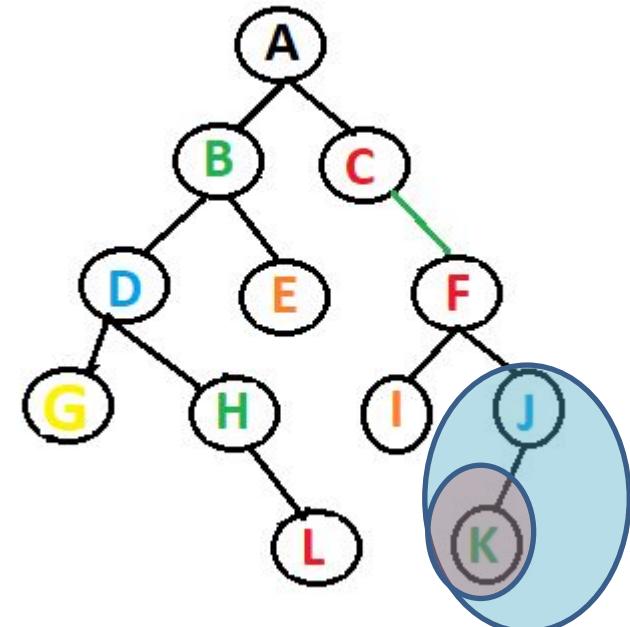


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. **Preorder(Tree->left)**
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C, F,
I, J, K



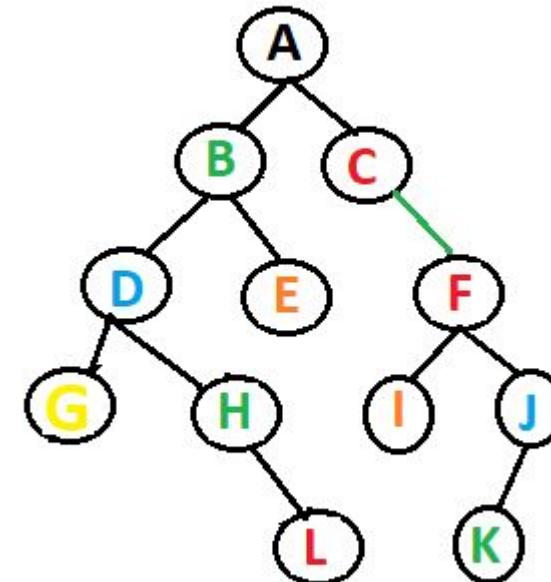


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. **Preorder(Tree->left)**
4. Preorder(Tree->right)
5. End

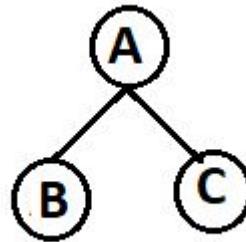
Result: A, B, D, G, H, L, E, C, F
I, J, K





Post– order Traversal

- Traverse left node, Traverse right node , visit root node



B – C – A

Traverse Left node – B

No left child for B subtree

No right child for B subtree

Visit root node B

Traverse Right node– C

No left child for C subtree

No right child for C subtree

Visit root node C

Visit root node A



Post-order Traversal

Algorithm: Postorder(Tree)

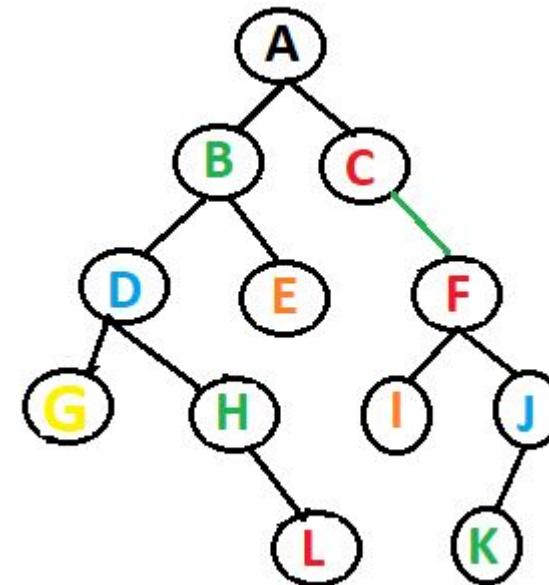
1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End



Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

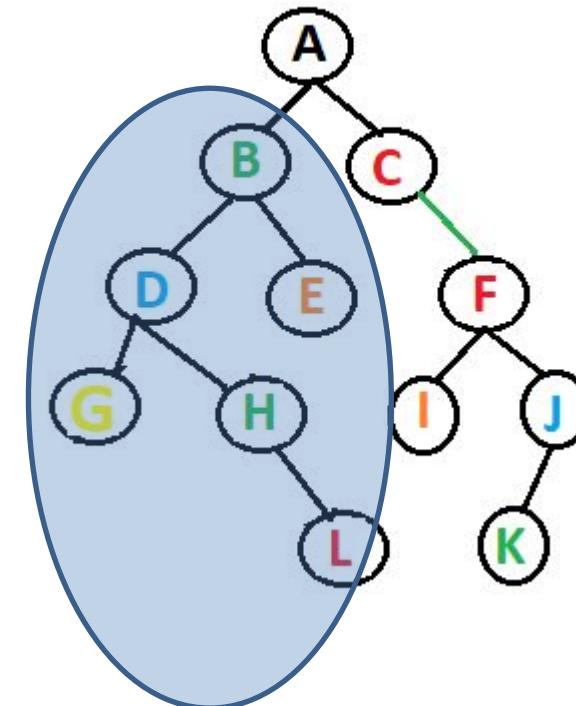




Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

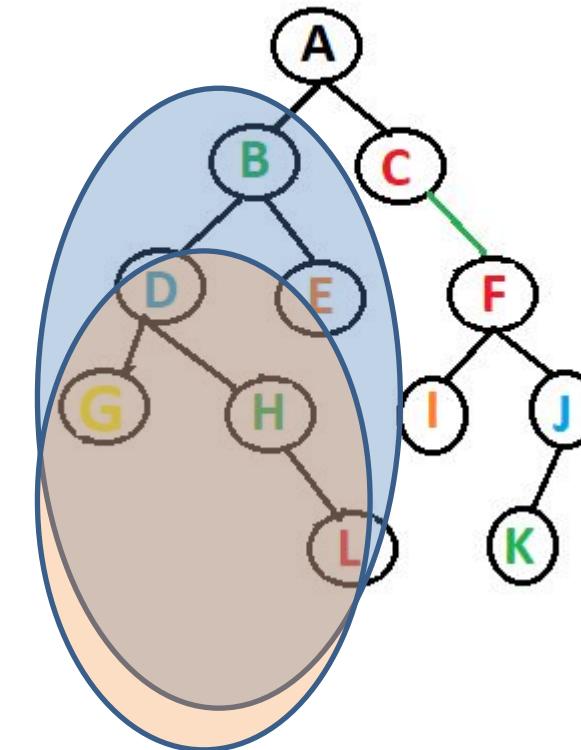




Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End



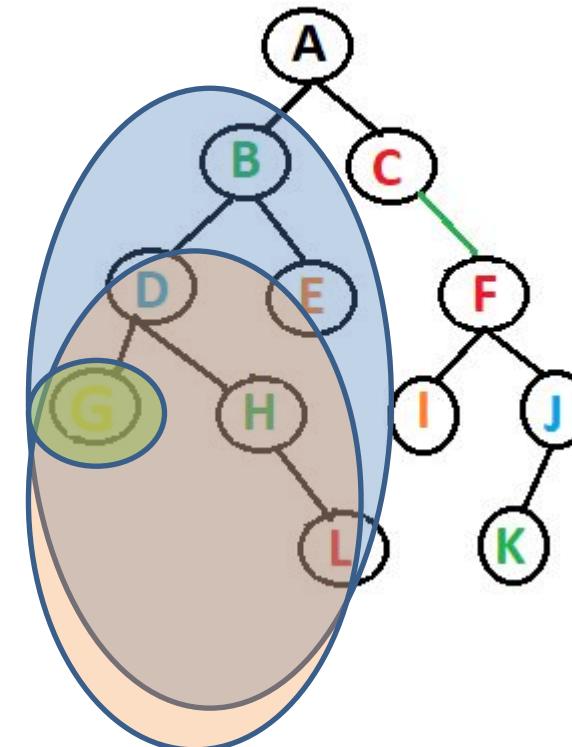


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G,



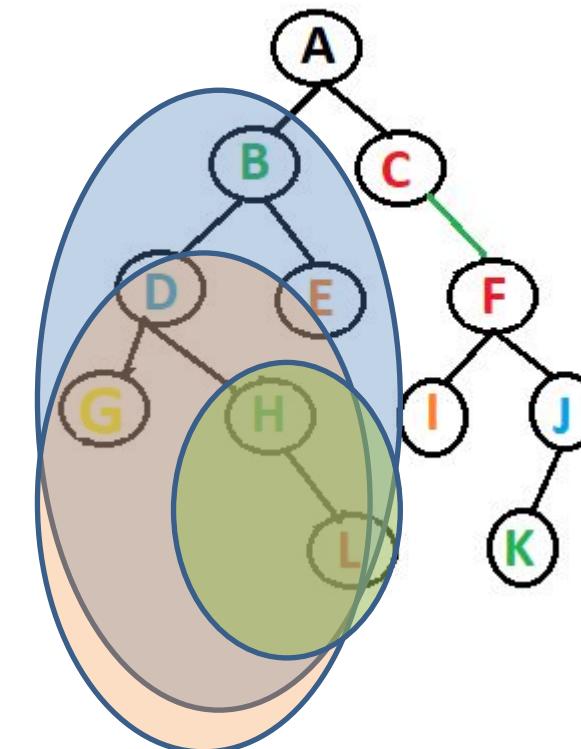


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. **Postorder(Tree->right)**
4. Write(Tree->Data)
5. End

Result : G,



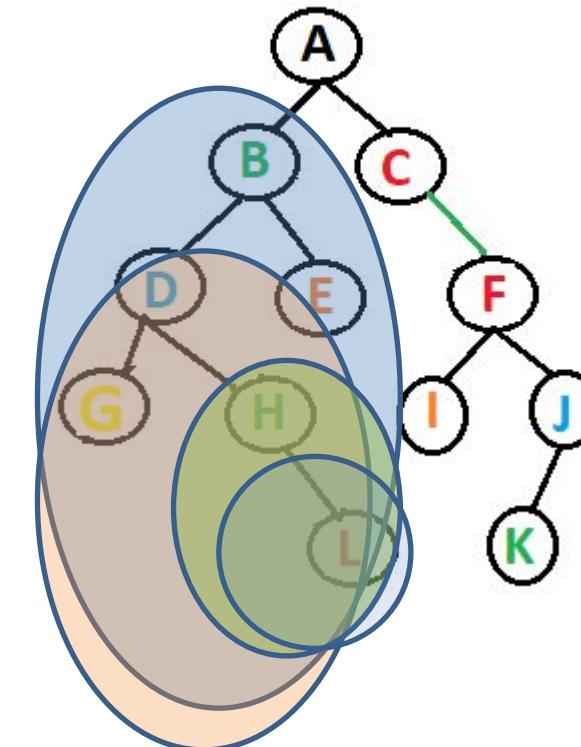


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. **Postorder(Tree->right)**
4. Write(Tree->Data)
5. End

Result : G, L,



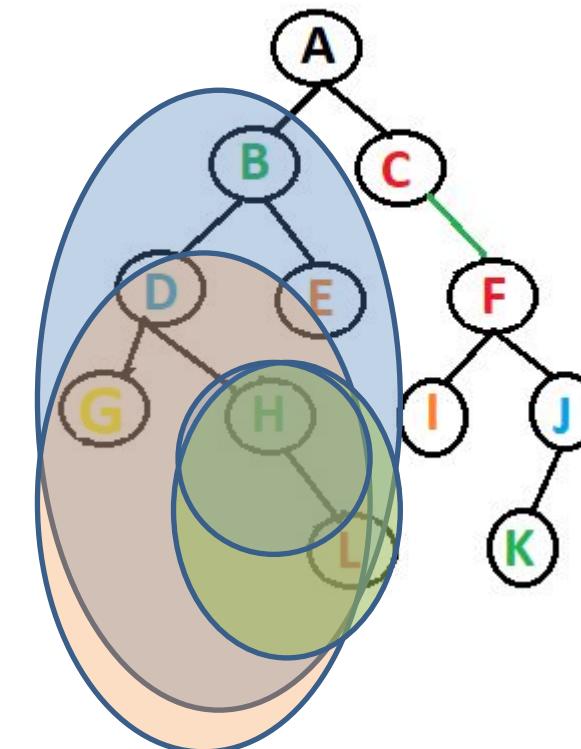


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H,



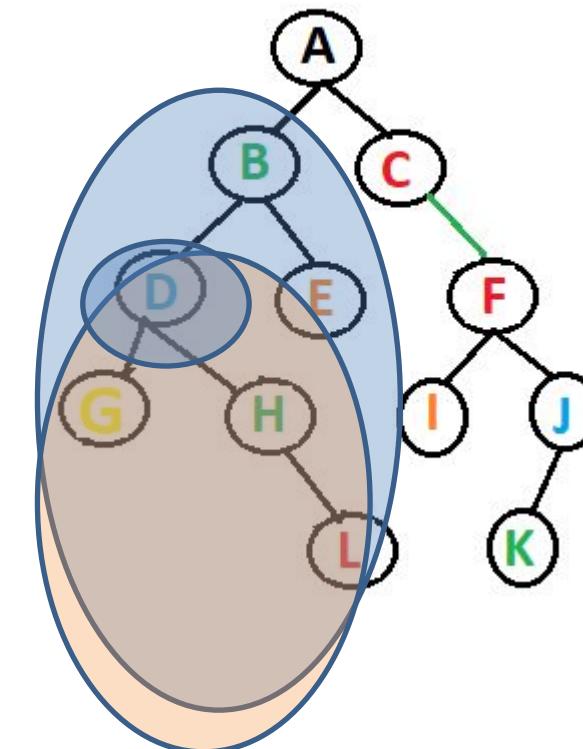


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D



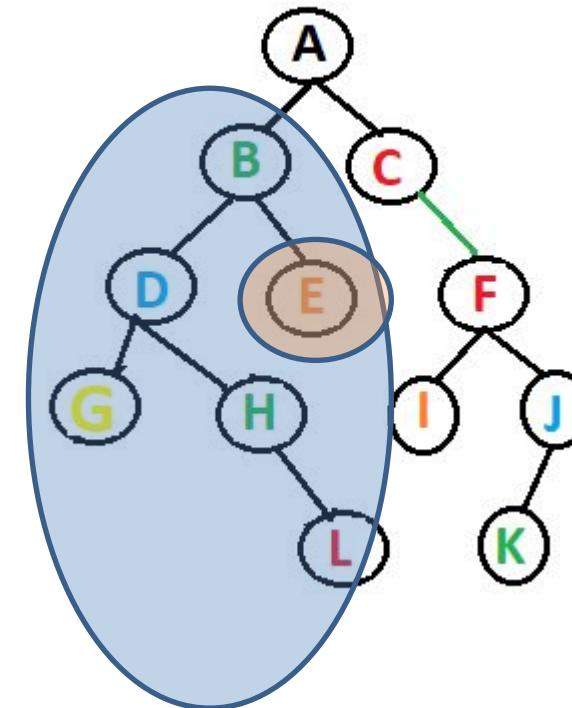


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. **Postorder(Tree->right)**
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E,



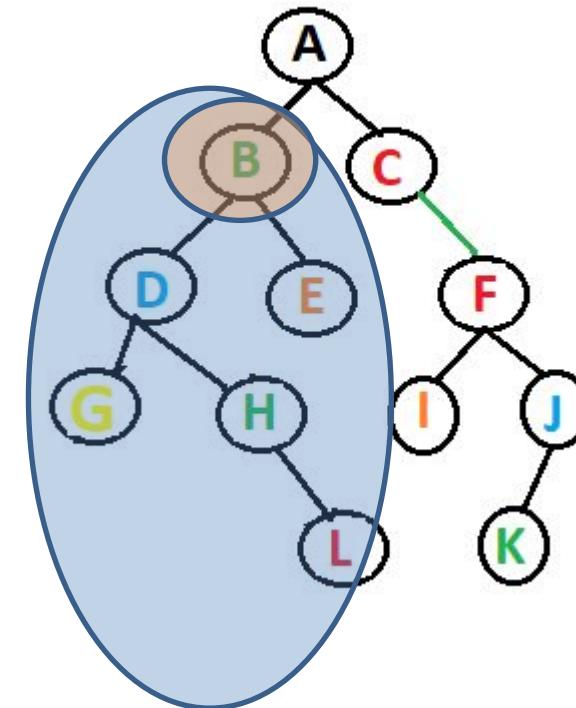


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B,



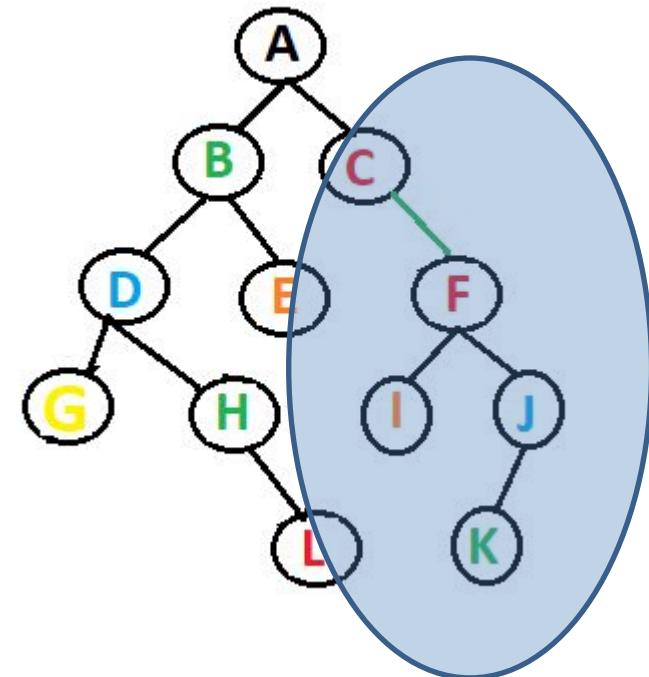


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B,



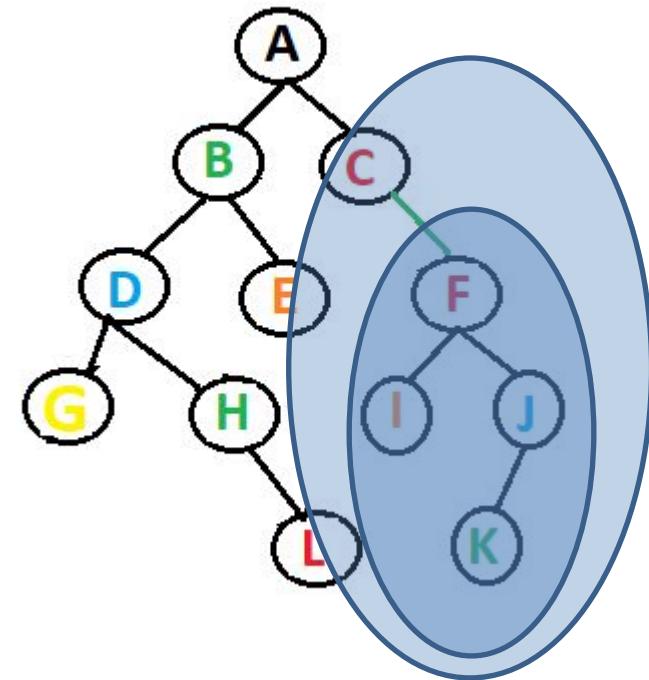


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B,



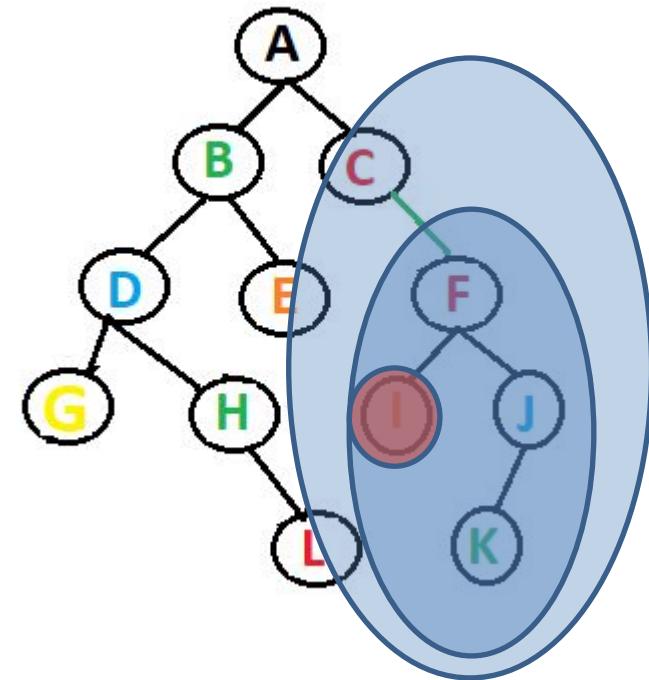


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I,



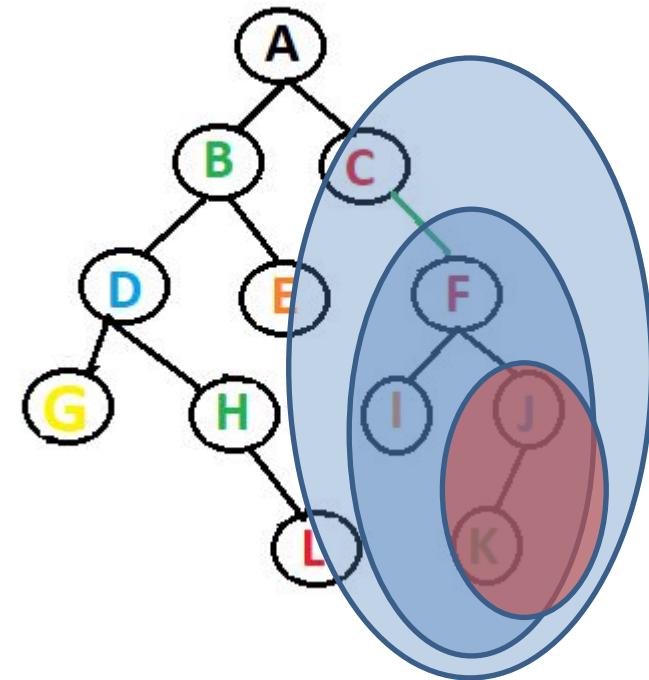


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. **Postorder(Tree->right)**
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I,



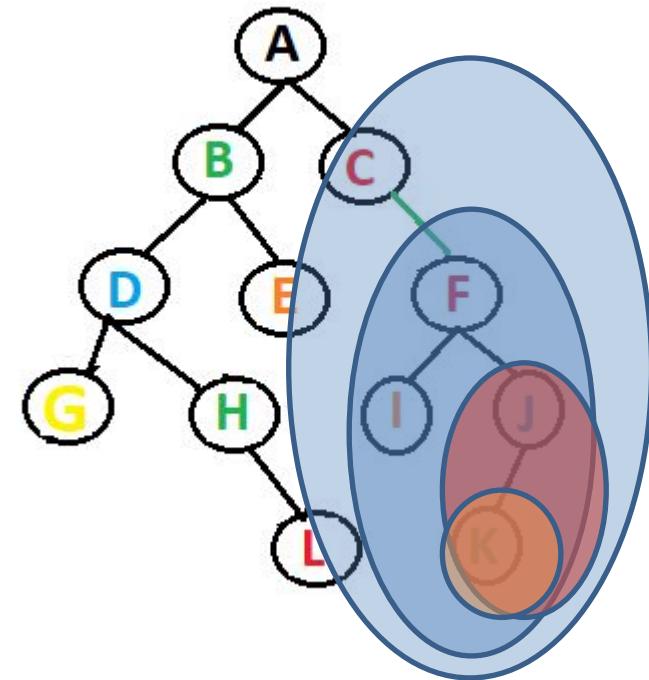


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I, K,



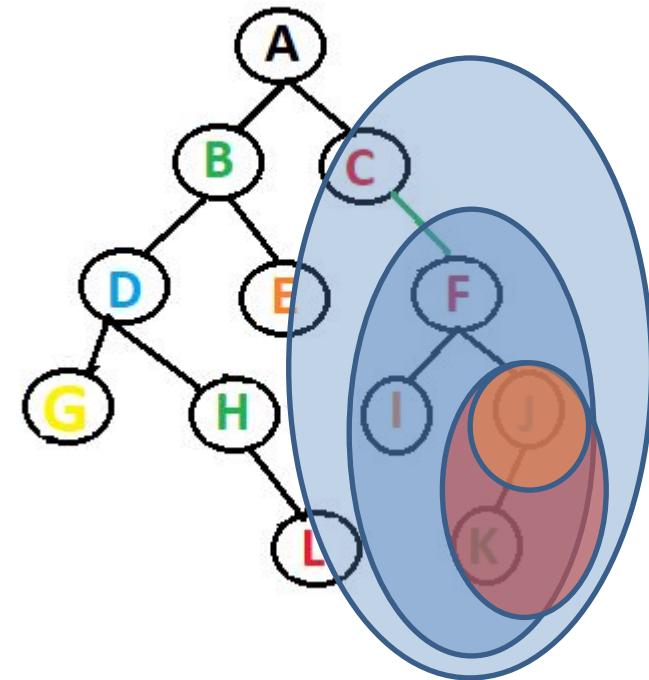


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I, K, J,



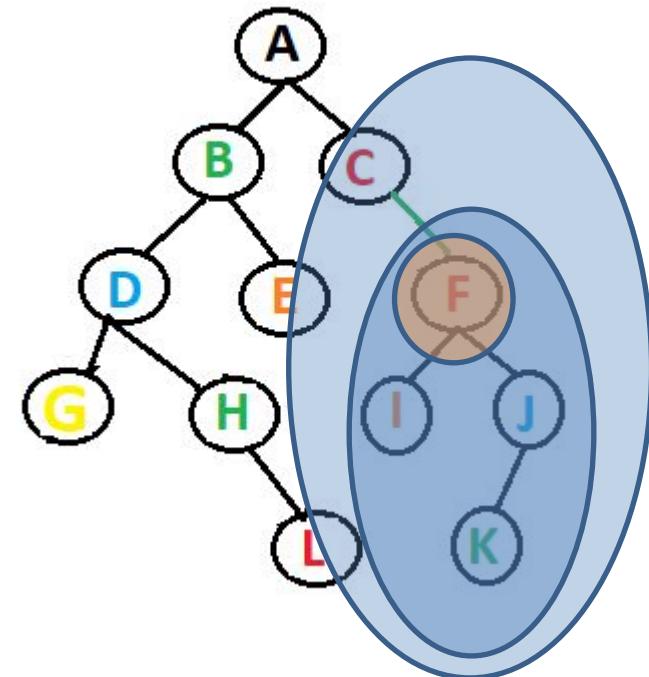


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

**Result : G, L, H, D, E, B, I, K, J,
F,**



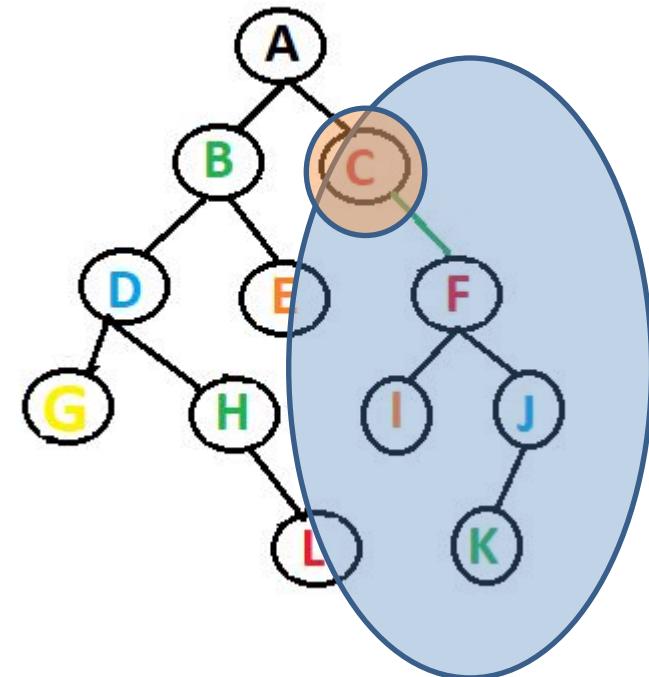


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I, K, J,
F, C,



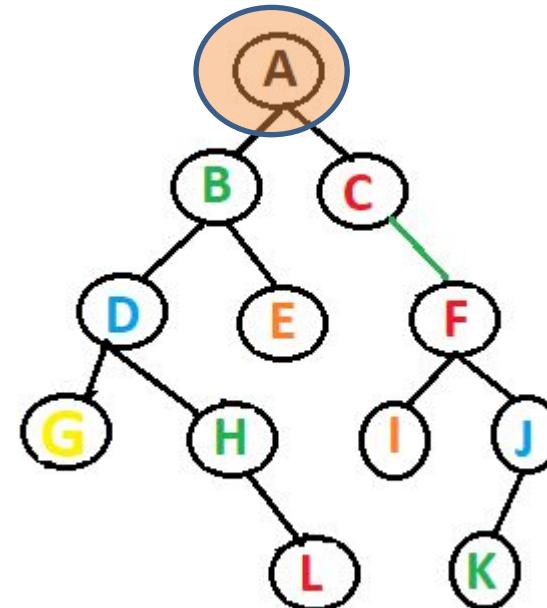


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

**Result : G, L, H, D, E, B, I, K, J,
F, C, A**



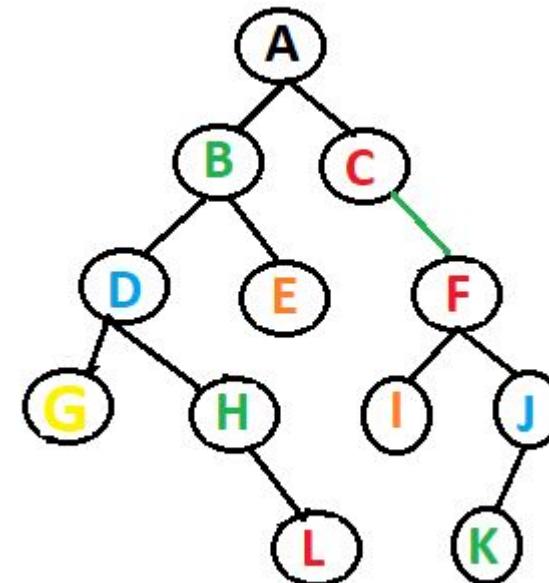


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

**Result : G, L, H, D, E, B, I, K, J,
F, C, A**





Threaded Binary Trees

- Nodes that does not have right child, have a thread to their inorder successor.
- Node structure:

Struct NODE1

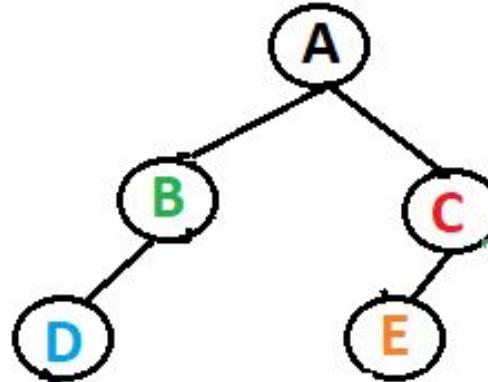
{

```
struct NODE1 *lchild;  
int Node1_data;  
Struct NODE1 *rchild;  
Struct NODE1 *trd;  
}
```



Threaded Binary Trees

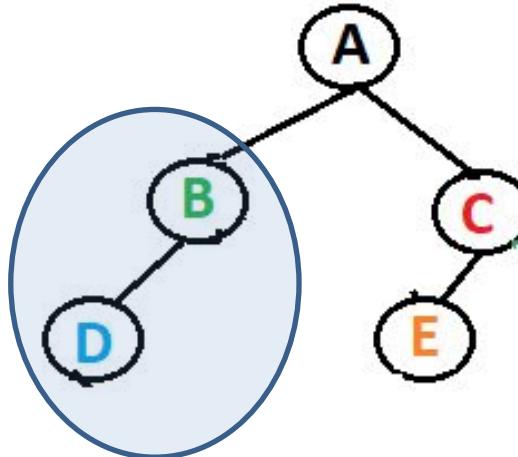
- Consider the following tree:
- Inorder traversal : D B A E C





Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C

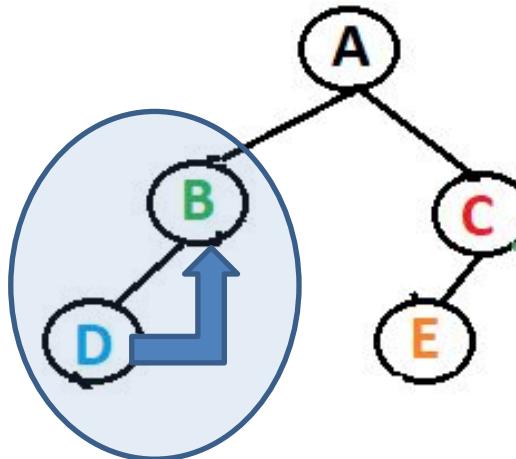


No Right Child for D – inorder traversal B comes after D



Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C



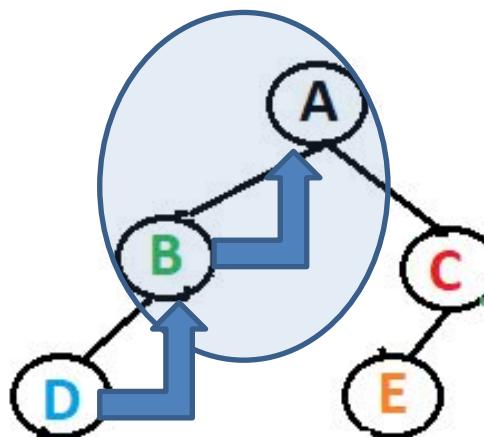
No Right Child for D – inorder traversal B comes after D

Create a thread from D to B



Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C



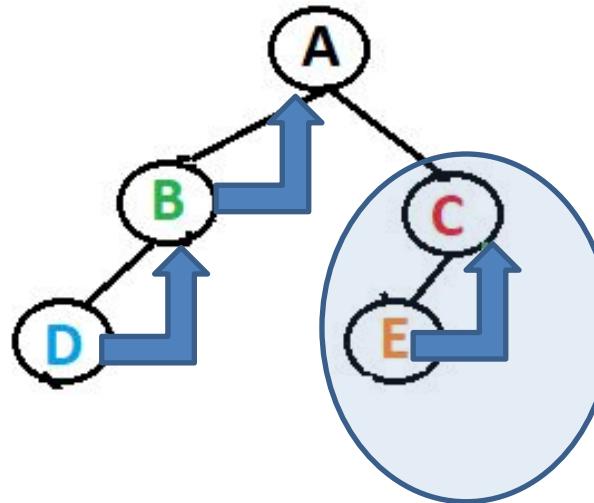
No Right Child for B – inorder traversal A comes after B

Create a thread from B to A



Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C



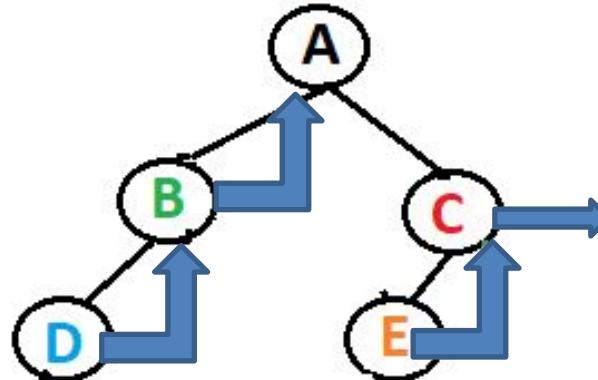
No Right Child for E – inorder traversal C comes after E

Create a thread from E to C



Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C



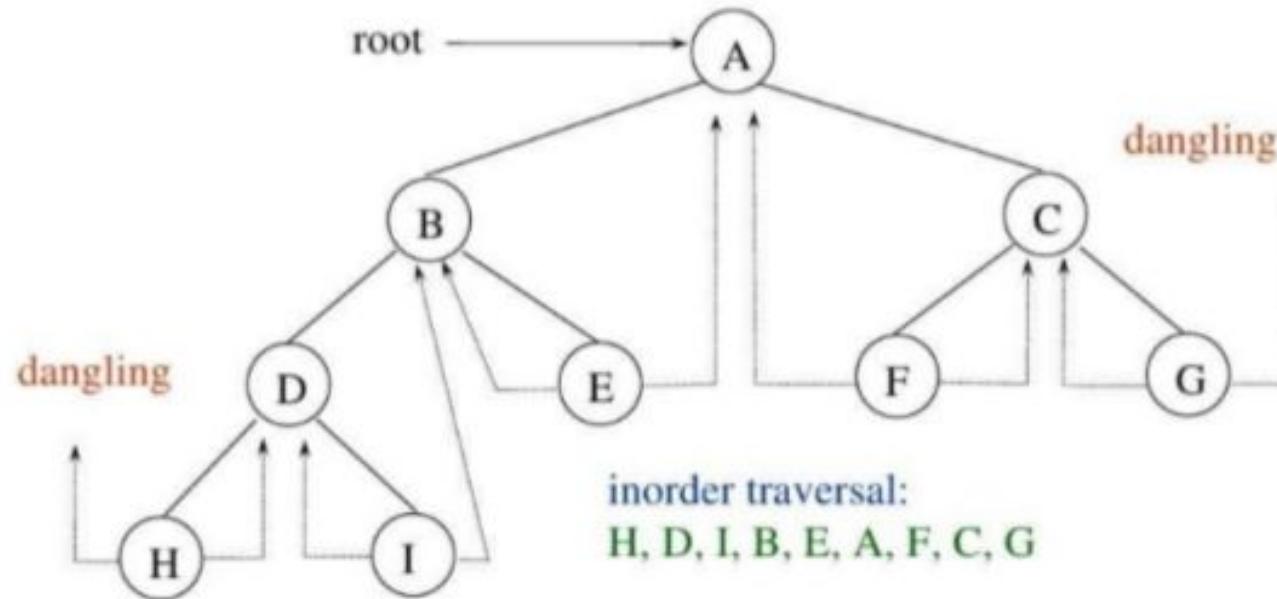
No Right Child for C – inorder traversal C comes at end

Create a hanging thread (dangling thread)



Threaded Binary Trees

A Threaded Binary Tree





SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Session- 7

BINARY SEARCH TREE



BINARY SEARCH TREE

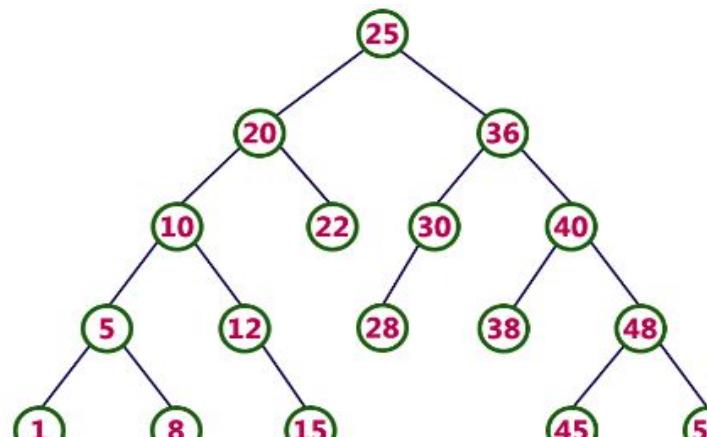
- Binary Search Tree is a binary tree in which every **node** contains only smaller values in its left sub tree and only larger values in its right sub tree.
- Also called as **ORDERED** binary tree

BST– properties:

- It should be Binary tree.
- Left subtree $<$ **Root Node** \leq Right subtree
(or)

Left subtree \leq **Root Node** $<$ Right subtree

Example:

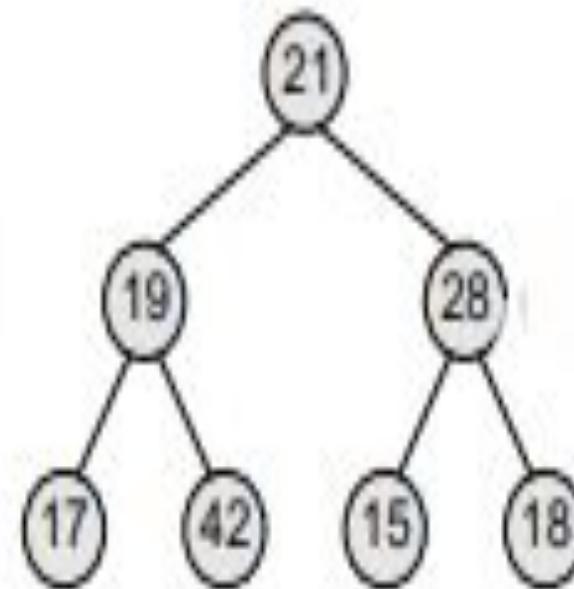
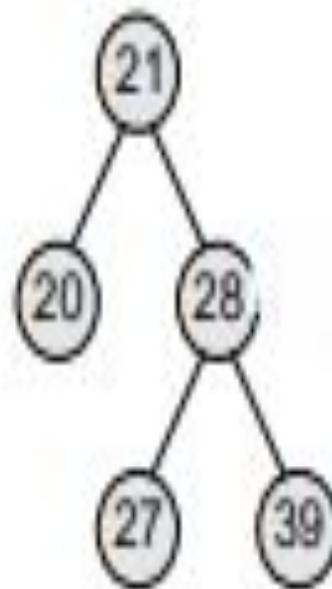
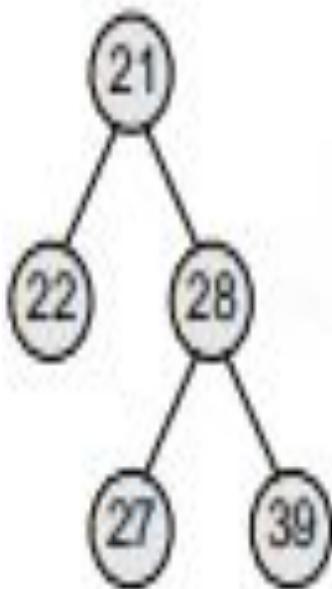




SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Binary search trees or not ?



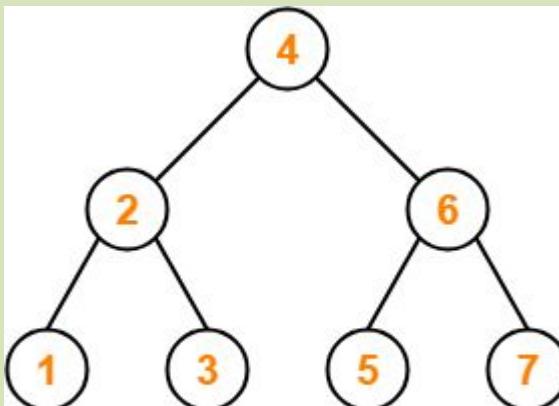


- **Operations:** Searching, Insertion, Deletion of a Node
- **TimeComplexity :**

BEST CASE

- Search Operation - $O(\log n)$
- Insertion Operation- $O(\log n)$
- Deletion Operation - $O(\log n)$

Example:

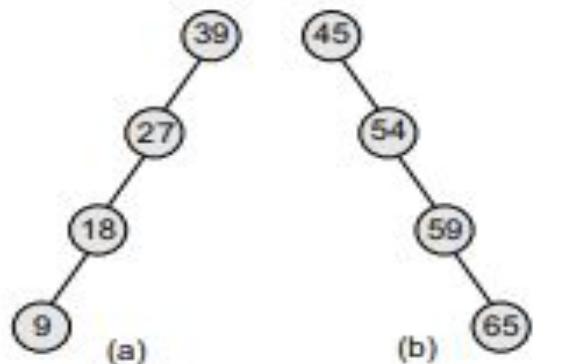


Balanced Binary Search Tree

WORST CASE

- Search Operation - $O(n)$
 - Insertion Operation- $O(1)$
 - Deletion Operation - $O(n)$
- (note: Height of the binary search tree becomes n)

Example:



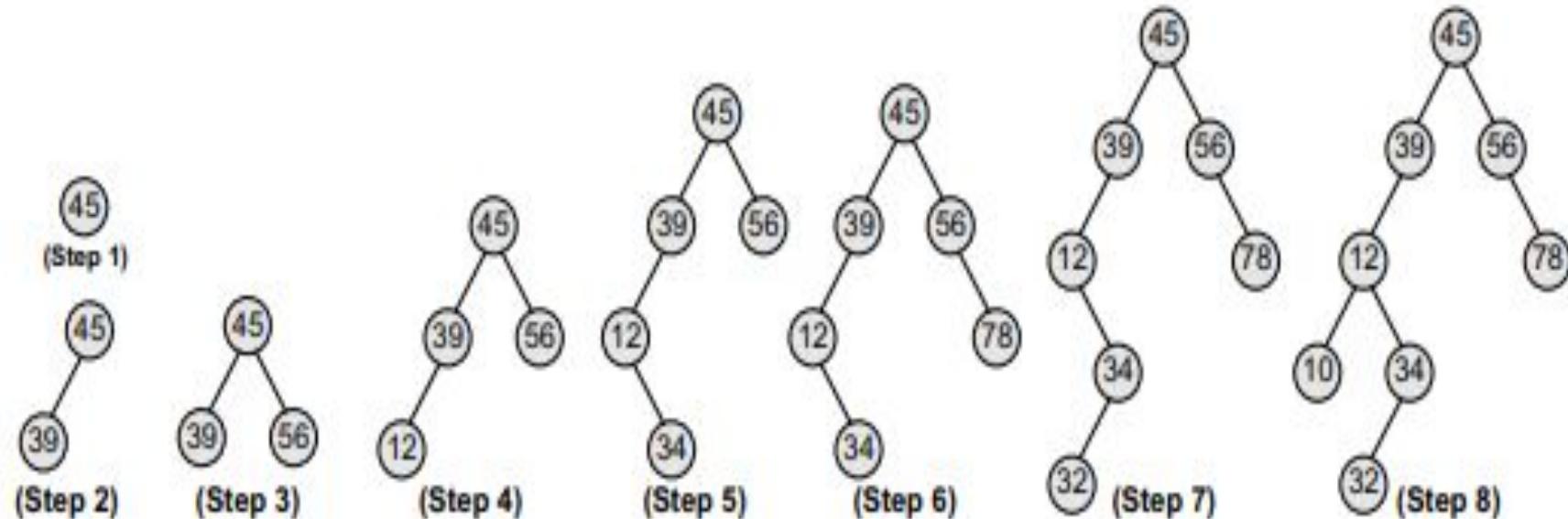
(a) **Left skewed**, and (b) **right skewed** binary search trees



Binary search tree Construction

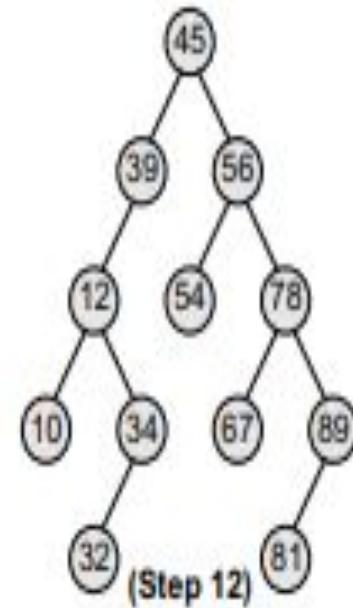
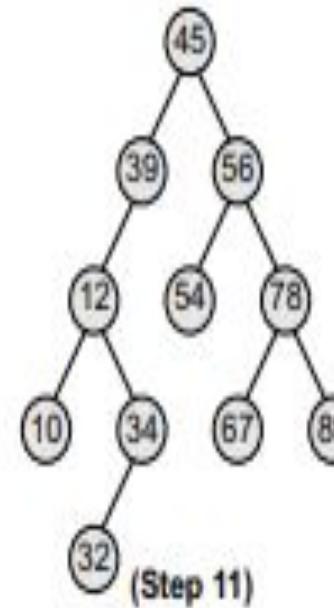
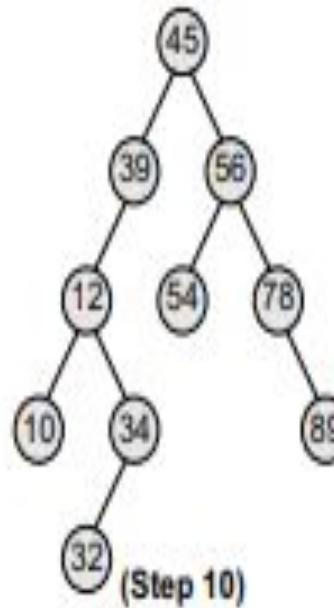
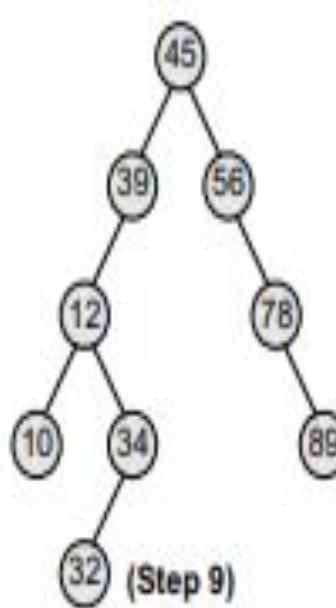
- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

Solution





Remaining: 89, 54, 67, 81





SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

SEARCHING A NODE IN BST

SearchElement (TREE, VAL)

Step 1: IF TREE DATA = VAL OR TREE = NULL

 Return TREE

ELSE

 IF VAL < TREE DATA

 Return searchElement(TREE LEFT, VAL)

ELSE

 Return searchElement(TREE RIGHT, VAL)

 [END OF IF]

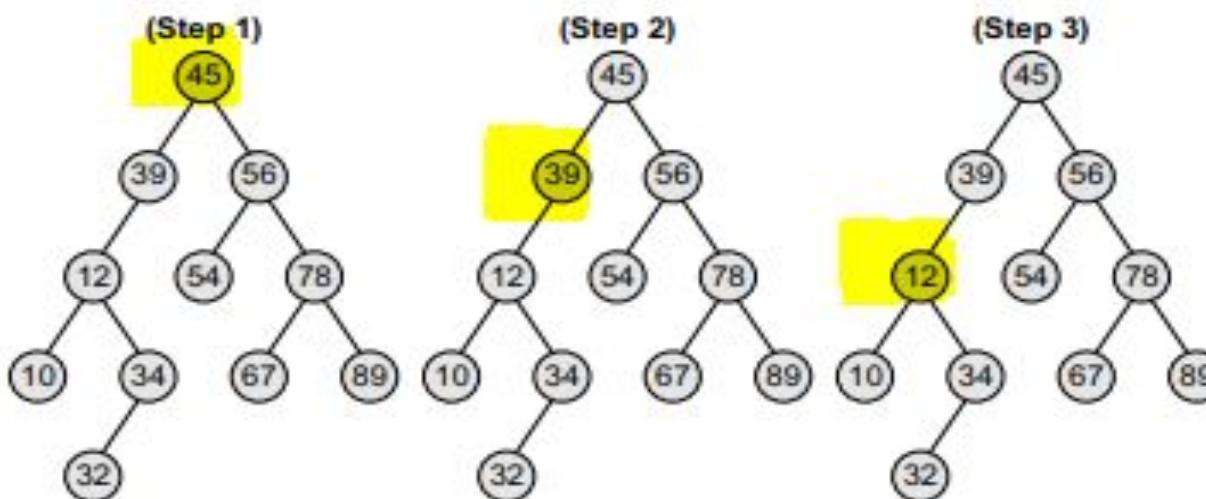
 [END OF IF]

Step 2: END



EXAMPLE 1:

Searching a node with value **12** in the given binary search tree



We start our search from the root node **45**.

As **12 < 45**, so we search in 45's **LEFT** subtree.

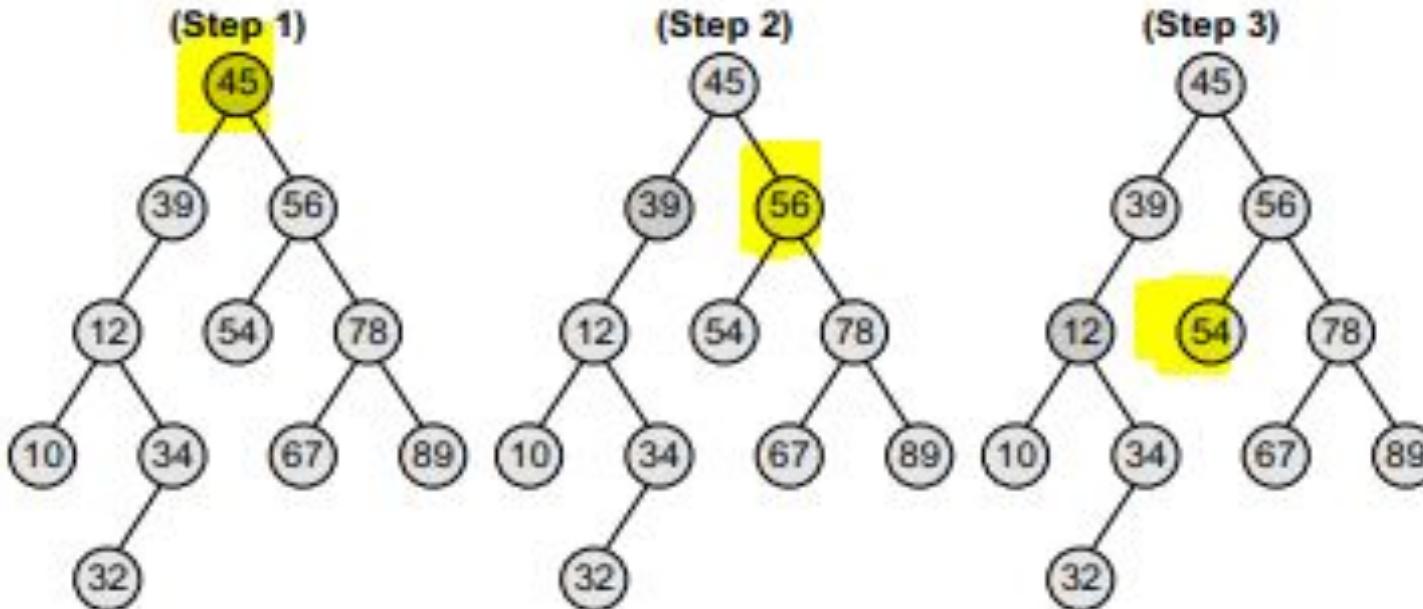
As **12 < 39**, so we search in 39's **LEFT** subtree.

So, we conclude that **12** is **present** in the above BST.



EXAMPLE 2:

Searching a node with value **52** in the given binary search tree



We start our search from the root node **45**.

As **52 > 45**, so we search in 45's **RIGHT** subtree.

As **52 < 56** so we search in 56's **LEFT** subtree.

As **52 < 54** so we search in 54's **LEFT** subtree.

But **54 is leaf node**

So, we conclude that **52 is not present** in the above BST.



SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

INSERTING A NODE IN BST

Insert (TREE, VAL)

Step 1: IF TREE = NULL

 Allocate memory for TREE

 SET TREE DATA = VAL

 SET TREE LEFT = TREE RIGHT = NULL

ELSE

 IF VAL < TREE DATA

 Insert(TREE LEFT, VAL)

ELSE

 Insert(TREE RIGHT, VAL)

[END OF IF]

[END OF IF]

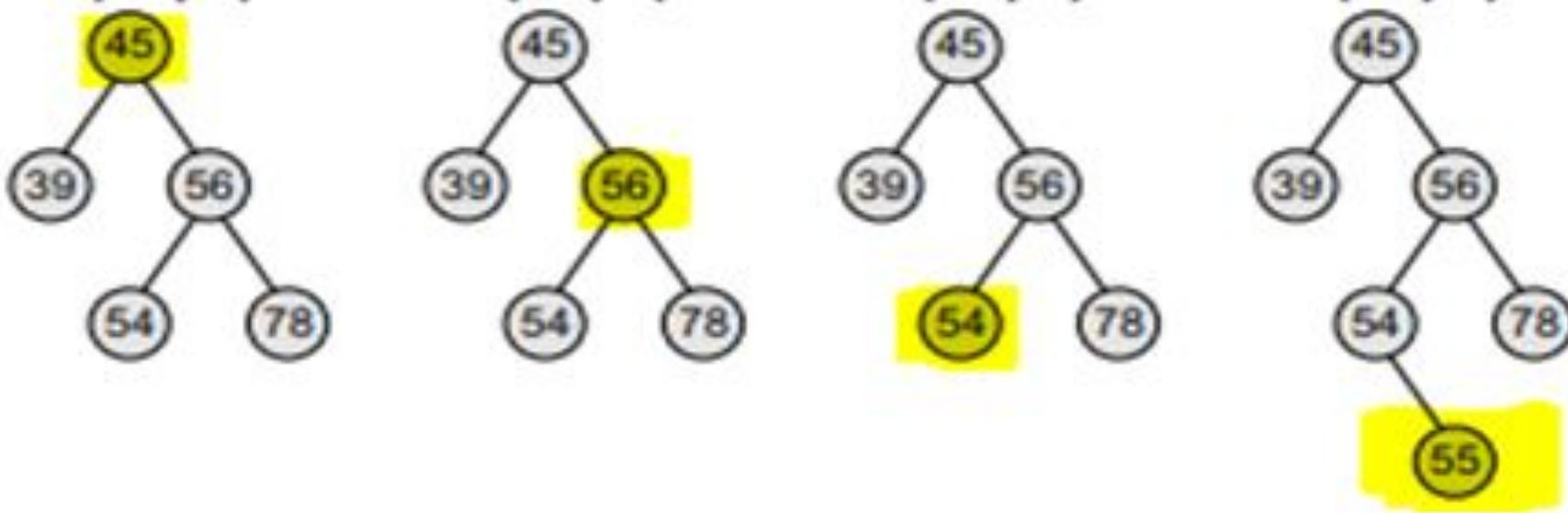
Step 2: END



SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

EXAMPLE : Inserting nodes with values 55 in the given binary search tree



We start searching for value **55** from the root node **45**.

As **55 > 45**, so we search in 45's **RIGHT** subtree.

As **55 < 56** so we search in 56's **LEFT** subtree.

As **55 > 54** so we add **55 to 54's right** subtree.



SRM Deletion Operation in BST

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Delete (TREE, VAL)

Step 1: IF TREE = NULL

 Write "VAL not found in the tree"

ELSE IF **VAL < TREE DATA**

 Delete(TREE->LEFT, VAL)

ELSE IF **VAL > TREE DATA**

 Delete(TREE RIGHT, VAL)

ELSE IF **TREE LEFT AND TREE RIGHT**

 SET TEMP = findLargestNode(TREE LEFT) **(INORDER PREDECESSOR)**
 (OR)

 SET TEMP = findSmallestNode(TREE RIGHT) **(INORDER SUCESSOR)**

 SET TREE DATA = TEMP DATA

 Delete(TREE LEFT, TEMP DATA) **(OR)** Delete(TREE RIGHT, TEMP DATA)

ELSE

 SET TEMP = TREE

IF TREE LEFT = NULL AND TREE RIGHT = NULL

 SET TREE = NULL

ELSE IF TREE LEFT != NULL

 SET TREE = TREE LEFT

ELSE

 SET TREE = TREE RIGHT

[END OF IF]

FREE TEMP

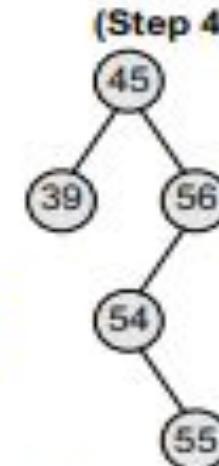
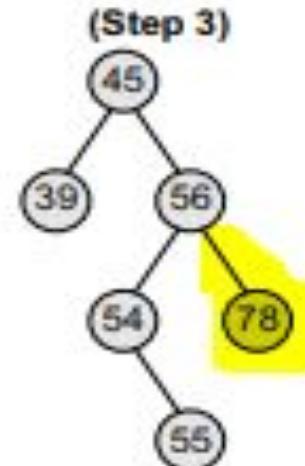
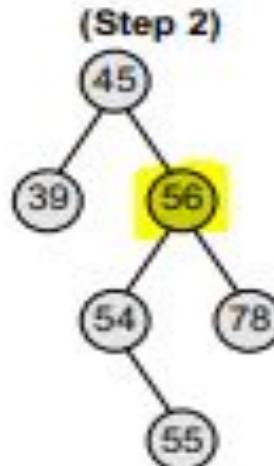
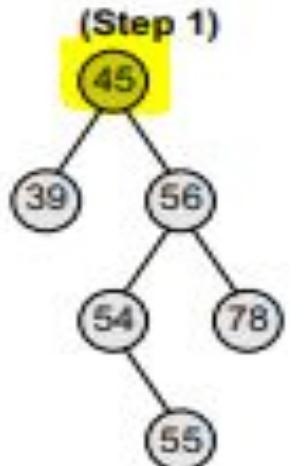
[END OF IF]

Step 2: END



Case 1: Deleting a Leaf node (A node with no children)

EXAMPLE : Deleting node 78 from the given binary search tree

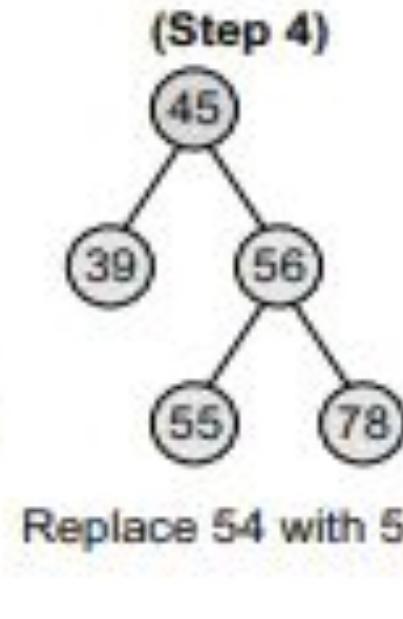
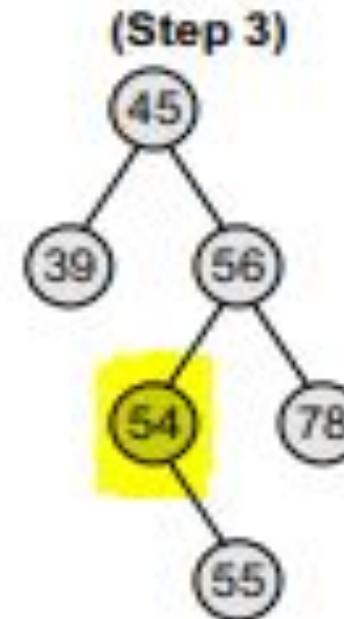
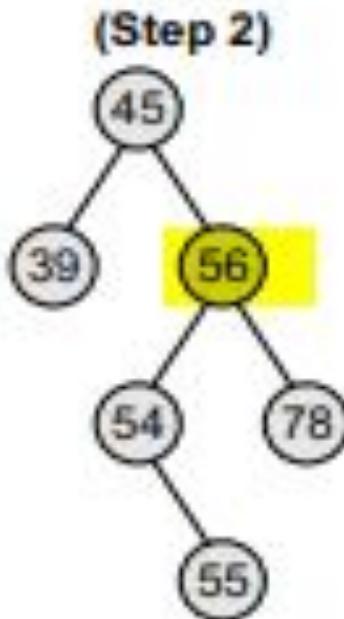
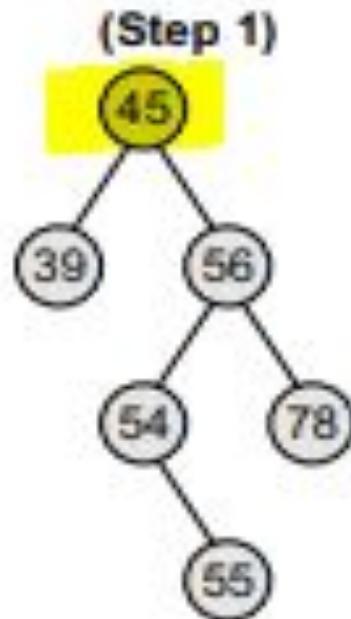


Delete node 78



Case 2: Deleting a node with one child

EXAMPLE : Deleting node 54 from the given binary search tree

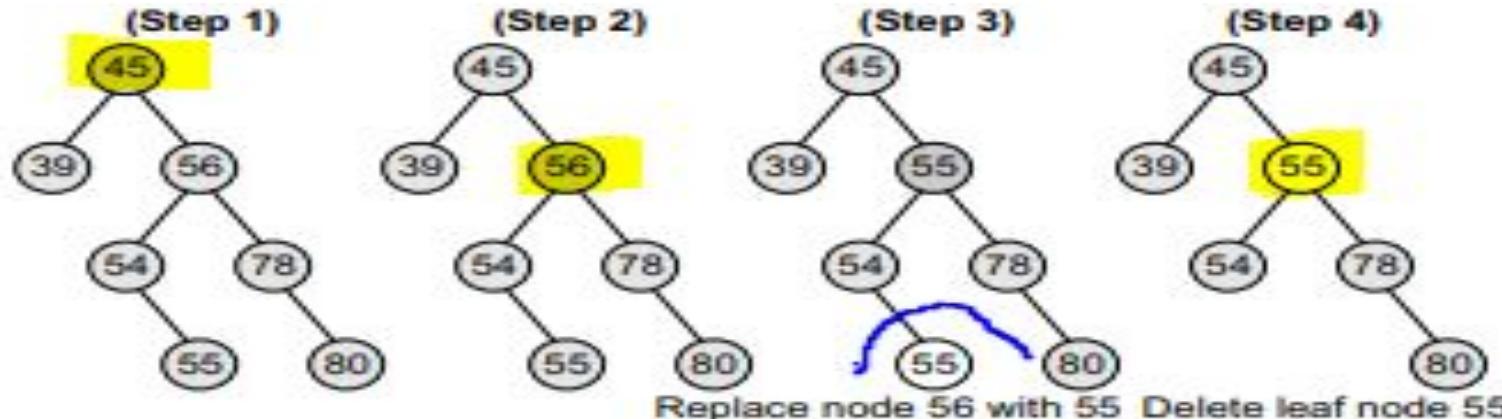




Case 3: Deleting a node with two children

EXAMPLE 1 : Deleting node 56 from the given binary search tree

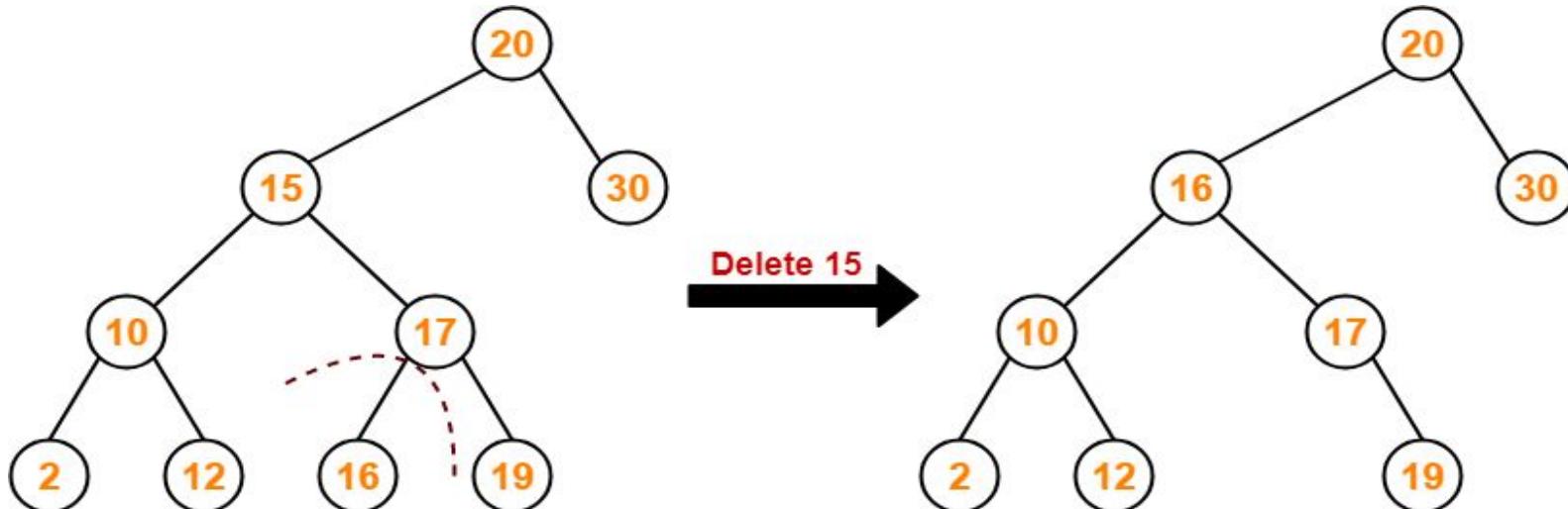
- Visit to the **left sub tree** of the deleting node.
- Grab the greatest value element called as **in-order predecessor**.
- Replace **the deleting element** with its **in-order predecessor**.





EXAMPLE 2 : Deleting node 15 from the given binary search tree

- Visit to the **right sub tree** of the deleting node.
- Pluck the **least value** element called as **inorder successor**.
- Replace the **deleting element** with its **inorder successor**.





Other possible operations:

- Determining the height of a tree
- Determining the no of nodes a tree
- Determining the no of internal nodes
- Determining the no of external nodes
- Determining the mirror images
- Removing the tree
- Finding the smallest node
- Finding the largest node



Session-8

AVL TREE

AVL TREE

- Named after Adelson-Velskii and Landis as AVL tree
- Also called as *self-balancing* binary search tree

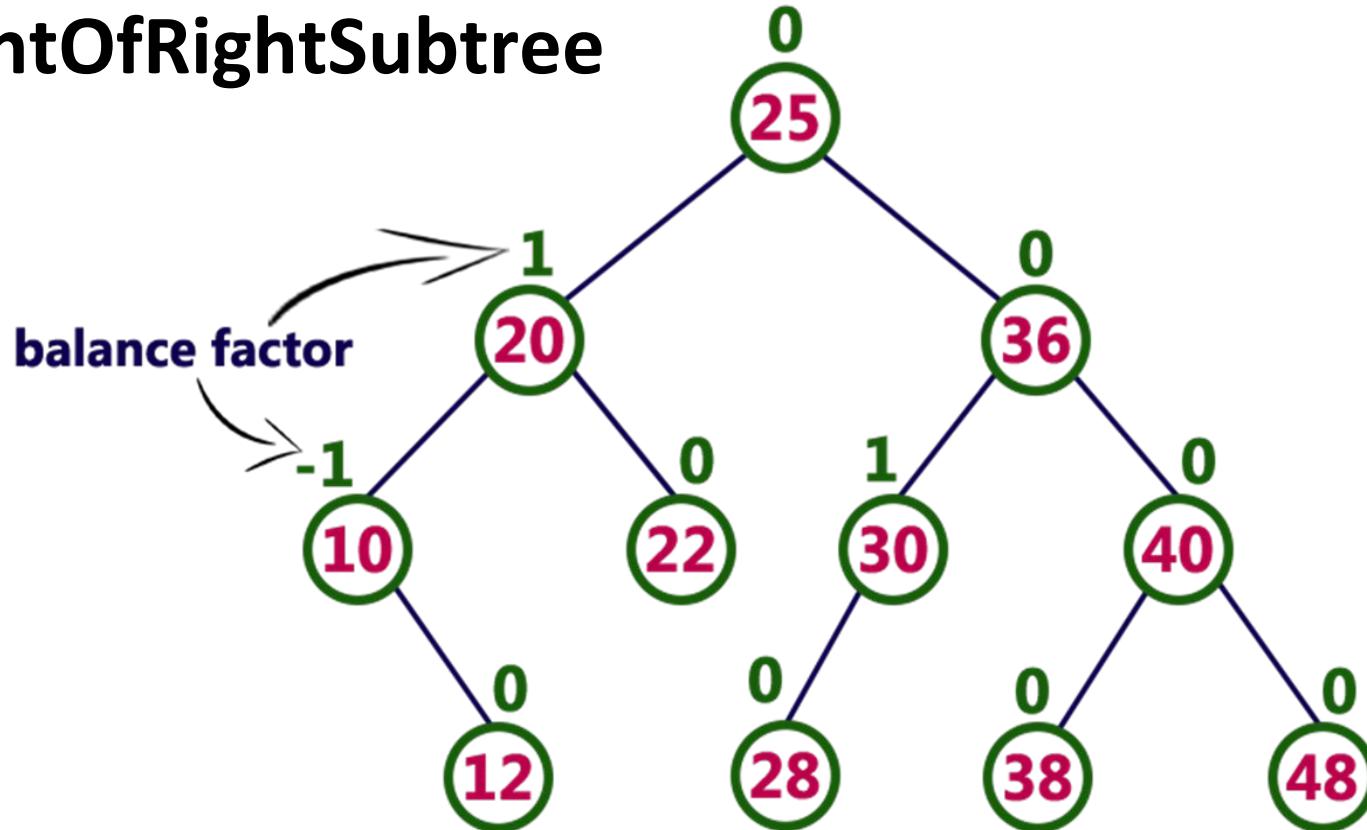
AVL tree – properties:

- It should be Binary search tree
- **Balancing factor:** balance of every node is either -1 or 0 or 1
where **balance(node)** = height(*node.left subtree*) – height(*node.right subtree*)
- Maximum possible number of nodes in AVL tree of height H
$$= 2^{H+1} - 1$$
- **Operations:** Searching, Insertion, Deletion of a Node
- **TimeComplexity :** $O(\log n)$



Balancing Factor

- Balance factor = heightOfLeftSubtree – heightOfRightSubtree

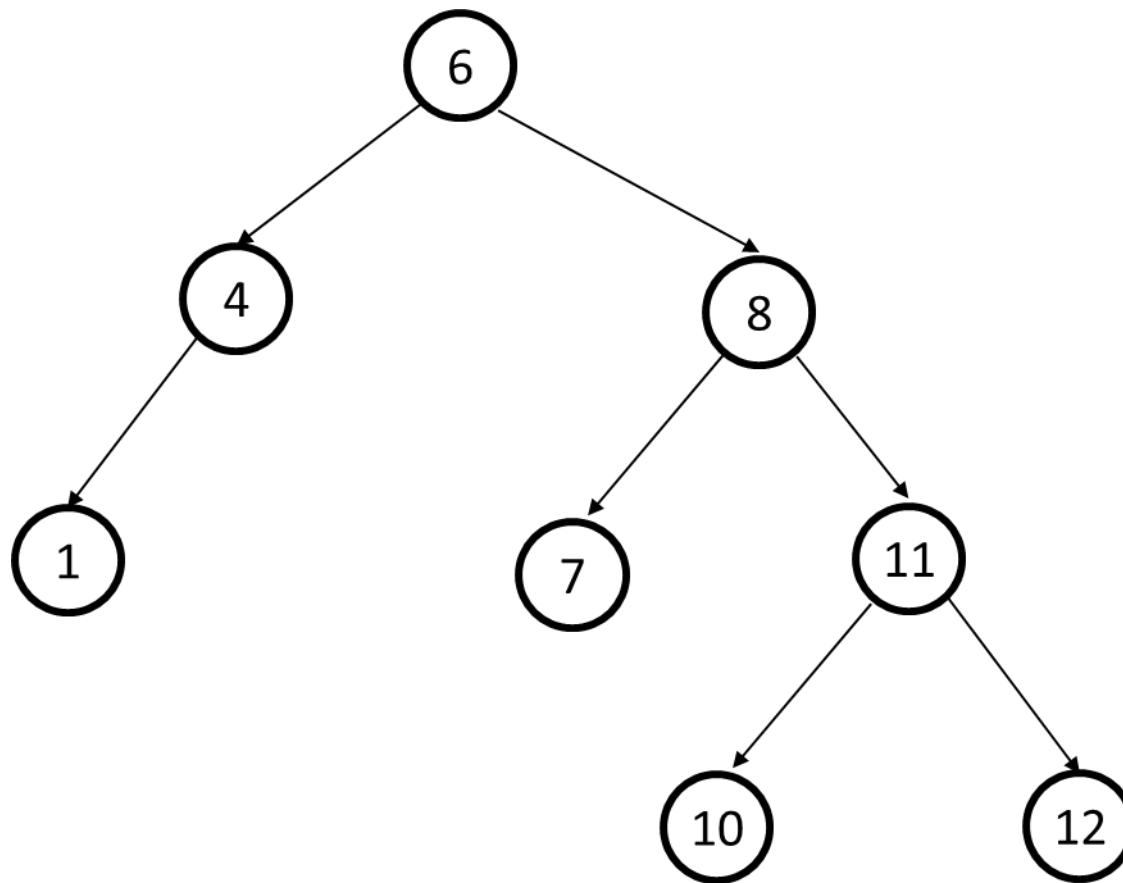




SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Example 1 : Check - AVL Tree?

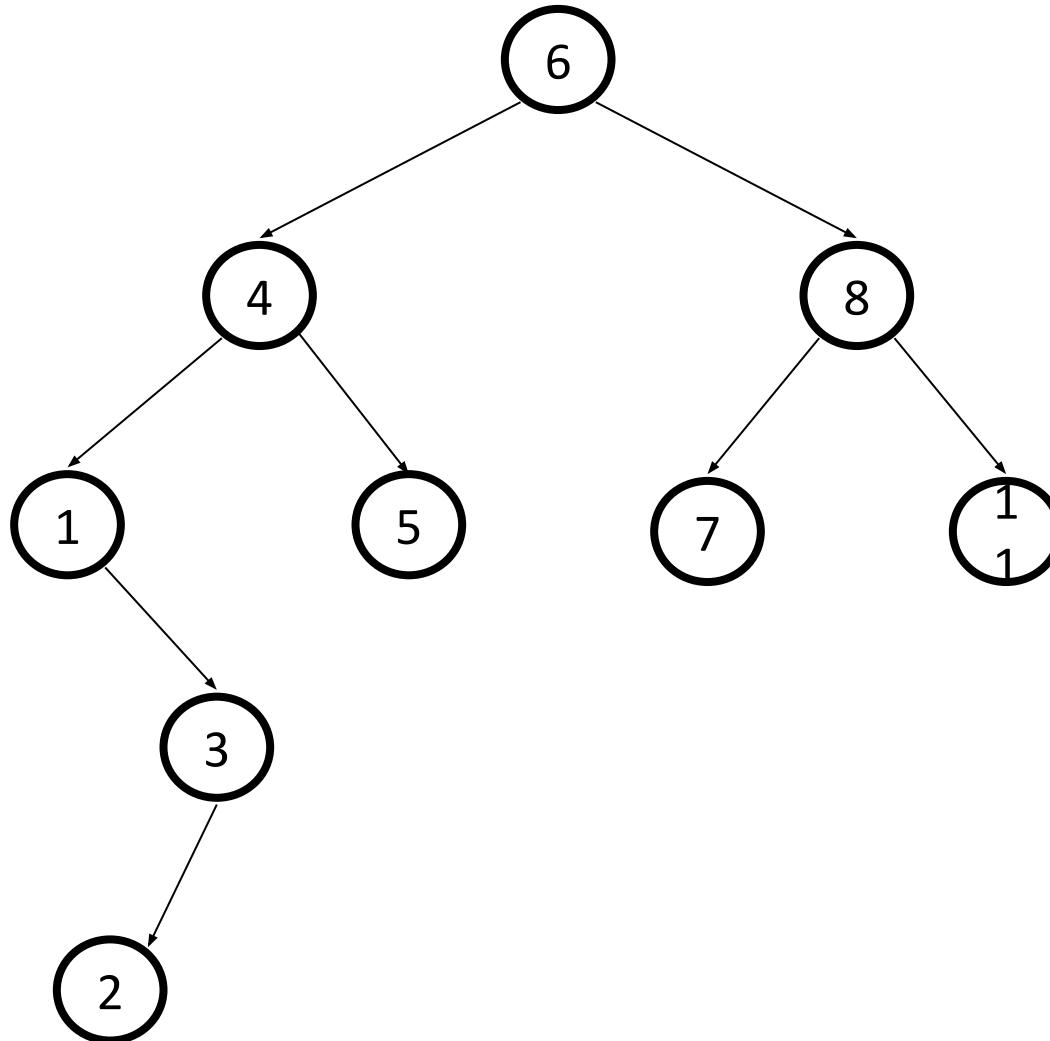




SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Example 2: Check - AVL Tree?





SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

operations on AVL tree

1. SEARCHING
2. INSERTION
3. DELETION



SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Search Operation in AVL Tree

ALGORITHM:

STEPS:

- 1 : Get the search element
- 2 : check search element == root node in the tree.
- 3 : If both are **exact match**, then display "**element found**" and end.
- 4 : If both are **not matched**, then check whether search element is < or > than that node value.
- 5: If search element is < then continue search in **left sub tree**.
- 6: If search element is > then continue search in **right sub tree**.
- 7: **Repeat** the step from **1 to 6** until we find the **exact element**
- 8: Still the search element is not found after reaching the leaf node display "**element not found**".



INSERTION or DELETION

- After performing any operation on AVL tree - the **balance factor** of **each node** is to be **checked**.
- After insertion or deletion there exists either any one of the following:

Scenario 1:

- After insertion or deletion , the balance factor of each node **is either 0 or 1 or -1**.
- If so AVL tree is considered to be **balanced**.
- The operation ends.

Scenario 1:

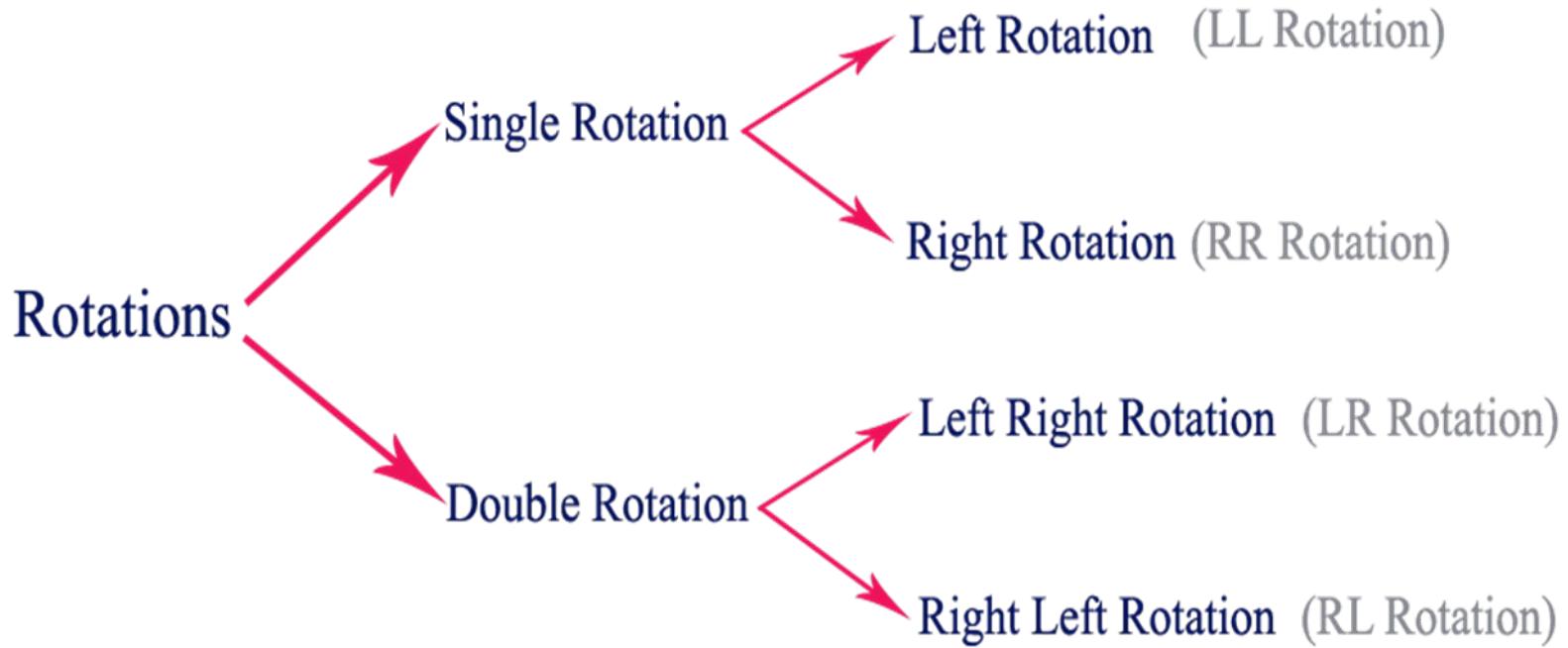
- After insertion or deletion, the balance factor **is not 0 or 1 or -1** for at least one node then
- The AVL tree is considered to be **imbalanced**.
- If so, **Rotations** are need to be performed **to balance** the tree in order to **make it as AVL TREE**.



SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

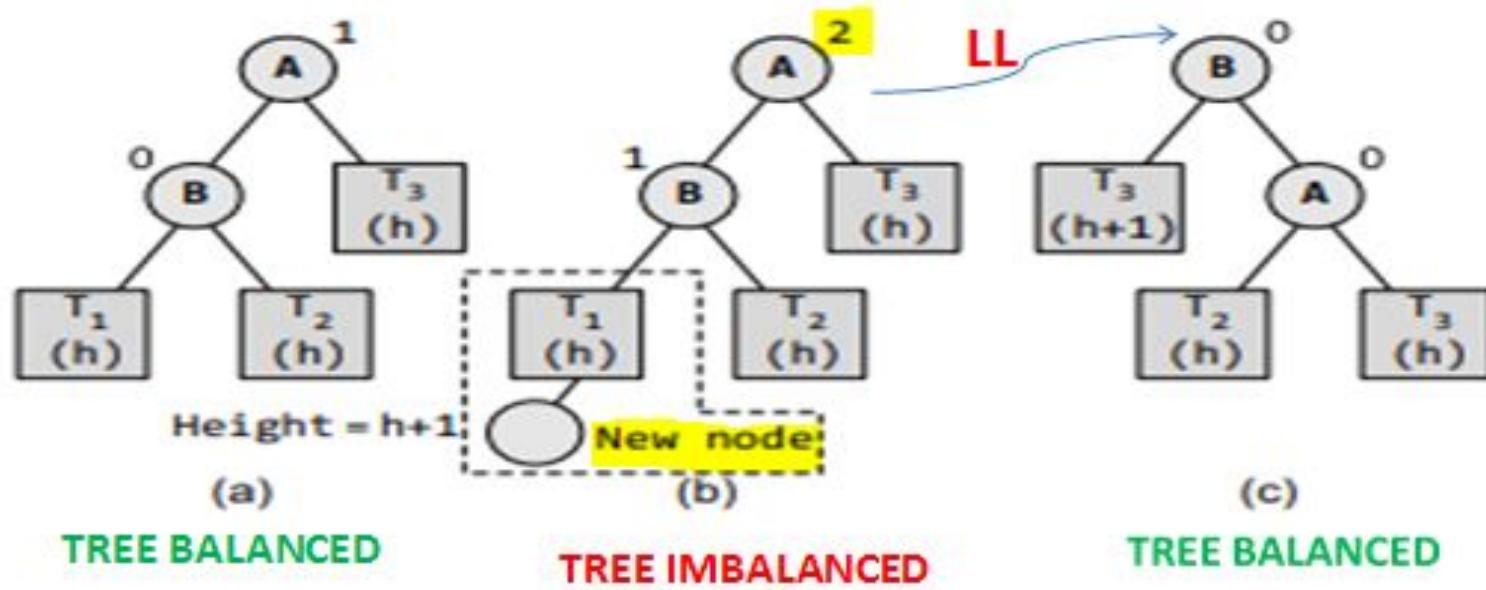
AVL TREE ROTATION





LL ROTATION

When new node is inserted in the **left sub-tree** of the **left sub-tree** of the critical

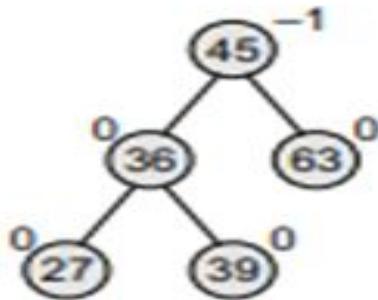




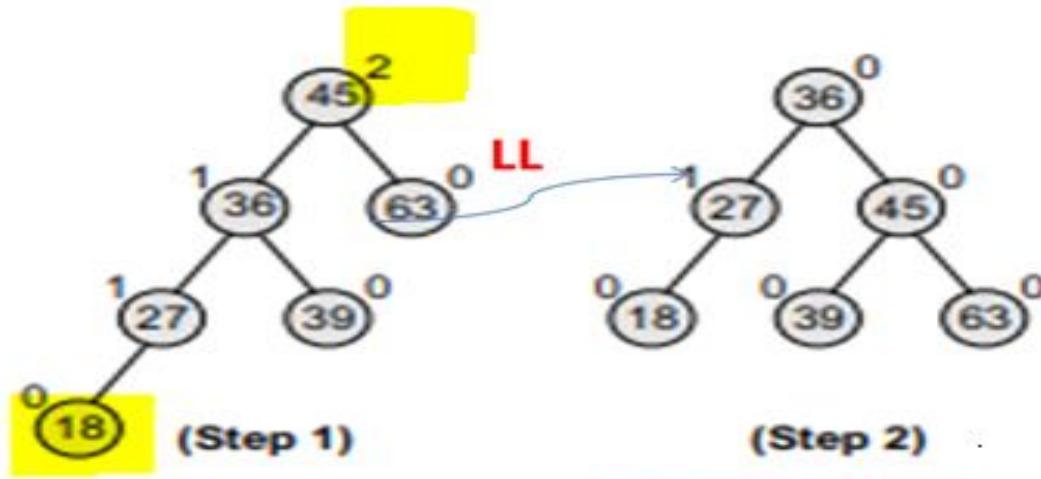
SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

LL ROTATION- Example



TREE BALANCED



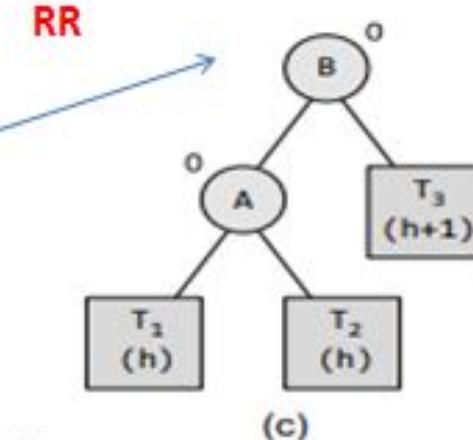
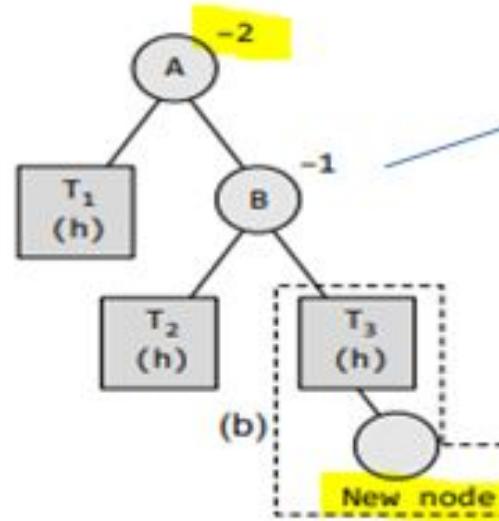
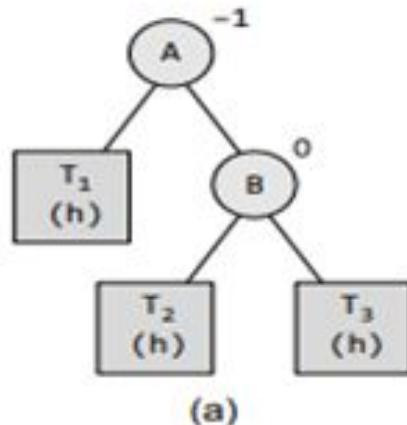
TREE IMBALANCED

TREE BALANCED



RR ROTATION

When new node is inserted in the **right sub-tree** of the **right sub-tree** of the critical



TREE BALANCED

TREE IMBALANCED

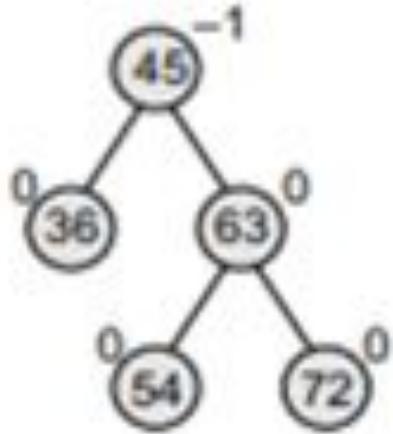
TREE BALANCED



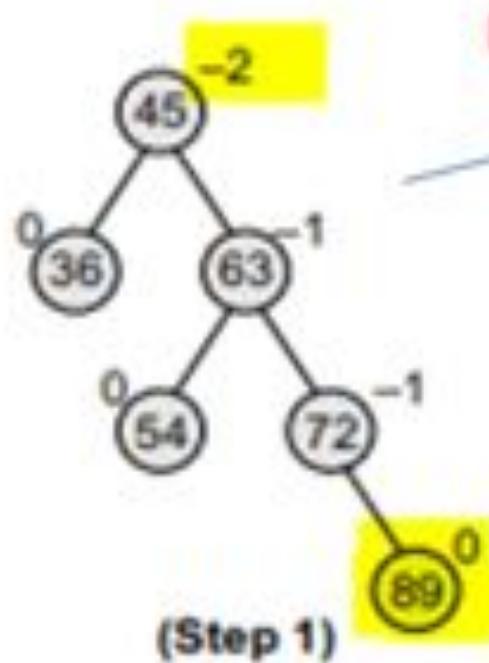
SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

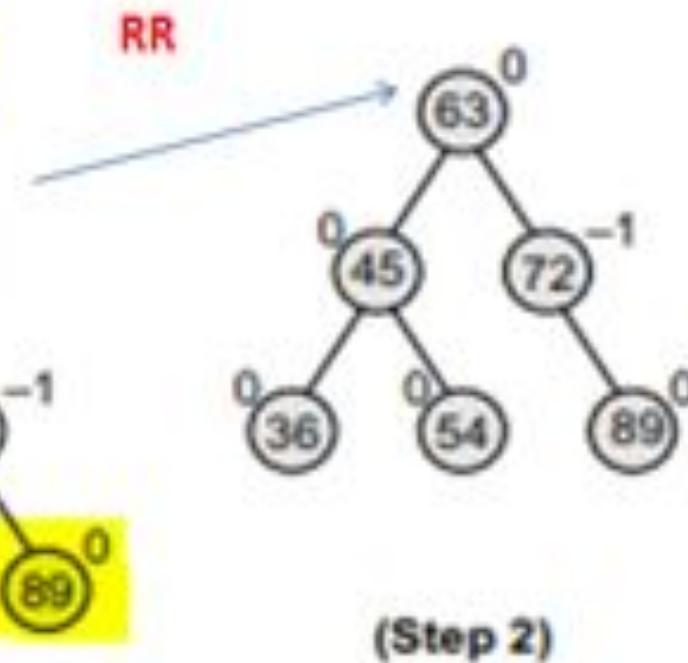
RR Rotation - Example



TREE BALANCED



TREE IMBALANCED



TREE BALANCED

RR

(Step 1)

(Step 2)

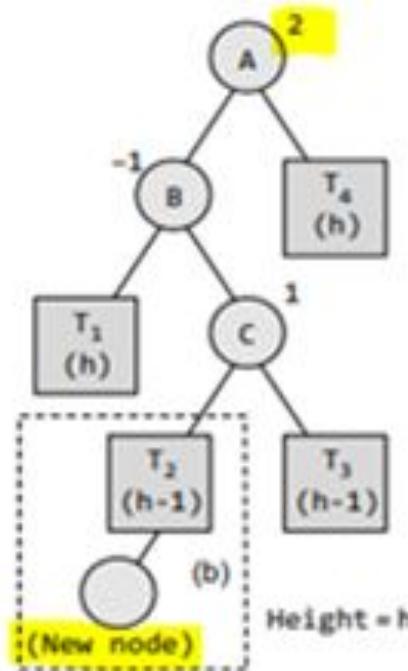
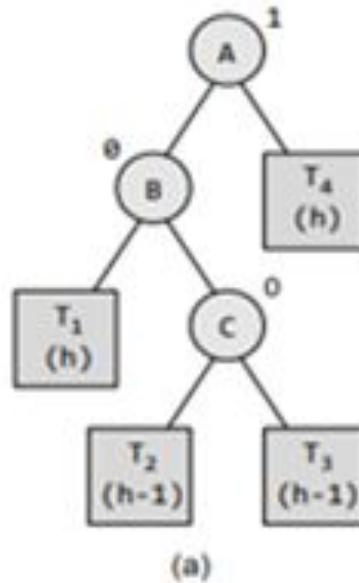


SRM

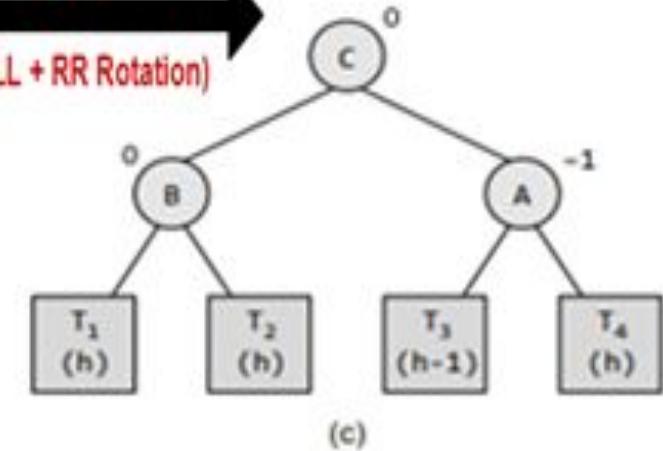
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

LR ROTATION

When new node is inserted in the **left sub-tree** of the **right sub-tree** of the critical



LR Rotation
(LL + RR Rotation)



TREE BALANCED

TREE IMBALANCED

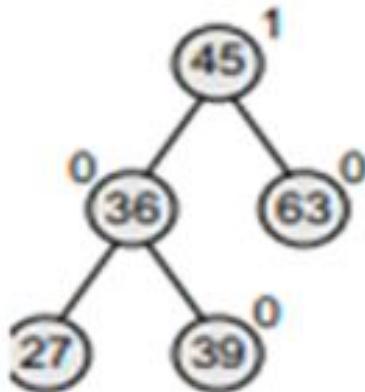
TREE BALANCED



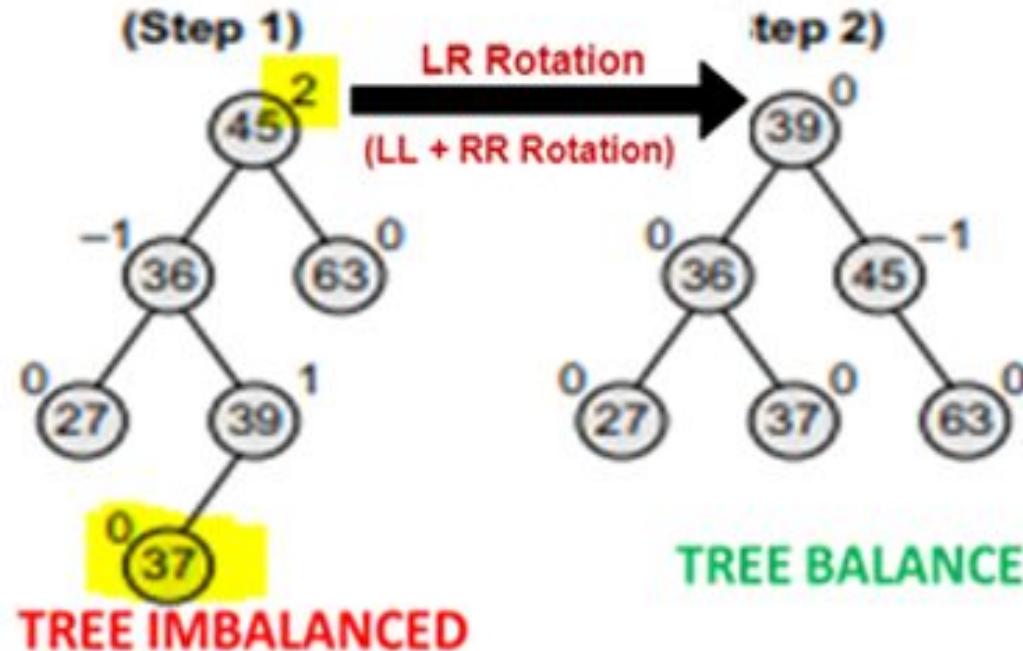
SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

LR Rotation - Example



TREE BALANCED

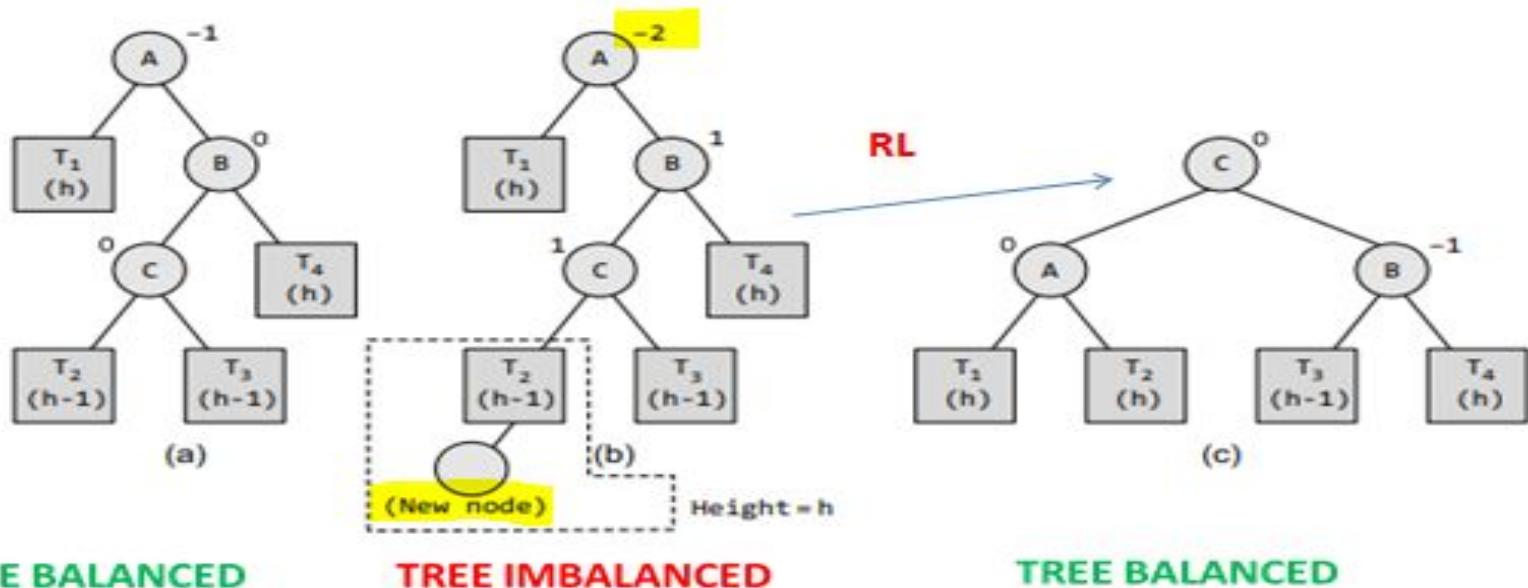


TREE BALANCED



RL ROTATION

When new node is inserted in the **right sub-tree of the left sub-tree** of the critical

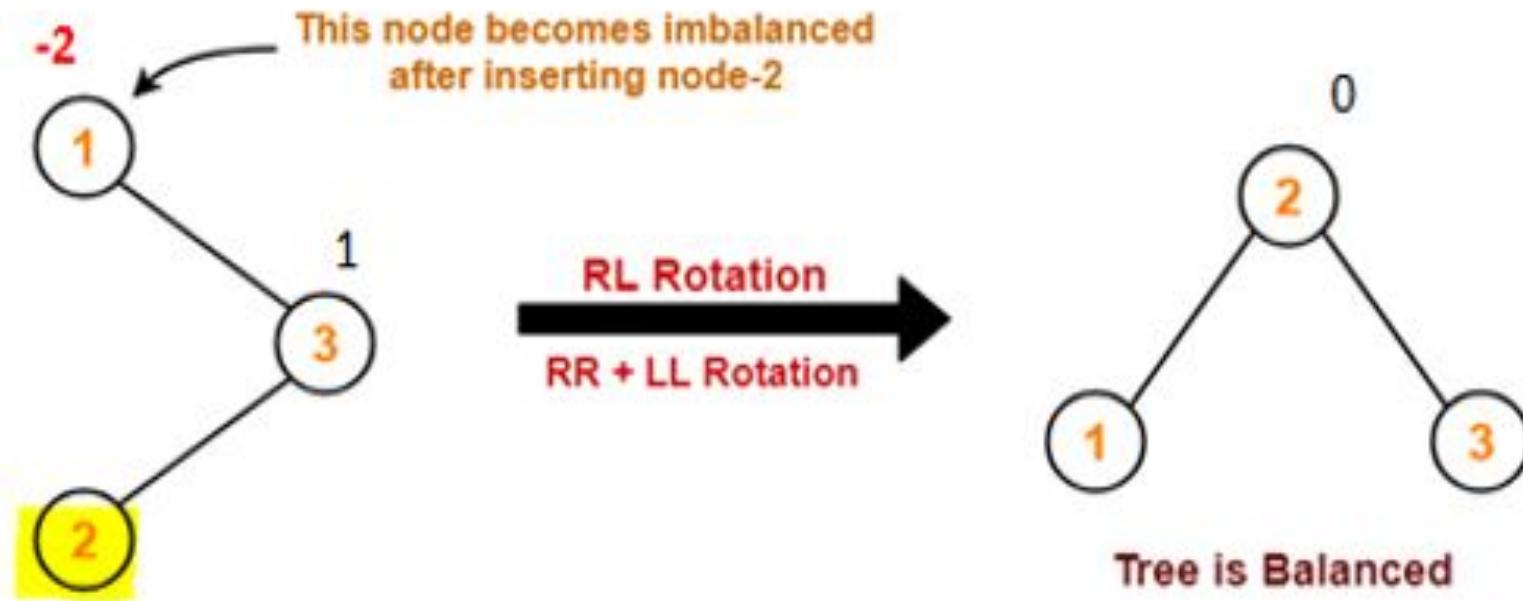




SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

RL Rotation - Example



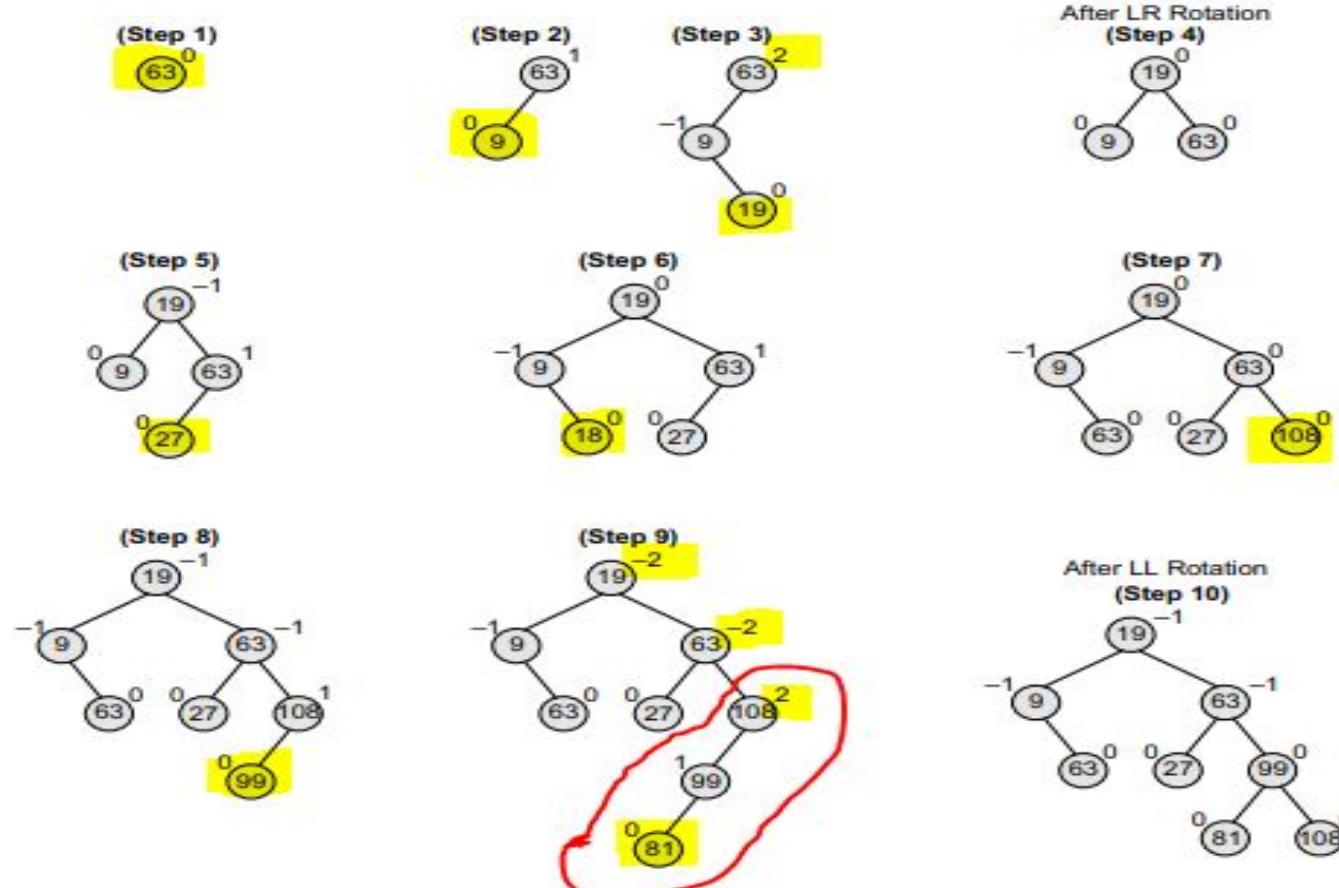


SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

AVL TREE CONSTRUCTION / NODE INSERTION - Example

Construct an AVL tree by inserting the following elements in the given order
63, 9, 19, 27, 18, 108, 99, 81.



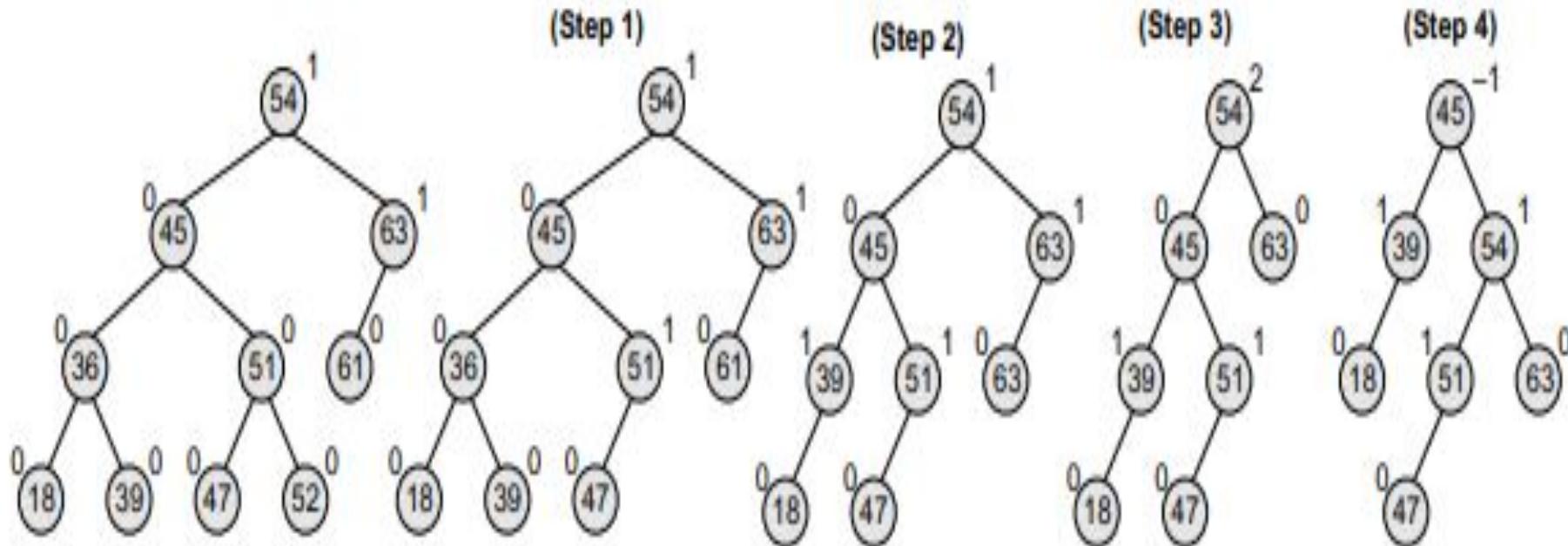


SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

AVL TREE – NODE DELETION - Example

- Delete nodes 52, 36, and 61 from the AVL tree given





SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

B-Trees



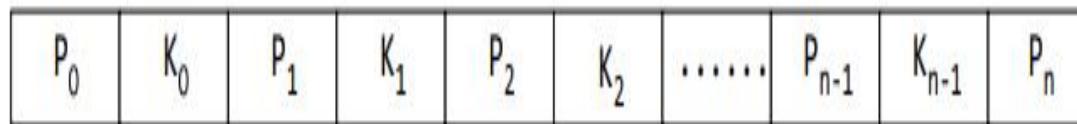
B-Trees

- A B-tree is called as an m -way tree where each node is allowed to have a maximum of m children.
- Its order is m .
- The number of keys in each non-leaf node is $m-1$.
- Leaves should be in the same level and should contain no more than $m-1$ keys.
- Non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children.
- The root can be either leaf node or it can have two to m children.



SRM Structure of an m -way search tree node

- The structure of an m -way search tree node is shown in figure



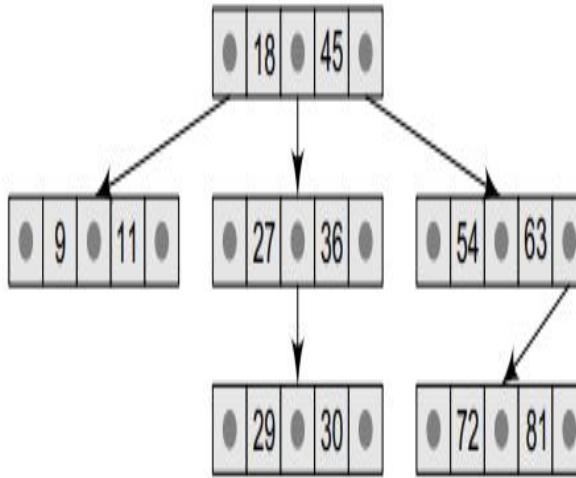
- Where $P_0, P_1, P_2, \dots, P_n$ are pointers to the node's sub-trees and $K_0, K_1, K_2, \dots, K_{n-1}$ are the key values of the node.
- All the key values are stored in ascending order.
- A B tree is a specialized m -way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access.

Source :

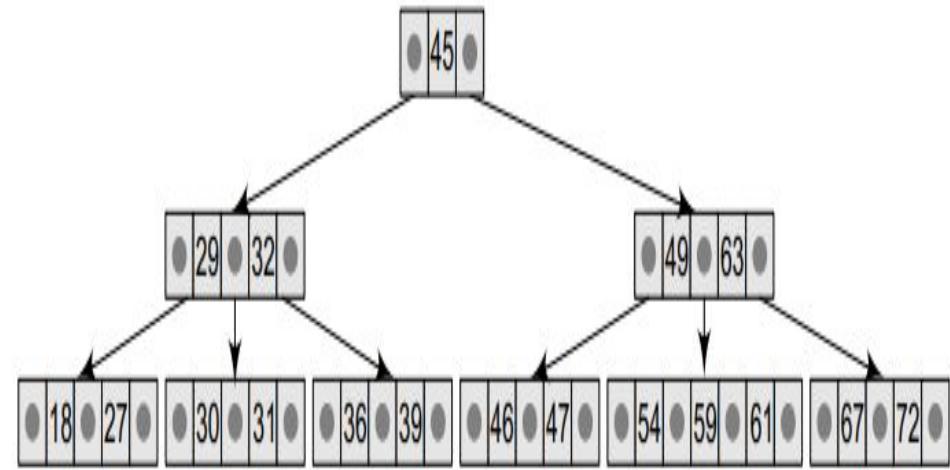
<http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>



B-Trees - Examples



B-Tree of order 3



B-Tree of order 4

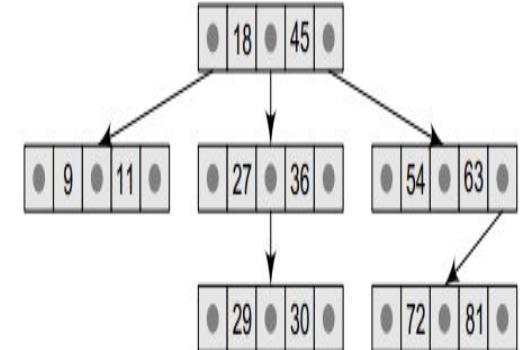
Source :

<http://masterragh.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>



Searching in a B-Tree

- Similar to searching in a binary search tree.
- Consider the B-Tree shown here.
- If we wish to search for 72 in this tree, first consider the root, the search key is greater than the values in the root node. So go to the right sub-tree.
- The right sub tree consists of two values and again 72 is greater than 63 so traverse to right sub tree of 63.
- The right sub tree consists of two values 72 and 81. So we found our value 72.



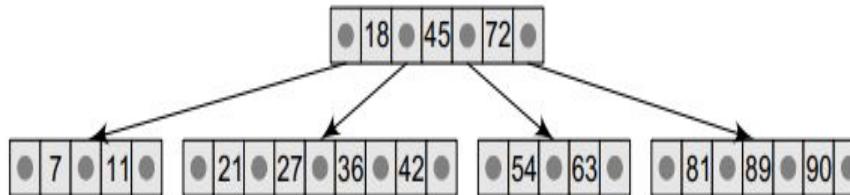
Insertion in a B-Tree

- Insertions are performed at the leaf level.
- Search the B-Tree to find suitable place to insert the new element.
- If the leaf node is not full, the new element can be inserted in the leaf level.
- If the leaf is full,
 - insert the new element in order into the existing set of keys.
 - split the node at its median into two nodes.
 - push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.



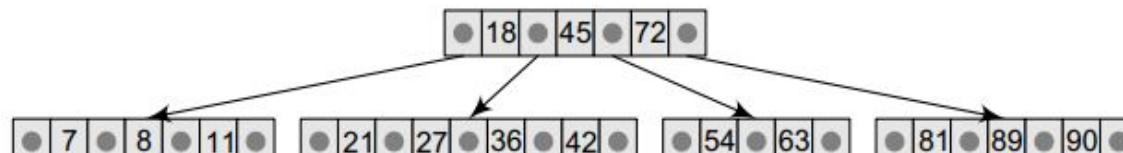
Insertion - Example

- Consider the B-Tree of order 5

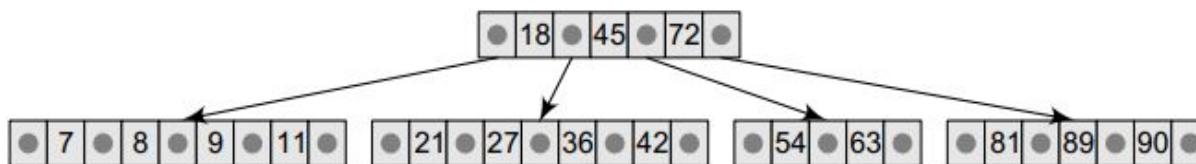


- Try to insert 8, 9, 39 and 4 into it.

Step 1: Insert 8



Step 2: Insert 9



Source :

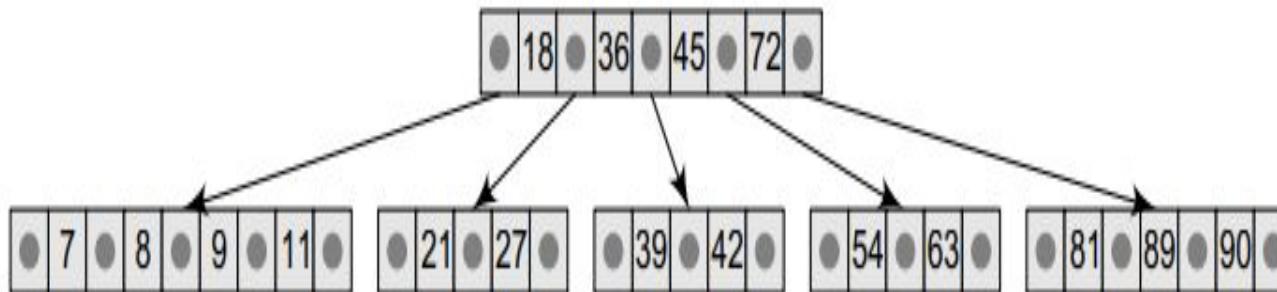
<http://masterragh.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>



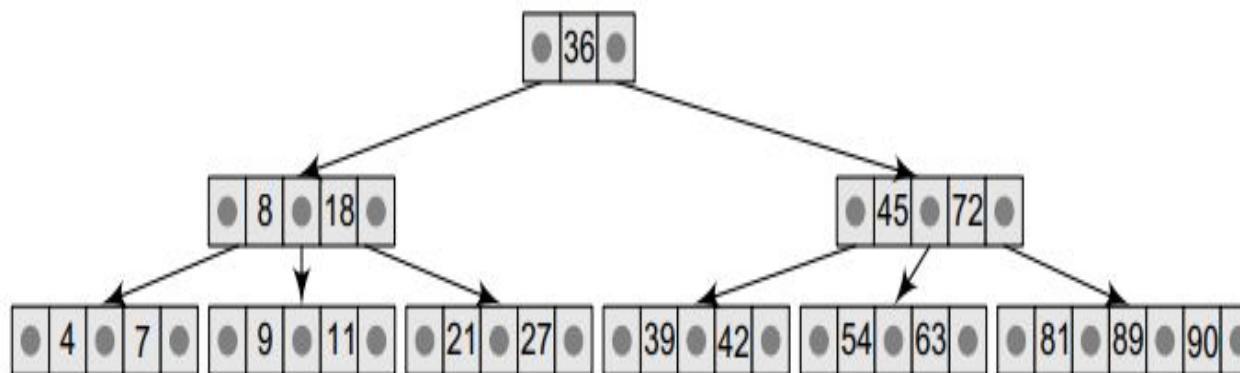
SRM INSTITUTION OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

Insertion - Example

Step 3: Insert 39



Step 4: Insert 4



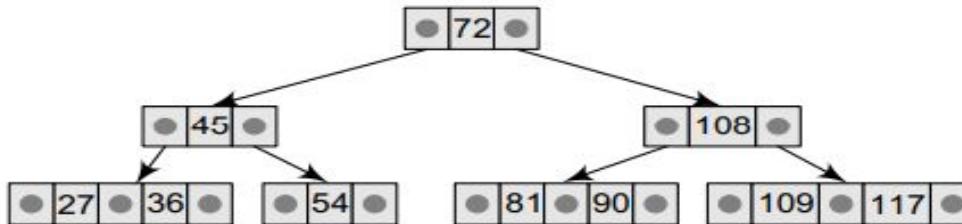
Source :

<http://masterraghlu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>



Exercise

- Consider the B-Tree of order 3, try to insert 121 and 87.



- Create a B-Tree of order 5, by inserting the following elements
 - 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.

Deletion in a B-Tree

- Like insertion, deletion also should be performed from leaf nodes.
- There are two cases in deletion.
 - To delete a leaf node.
 - To delete an internal node.
- Deleting leaf node
 - Search for the element to be deleted, if it is in the leaf node and the leaf node has more than $m/2$ elements then delete the element.
 - If the leaf node does not contain $m/2$ elements then take an element from either left or right sub tree.
 - If both the left and right sub tree contain only minimum number of elements then create a new leaf node.



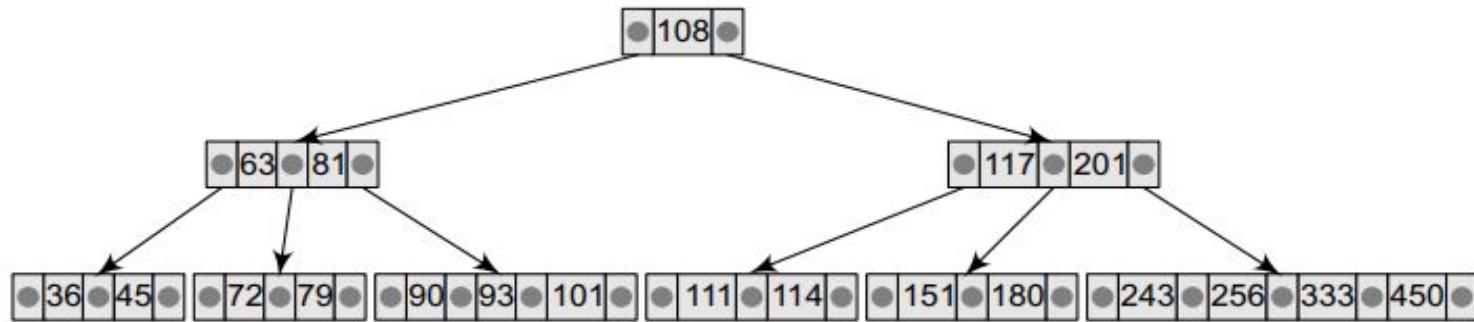
Deletion in a B-Tree

- Deleting an internal node
 - If the element to be deleted is in an internal node then find the predecessor or successor of the element to be deleted and place it in the deleted element position.
 - The predecessor or successor of the element to be deleted will always be in the leaf node.
 - So the procedure will be similar to deleting an element from the leaf node.

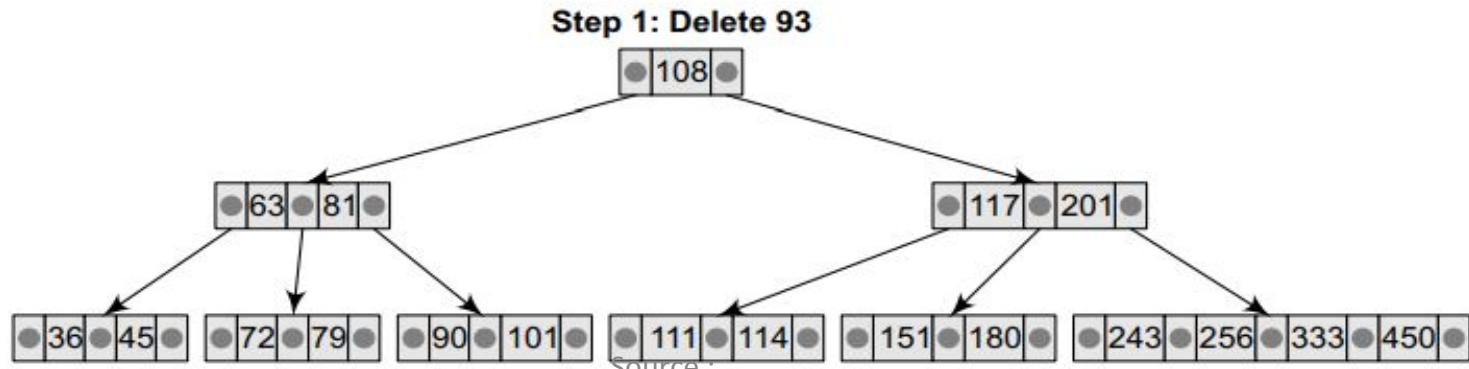


Deletion - Example

- Consider the B-Tree of order 5



- Try to delete values 93, 201, 180 and 72 from it.



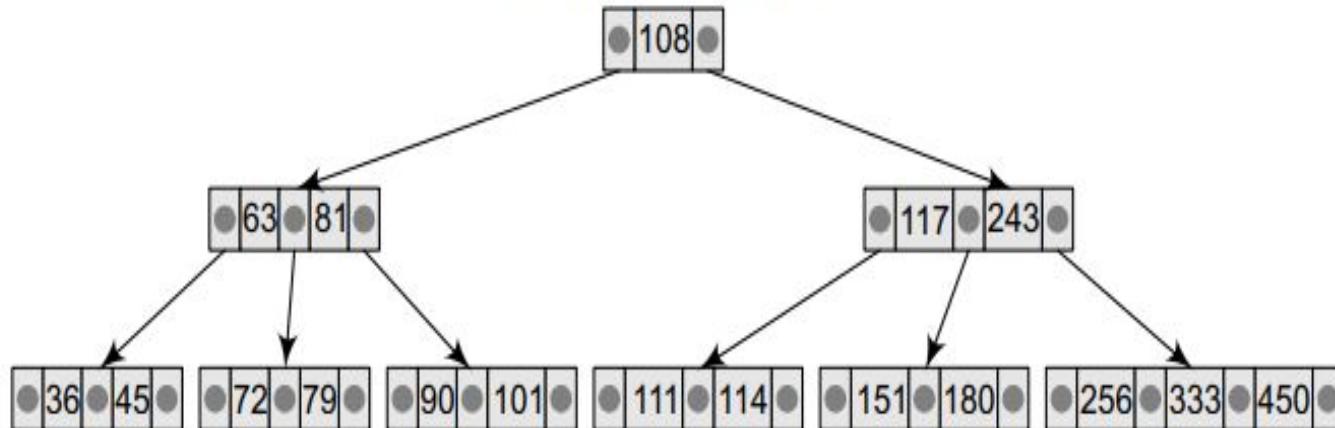
Source :

<http://masterragh.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>

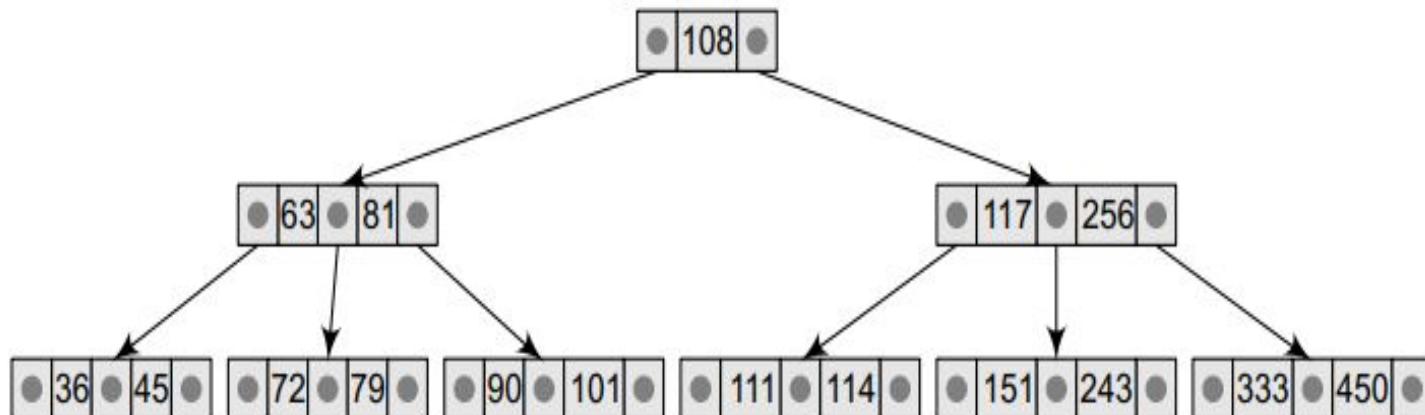


Deletion - Example

Step 2: Delete 201



Step 3: Delete 180



Source :

<http://masterraghlu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>

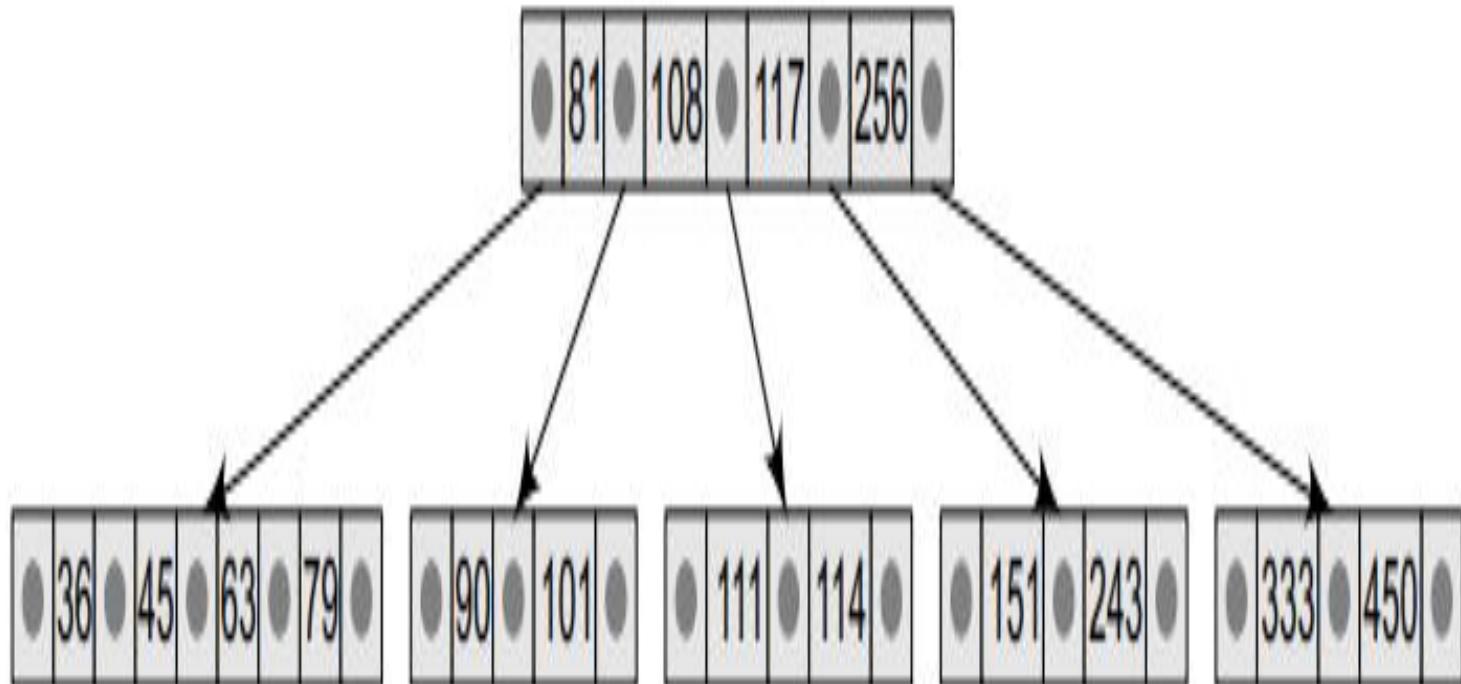


SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Deletion - Example

Step 4: Delete 72



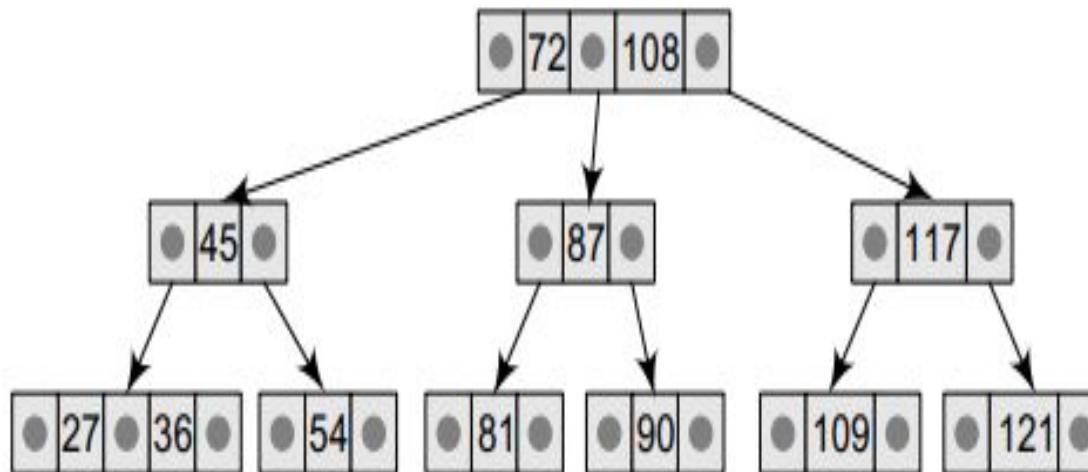
Source :

<http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>



Exercise

- Consider the B-Tree of order 3, try to delete 36 and 109





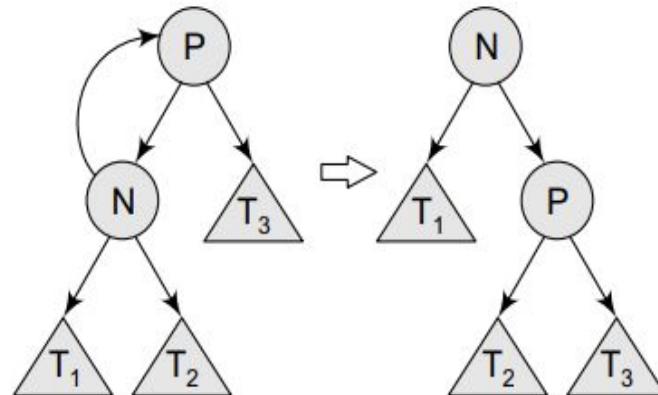
Splay Trees

- Self-balancing BST with an additional property that recently accessed elements can be re-accessed fast.
- All operations can be performed in $O(\log n)$ time.
- All operations can be performed by combining with the basic operation called splaying.
- Splaying the tree for a particular node rearranges the tree to place that node at the root.
- Each splay step depends on three factors:
- Whether N is the left or right child of its parent P,
- Whether P is the root or not, and if not,
- Whether P is the left or right child of its parent, G (N's grandparent).



Splay Trees

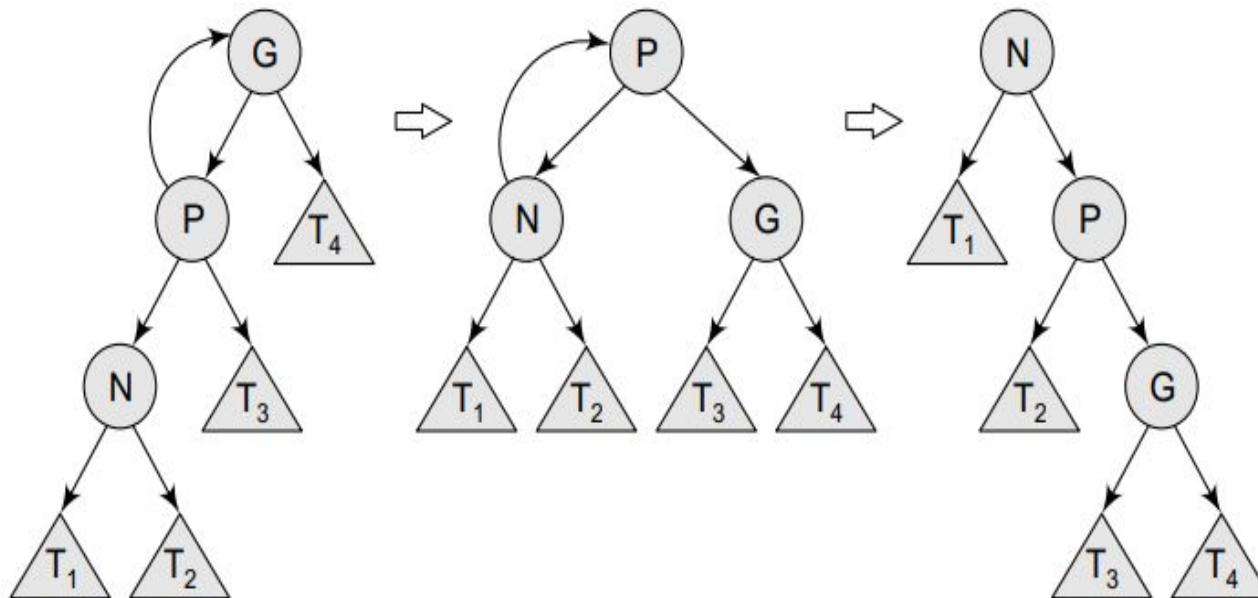
- Depending on these three factors, we have one splay step based on each factor.
- Zig step





Splay Trees

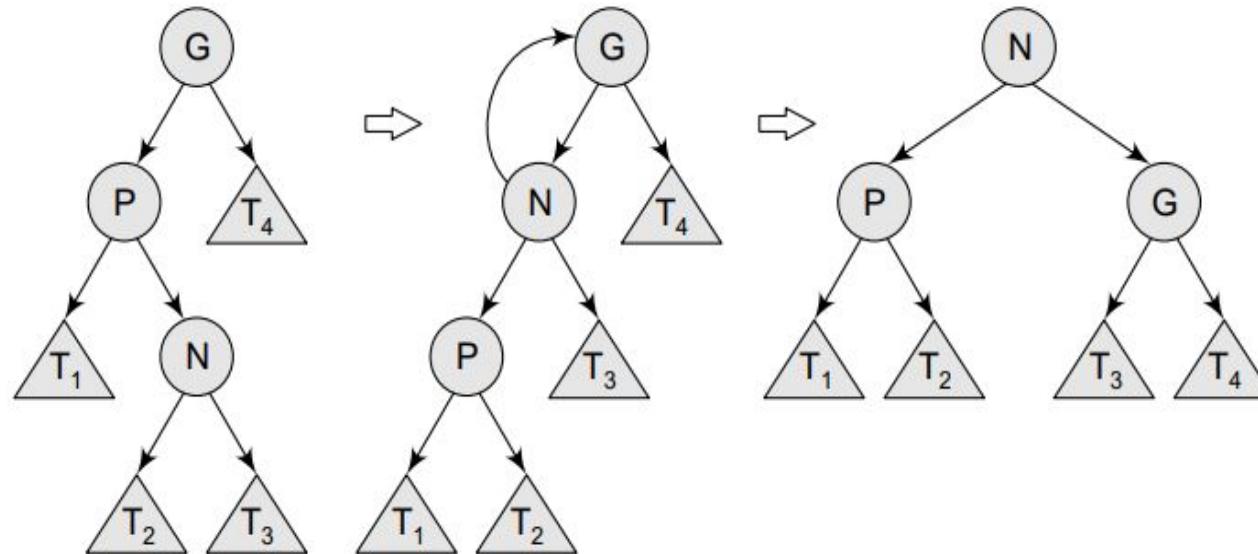
- Zig – Zig Step





Splay Trees

- Zig – Zag Step



Red-Black Trees

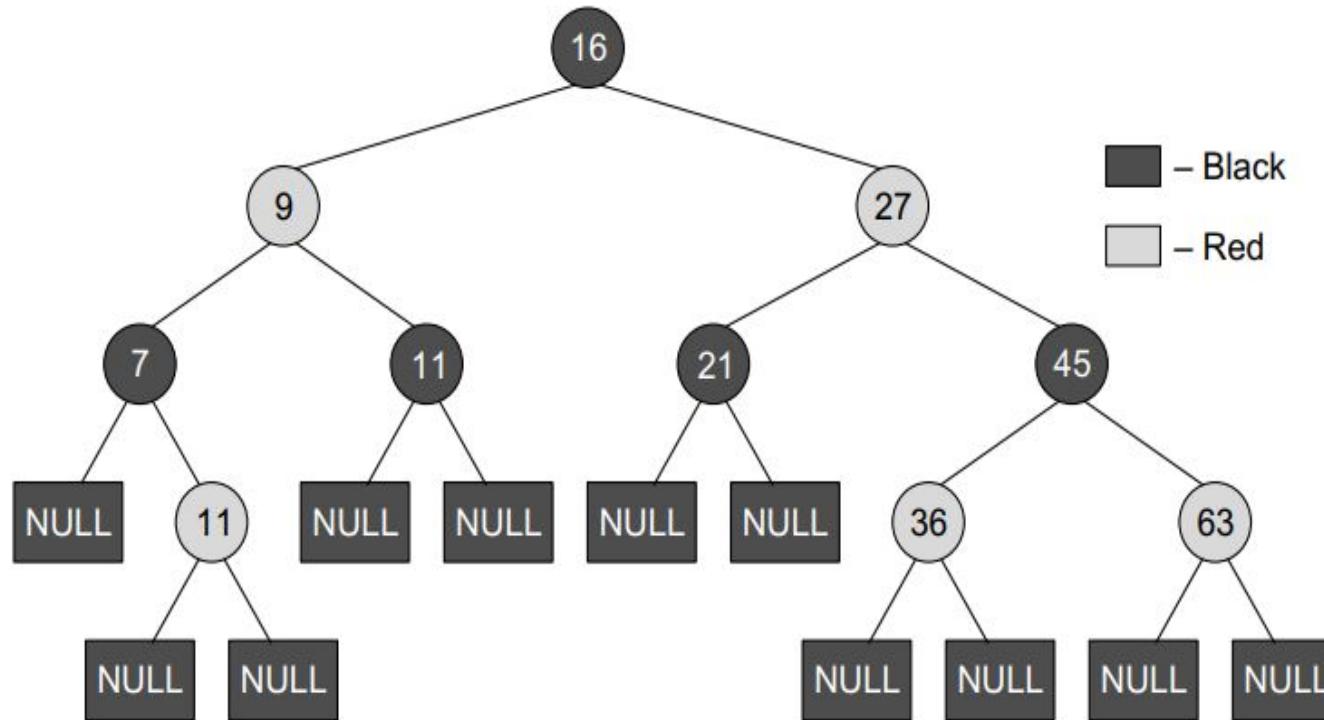
- Self balancing binary search tree
- Also called as symmetric binary B-tree
- Although red-black tree is complex, all operations can be done in a worst case time complexity of $O(\log n)$ where n is the number of nodes in the tree.
- Properties of red-black trees
- A red-black tree is a BST which has the following properties
 1. The color of a node is either red or black.
 2. The color of the root node is always black.
 3. All leaf nodes are black.
 4. Every red node has both the children colored in black.
 5. Every simple path from a given node to any of its leaf nodes has an equal number of black nodes



SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Red-Black Tree - Example



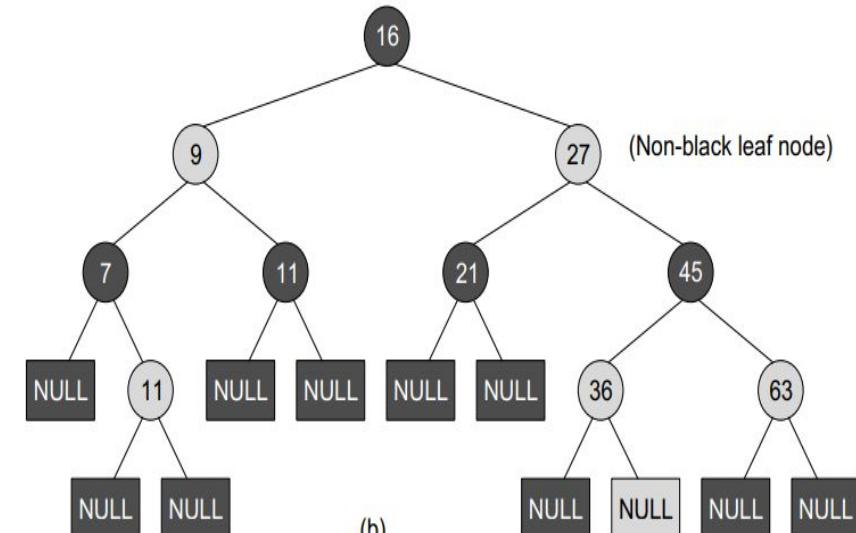
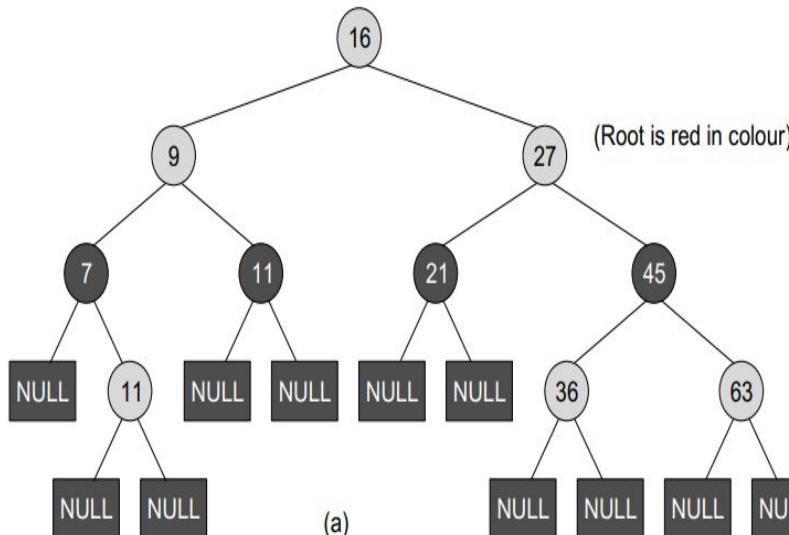
Source :

<http://masterragh.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>



Exercise

- Say whether the following trees are red-black or not.



Source :

<http://masterraghlu.com/subjects/Datastructuress/ebooks/rema%20thareja.pdf>



SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Red Black Trees - Insertion

- Insertion operation starts in the same way as we add new node in the BST.
- The difference is, in BST the new node is added as a leaf node whereas in red-black tree there is no data in the leaf node.
- So we add the new node as a red interior node which has two black child nodes.
- When a new node is added, it may violate some properties of the red-black tree.
- So in order to restore their property, we check for certain cases and restore the property depending on the case that turns up after insertion.
- Terminology
- Grandparent node (G) of node (N) - parent of N's parent (P)
- Uncle node (U) of node (N) - sibling of N's parent (P)



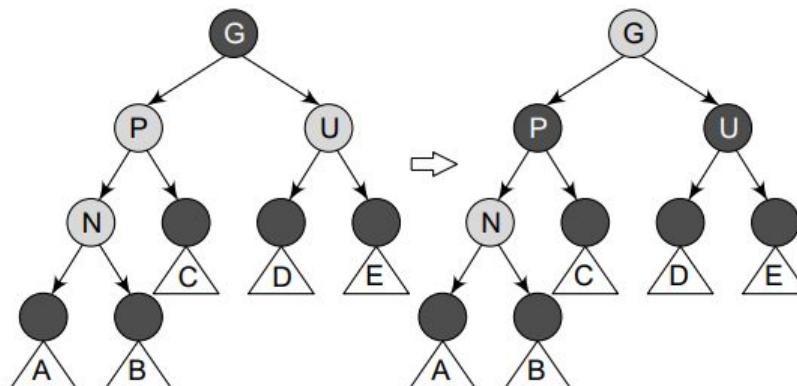
Red Black Trees - Insertion

- When we insert a new node in a red-black tree, note the following:
- All leaf nodes are always black. So property 3 always holds true.
- Property 4 (both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.
- Property 5 (all paths from any given node to its leaf nodes has equal number of black nodes) is threatened only by adding a black node, repainting a red node black, or a rotation.
- Case 1: The new node N is added as the root of the Tree
 - In this case, N is repainted black, as the root should be black always.
- Case 2: The new node's parent P is black
 - In this case, both children of every red node are black, so Property 4 is not invalidated. Property 5 is also not threatened.



Red Black Trees - Insertion

- Case 3: If Both the Parent (P) and the Uncle (U) are Red
- In this case, Property 5 which says all paths from any given node to its leaf nodes have an equal number of black nodes is violated.



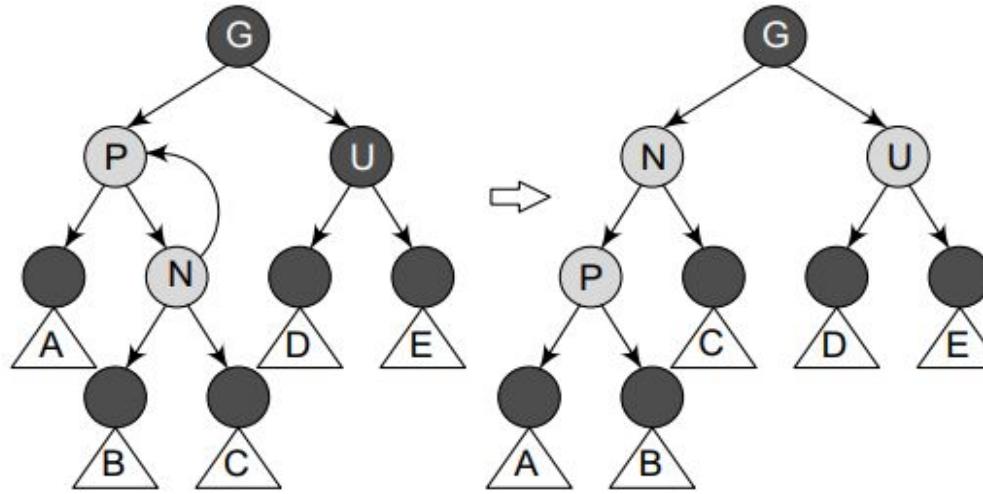
Source :

[http://masterraghur.com/subjects/Datastructu
res/ebooks/rema%20thareja.pdf](http://masterraghur.com/subjects/Datastructu res/ebooks/rema%20thareja.pdf)



Red Black Trees - Insertion

- Case 4: The Parent P is Red but the Uncle U is Black and N is the Right Child of P and P is the Left Child of G



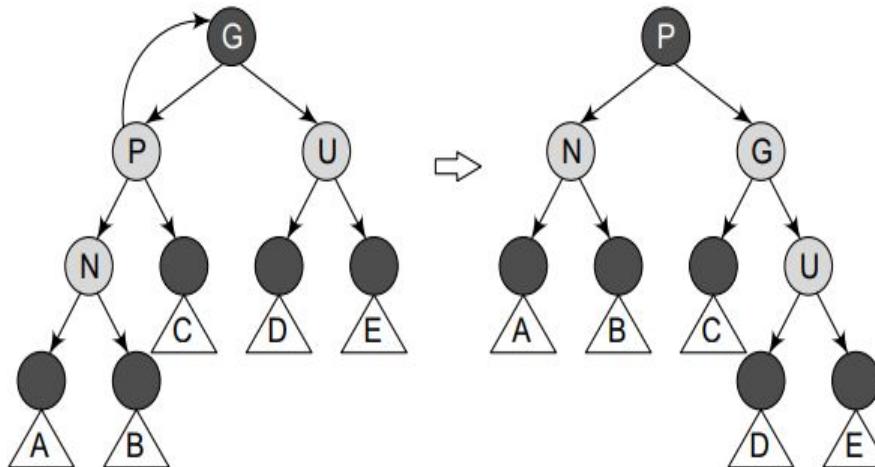
Source :

<http://masterraghur.com/subjects/Datastructuress/ebooks/rema%20thareja.pdf>



Red Black Trees - Insertion

- Case 5: The parent P is red but the uncle U is black and the new node N is the left child of P, and P is the left child of its parent G.



Source :

<http://masterraghul.com/subjects/Datastructures/ebooks/rema%20thareja.pdf>

References

1. Reema Thareja, Data Structures Using C, 1st ed., Oxford Higher Education, 2011
2. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, Introduction to Algorithms 3rd ed., The MIT Press Cambridge, 2014
3. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
4. <http://masterraghu.com/subjects/Datastructures/ebooks/remathareja.pdf>



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.
18CSC201J

DATA STRUCTURES AND ALGORITHMS

Unit- V





SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

- GRAPH TERMINOLOGY
- GRAPH TRAVERSAL



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

- A graph - an abstract data structure that is used to implement the graph concept from mathematics.
- A collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship between tree nodes.
- Any kind of complex relationships between the nodes can be represented.



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

Why graphs are useful?

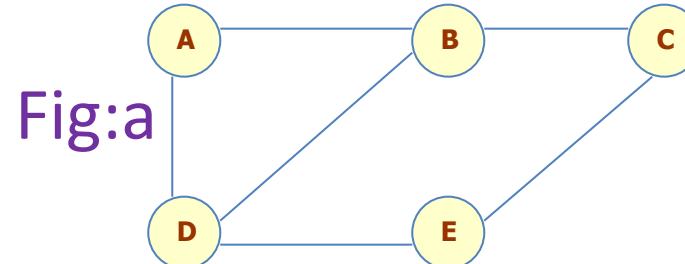
- Graphs are widely used to model any situation where entities or things are related to each other in pairs; for example, the following information can be represented by graphs:
- *Family trees* in which the member nodes have an edge from parent to each of their children.
- *Transportation networks* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Definition

- A graph G is defined as an ordered set (V, E) , where $V(G)$ represent the set of vertices and $E(G)$ represents the edges that connect the vertices.
- The figure given shows a graph with $V(G) = \{ A, B, C, D \}$ and $E(G) = \{ (A, B), (B, C), (A, D), (B, D), (D, E), (C, E) \}$. Note that there are 5 vertices or nodes and 6 edges in the graph.





SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**

CHEENNAI.

A graph can be directed (Fig a) or undirected (Fig b).

Fig:a

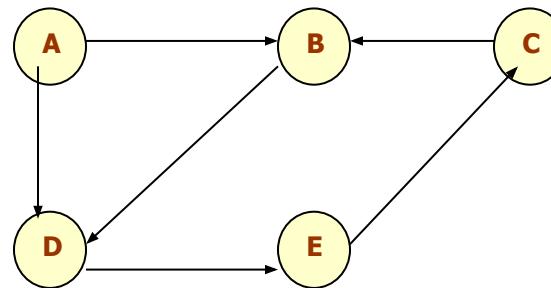
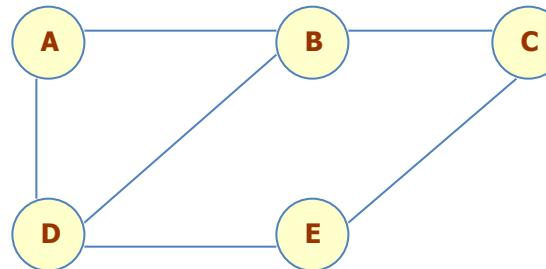


Fig:b





SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Graph Terminology:

Adjacent Nodes or Neighbors:

For every edge, $e = (u, v)$ that connects nodes u and v ; the nodes u and v are the end-points and are said to be the adjacent nodes or neighbors.

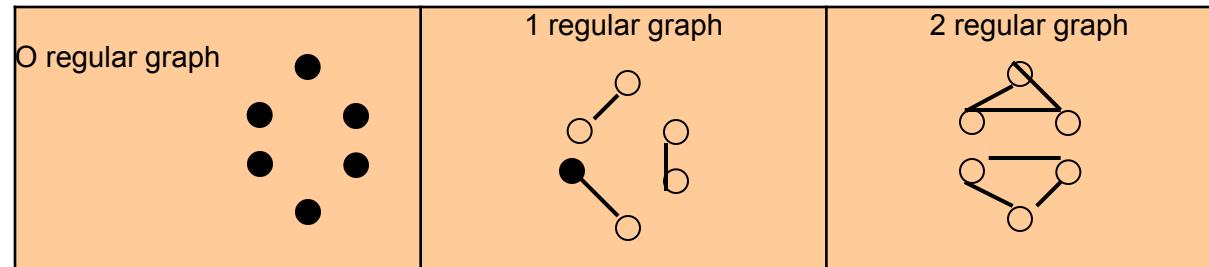
Degree of a node:

Degree of a node u , $\deg(u)$, is the total number of edges containing the node u . If $\deg(u) = 0$, it means that u does not belong to any edge and such a node is known as an isolated node.



SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Regular graph: Regular graph is a graph where each vertex has the same number of neighbors. That is every node has the same degree. A regular graph with vertices of degree k is called a k -regular graph or regular graph of degree k .





SR

INSTITUTE OF SCIENCE AND TECHNOLOGY,

- ***Path:*** A path P, written as ~~CHENNAI.~~ {v₀, v₁, v₂, ..., v_n}, of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v₀, v = v_n and v_{i-1} is adjacent to v_i for i = 1, 2, 3, ..., n.
- ***Closed path:*** A path P is known as a closed path if the edge has the same end-points. That is, if v₀ = v_n.
- ***Simple path:*** A path P is known as a simple path if all the nodes in the path are distinct with an exception that v₀ may be equal to v_n. If v₀ = v_n, then the path is called a closed simple path.
- ***Cycle:*** A closed simple path with length 3 or more is known as a cycle. A cycle of length k is called a $k - \text{cycle}$.



SR

INSTITUTE OF SCIENCE AND TECHNOLOGY,

- *Connected graph:* A graph in which there exists a path between any two of its nodes is called a connected graph. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree.
- *Complete graph:* A graph G is said to be a complete, if all its nodes are fully connected, that is, there is a path from one node to every other node in the graph. A complete graph has $n(n-1)/2$ edges, where n is the number of nodes in G .



SR

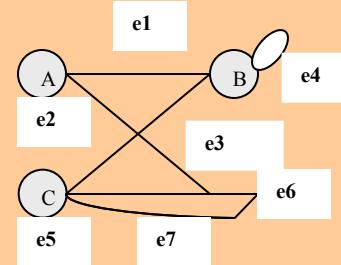
INSTITUTE OF SCIENCE AND TECHNOLOGY,

- *Labeled graph or weighted graph:* A graph is said to be labeled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. Weight of the edge, denoted by $w(e)$ is a positive value which indicates the cost of traversing the edge.
- *Multiple edges:* Distinct edges which connect the same end points are called multiple edges. That is, $e = \{u, v\}$ and $e' = (u, v)$ are known as multiple edges of G .
- *Loop:* An edge that has identical end-points is called a loop. That is, $e = (u, u)$.
- *Multi-graph:* A graph with multiple edges and/or a loop is called a multi-graph.
- *Size of the graph:* The size of a graph is the total number of edges in it.

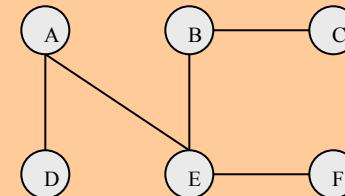


SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

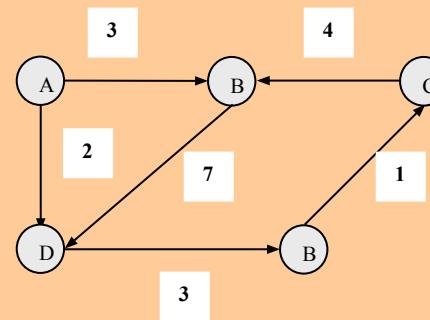
(a) Multi-graph



(b) Tree



(c) Weighted Graph





SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

Directed Graph:

- A directed graph G , also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G . For an edge (u, v) -
 - The edge begins at u and terminates at v
 - U is known as the origin or initial point of e . Correspondingly, v is known as the destination or terminal point of e
 - U is the predecessor of v . Correspondingly, v is the successor of u
 - nodes u and v are adjacent to each other.



SR

INSTITUTE OF SCIENCE AND TECHNOLOGY,

TERMINOLOGIES IN DIRECTED GRAPH: CHENNAI.

- *Out-degree of a node:* The out degree of a node u , written as $\text{outdeg}(u)$, is the number of edges that originate at u .
- *In-degree of a node:* The in degree of a node u , written as $\text{indeg}(u)$, is the number of edges that terminate at u .
- *Degree of a node:* Degree of a node written as $\text{deg}(u)$ is equal to the sum of in-degree and out-degree of that node. Therefore, $\text{deg}(u) = \text{indeg}(u) + \text{outdeg}(u)$
- *Source:* A node u is known as a source if it has a positive out-degree but an in-degree = 0.
- *Sink:* A node u is known as a sink if it has a positive in degree but a zero out-degree.
- *Reachability:* A node v is said to be reachable from node u , if and only if there exists a (directed) path from node u to node v .



SR

INSTITUTE OF SCIENCE AND TECHNOLOGY,

CHENNAI.

- *Strongly connected directed graph:* A digraph is said to be strongly connected if and only if there exists a path from every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.
- *Unilaterally connected graph:* A digraph is said to be unilaterally connected if there exists a path from any pair of nodes u, v in G such that there is a path from u to v or a path from v to u but not both.
- *Parallel/Multiple edges:* Distinct edges which connect the same end points are called multiple edges. That is, $e = \{u, v\}$ and $e' = (u, v)$ are known as multiple edges of G.
- *Simple directed graph:* A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycle with an exception that it cannot have more than one loop at a given node



SR

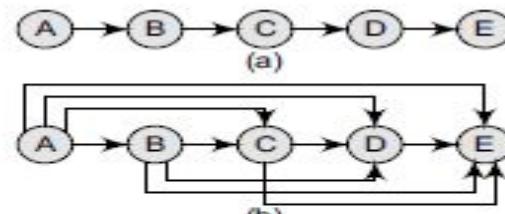
INSTITUTE OF SCIENCE AND TECHNOLOGY.

Transitive Closure of a Directed Graph:

- It is same as the adjacency list in graph terminology.
- It is stored as a matrix T.

Definition

For a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph $G^* = (V, E^*)$. In G^* , for every vertex pair v, w in V there is an edge (v, w) in E^* if and only if there is a valid path from v to w in G .



(a) A graph G and its
(b) transitive closure
 G^*



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

- A binary relation indicates only whether the node A is connected to node B, whether node B is connected to node C, etc.
- But once the transitive closure is constructed as shown in Fig.
- we can easily determine in $O(1)$ time whether node E is reachable from node A or not.

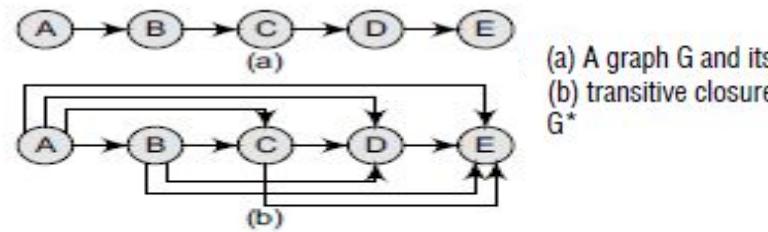


SR

INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Definition

For a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph $G^* = (V, E^*)$. In G^* , for every vertex pair v, w in V there is an edge (v, w) in E^* if and only if there is a valid path from v to w in G .



- A binary relation indicates only whether the node A is connected to node B, whether node B is connected to node C, etc. But once the transitive closure is constructed as shown in Fig. we can easily determine in $O(1)$ time whether node E is reachable from node A or not.



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Graph Representation:

- There are three common ways of storing graphs in the computer's memory.
- They are:
 - *Sequential representation* by using an adjacency matrix.
 - *Linked representation* by using an adjacency list that stores the neighbors of a node using a linked list.
 - *Adjacency multi-list* which is an extension of linked representation.

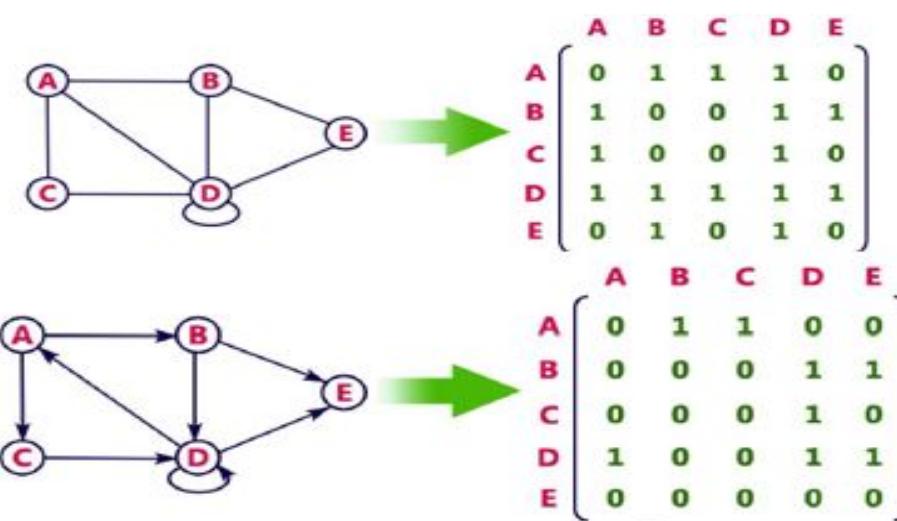


SR

INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Adjacency Matrix Representation:

- An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.
- In this representation, graph can be represented using a matrix of size $V \times V$.
- No matter how few edges the graph has, the matrix has $O(n^2)$ in memory.





SR

INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

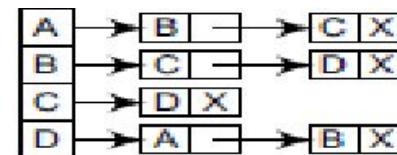
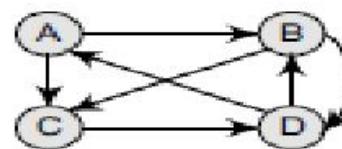
Adjacency List Representation:

- An adjacency list is another way in which graphs can be represented in the computer's memory.
- This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.
- The key advantages of using an adjacency list are:
 - It is easy to follow and clearly shows the adjacent nodes of a particular node.
 - It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
 - Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

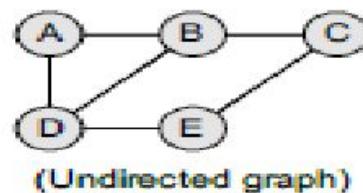


SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

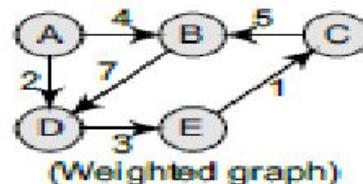
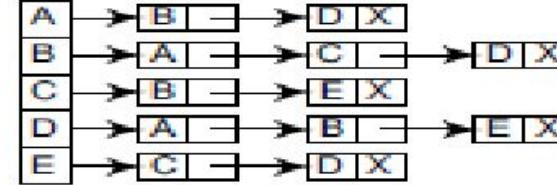
Adjacency List Representation:



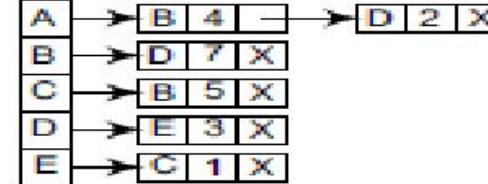
Graph G and its adjacency list



(Undirected graph)



(Weighted graph)



Adjacency list for an undirected graph
and a weighted graph



SR

INSTITUTE OF SCIENCE AND TECHNOLOGY,

Adjacency Multi-list Representation.

- Graphs can also be represented using multi-lists which can be said to be modified version of adjacency lists.
- Adjacency multi-list is an edge-based rather than a vertex-based representation of graphs.
- A multi-list representation basically consists of two parts—
 - a directory of nodes' information and a set of linked lists storing information about edges.
 - While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge).
 - For example, the directory entry for node i points to the adjacency list for node i . This means that the nodes are shared among several lists.



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

In a multi-list representation, the information about an edge (vi, vj) of an undirected graph can be stored using the following attributes:

M: A single bit field to indicate whether the edge has been examined or not.

vi : A vertex in the graph that is connected to vertex vj by an edge.

vj : A vertex in the graph that is connected to vertex vi by an edge.

Link i for vi : A link that points to another node that has an edge incident on vi .

Link j for vi : A link that points to another node that has an edge incident on vj .

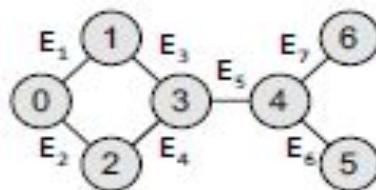


SR

INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Adjacency Multi-list Representation:

Consider the undirected graph given in Fig. The adjacency multi-list for the graph can be given as:



Edge 1		0	1	Edge 2	Edge 3
Edge 2		0	2	NULL	Edge 4
Edge 3		1	3	NULL	Edge 4
Edge 4		2	3	NULL	Edge 5
Edge 5		3	4	NULL	Edge 6
Edge 6		4	5	Edge 7	NULL
Edge 7		4	6	NULL	NULL



SR

INSTITUTE OF SCIENCE AND TECHNOLOGY,

CHEENAI.

GRAPH TRAVERSAL ALGORITHMS:

- This technique is used for searching a vertex in a graph.
- It is also used to decide the order of vertices to be visit in the search process.
- There are two standard methods of graph traversal:
 1. Breadth-first search
 2. Depth-first search
- While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing.
- The depth-first search scheme uses a stack.



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

Breadth-First Search Algorithm:

- It is a graph search algorithm that begins at the root node and explores all the neighboring nodes.
- Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal.
- This technique is used for searching a vertex in a graph.
- It produces a spanning tree as a final result.
- Spanning tree is a graph without any loops.
- Here we use Queue data structure with maximum size of total number of vertices in the graph.



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

Breadth-First Search Algorithm:

STEPS:

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



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Breadth-First Search Algorithm:

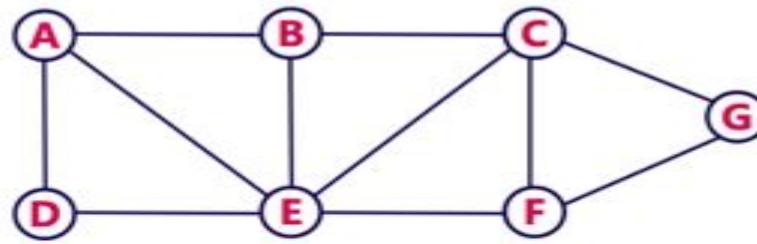
```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
                (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
                (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
                (whose STATUS = 1) and set
        their STATUS = 2
                (waiting state)
                [END OF LOOP]
Step 6: EXIT
```



SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

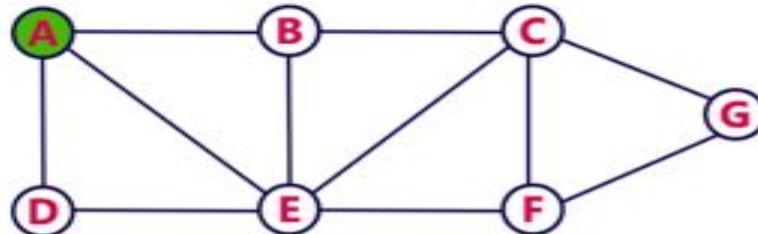
Breadth-First Search Algorithm:

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



Queue



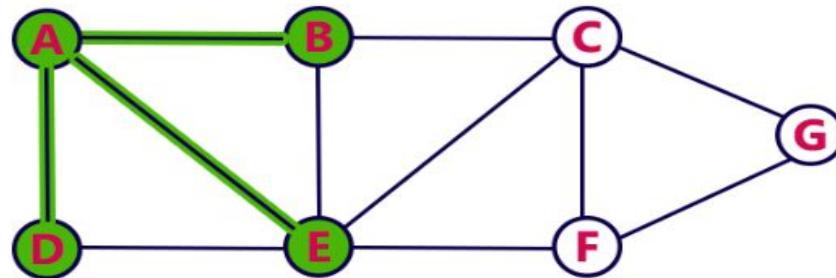


SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Breadth-First Search Algorithm:

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

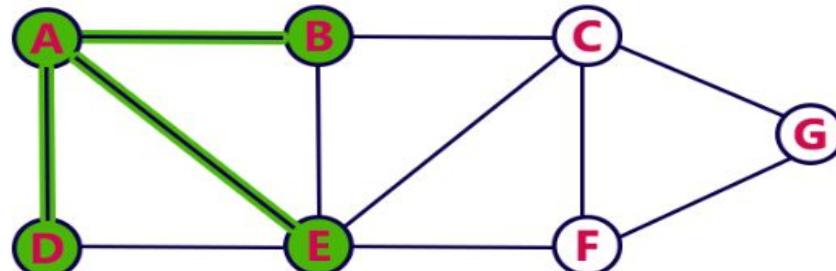


Queue



Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



Queue



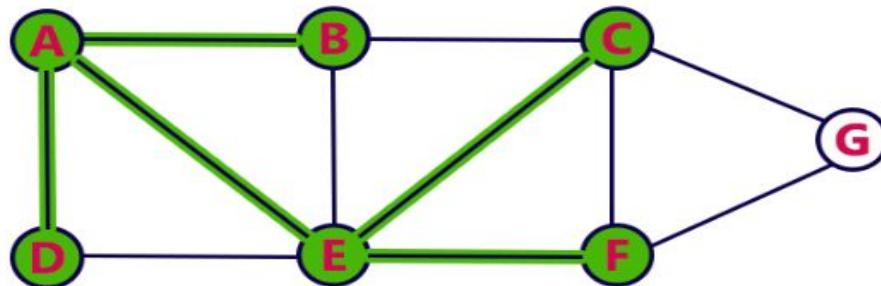


SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Breadth-First Search Algorithm:

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C, F**).
- Insert newly visited vertices into the Queue and delete **E** from the Queue.

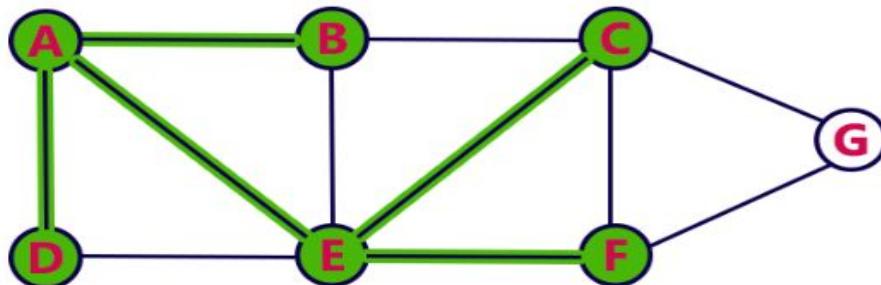


Queue



Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



Queue



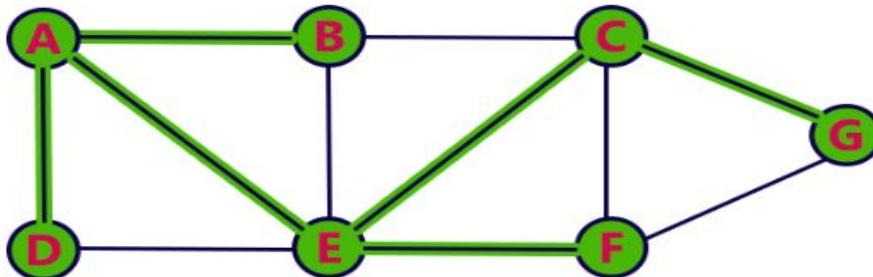


SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Breadth-First Search Algorithm:

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

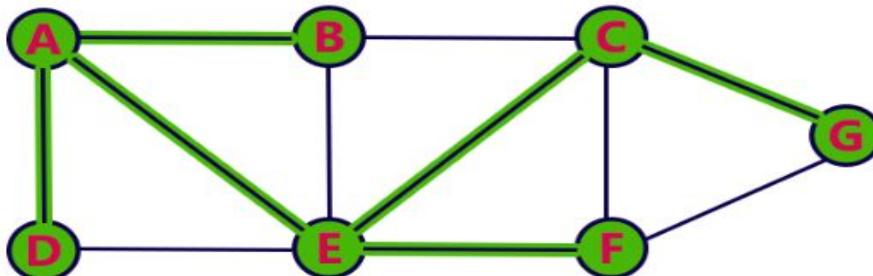


Queue



Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Queue



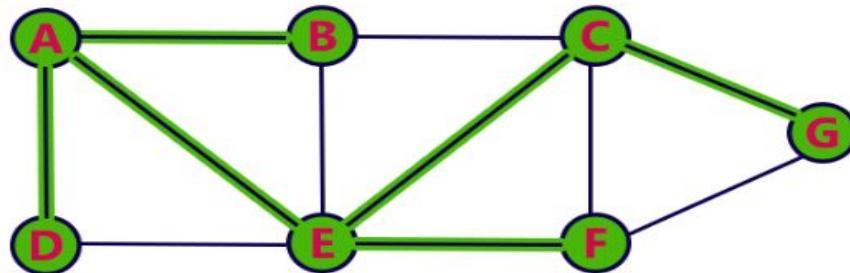


SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Breadth-First Search Algorithm:

Step 8:

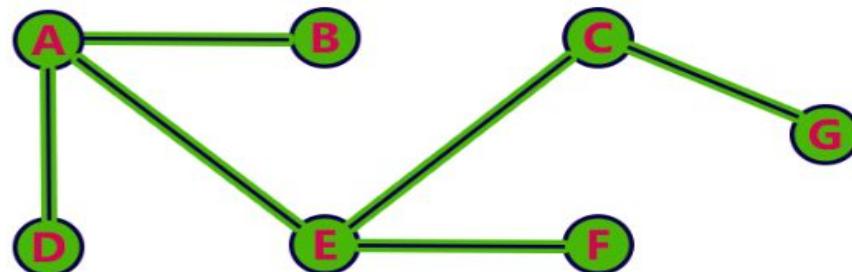
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...





SR

INSTITUTE OF SCIENCE AND TECHNOLOGY,

Features of Breadth-First Search Algorithm:

- ***Space complexity***

- In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated.
- The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d , the asymptotic space complexity is the number of nodes at the deepest level $O(bd)$.

- ***Time complexity***

- In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches $O(bd)$.
- However, the time complexity can also be expressed as $O(|E| + |V|)$, since every vertex and every edge will be explored in the worst case.



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

• *Completeness*

- Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph.
- But in case of an infinite graph where there is no possible solution, it will diverge.

• *Optimality*

- Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node.
- But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

Applications of Breadth-First Search Algorithm:

- Breadth-first search can be used to solve many problems such as:
- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

Depth-first Search Algorithm:

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbor of neighbor of A, and so on.



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Depth-first Search Algorithm:

- During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
- This technique is used for searching a vertex in a graph.
- It produces a spanning tree as a final result.
- Spanning tree is a graph without any loops.
- Here we use Stack data structure with maximum size of total number of vertices in the graph.



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Depth-first Search Algorithm:

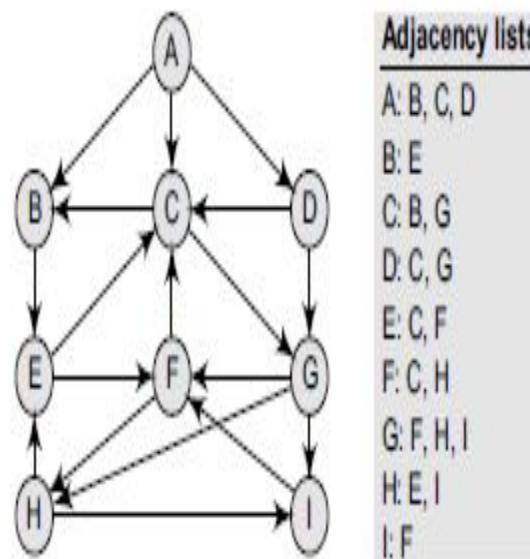
```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
          [END OF LOOP]
Step 6: EXIT
```



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Depth-first Search Algorithm:

Consider the graph and find the spanning tree,





SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

Depth-first Search Algorithm:

-
(a) Push H onto the stack.

STACK: H

- (b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H

STACK: E, I

- (c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I

STACK: E, F

- (d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F

STACK: E, C

- (e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C

STACK: E, B, G



SR INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Depth-first Search Algorithm:

- (f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G

STACK: E, B

- (g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B

STACK: E

- (h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E

STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

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

These are the nodes which are reachable from the node H.



SR

INSTITUTE OF SCIENCE AND TECHNOLOGY,

Features of Depth-First Search Algorithm

Space complexity The space complexity of a depth-first search is lower than that of a breadthfirst search.

Time complexity The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as $(O(|V| + |E|))$.

Completeness Depth-first search is said to be a complete algorithm. If there is a solution, depthfirst search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.



SR

**INSTITUTE OF SCIENCE AND
TECHNOLOGY,
CHENNAI.**

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

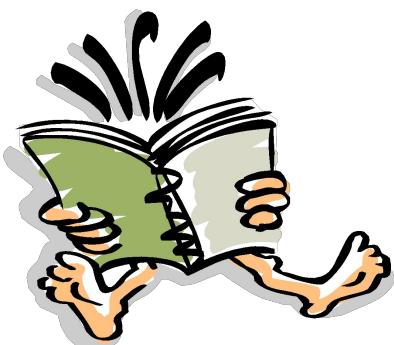
- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

References:

**ReemaThareja, “Data Structures Using C”,
Oxford Higher Education , First Edition,
2011**

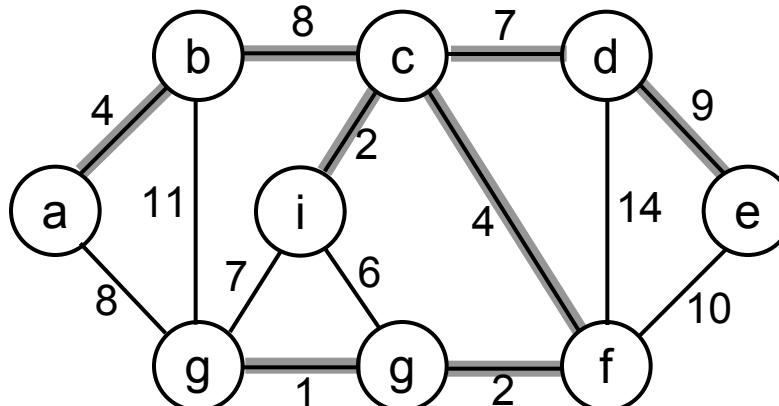
Minimum Spanning Tree

Minimum Spanning Trees (MST)



Minimum Spanning Trees

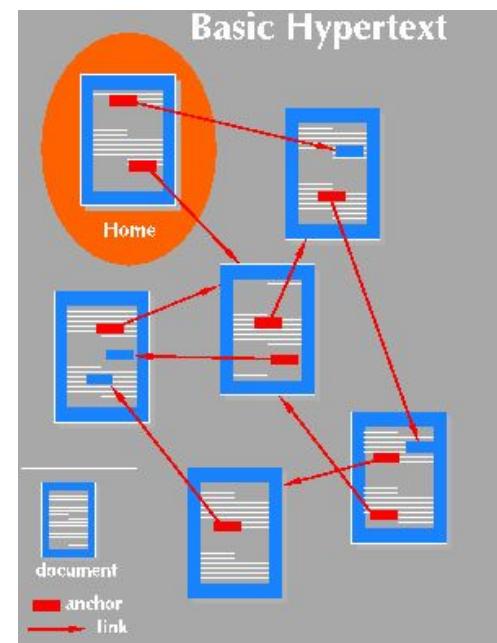
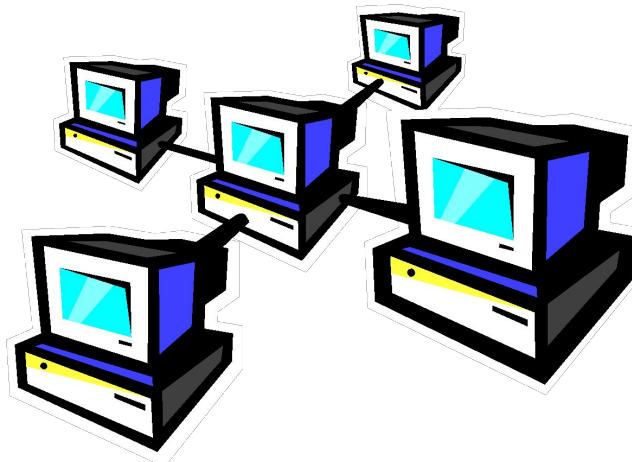
- **Spanning Tree**
 - A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph
- **Minimum Spanning Tree**
 - Spanning tree with the **minimum sum of weights**



- **Spanning forest**
 - If a graph is not connected, then there is a spanning tree for each connected component of the graph

Applications of MST

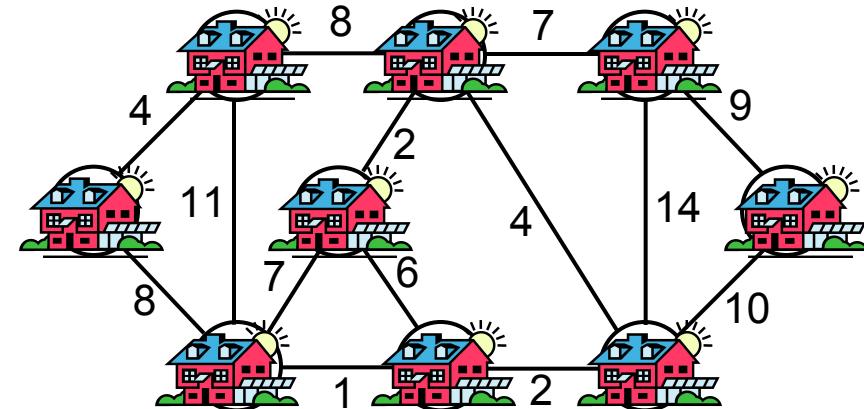
- Find the least expensive way to connect a set of cities, terminals, computers, etc.



Example

Problem

- A town has a set of houses and a set of roads
- A road connects 2 and only 2 houses
- A road connecting houses u and v has a repair cost $w(u, v)$



Goal: Repair enough (and no more) roads such that:

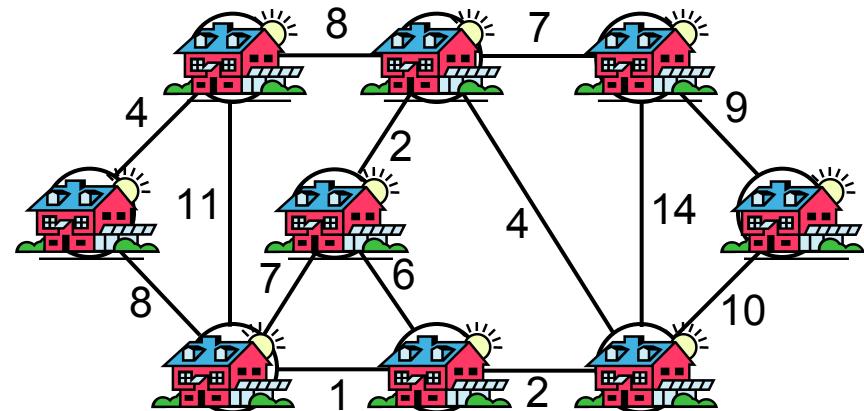
1. Everyone stays connected
i.e., can reach every house from all other houses
2. Total repair cost is minimum

Minimum Spanning Trees

- A connected, undirected graph:
 - Vertices = houses, Edges = roads
- A **weight** $w(u, v)$ on each edge (u, v) in E

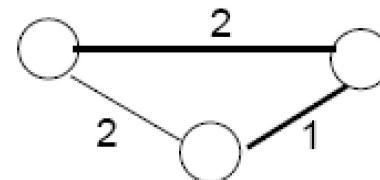
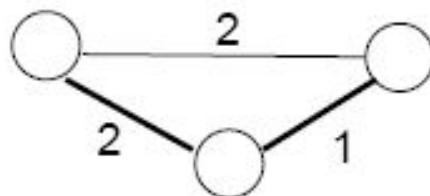
Find $T \subseteq E$ such that:

1. T connects all vertices
2. $w(T) = \sum_{(u,v) \in T} w(u, v)$ is minimized



Properties of Minimum Spanning Trees

- Minimum spanning tree is **not** unique



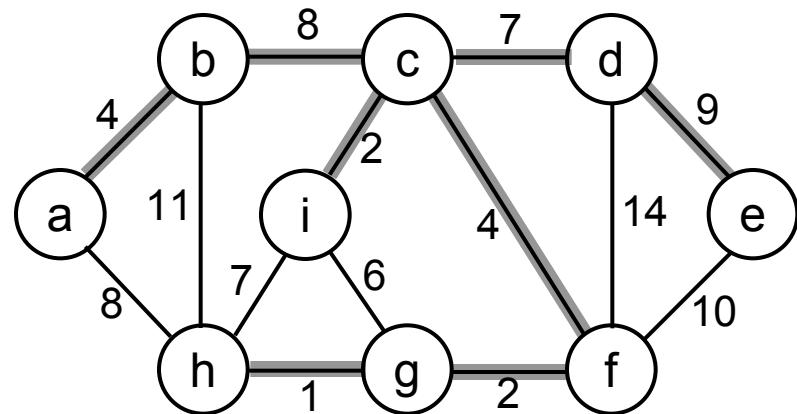
- MST has no cycles – see why:
 - We can take out an edge of a cycle, and still have the vertices connected while reducing the cost
- # of edges in a MST:
 - $|V| - 1$

Growing a MST – Generic Approach

- Grow a set A of edges (initially empty)
- Incrementally add edges to A such that they would belong to a MST

Idea: **add only “safe” edges**

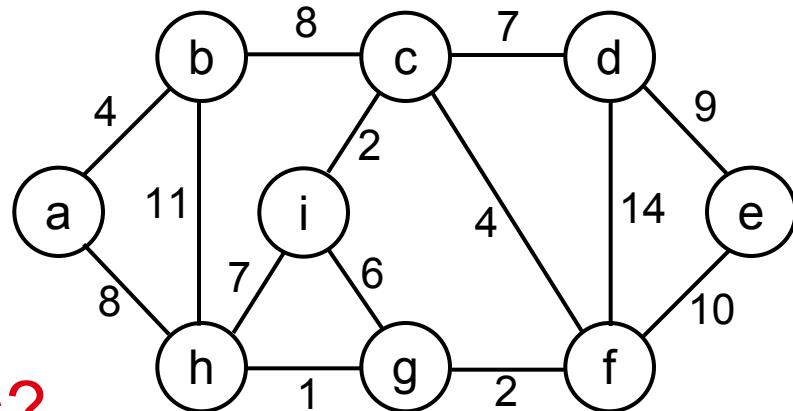
- An edge (u, v) is **safe** for A if and only if $A \cup \{(u, v)\}$ is also a subset of **some** MST



Generic MST algorithm

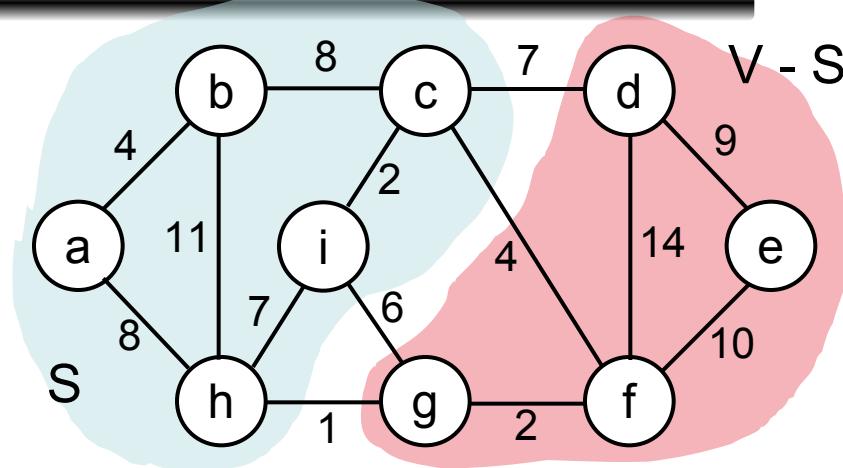
1. $A \leftarrow \emptyset$
2. **while** A is not a spanning tree
3. **do** find an edge (u, v) that is **safe** for A
4. $A \leftarrow A \cup \{(u, v)\}$
5. **return** A

- How do we find safe edges?



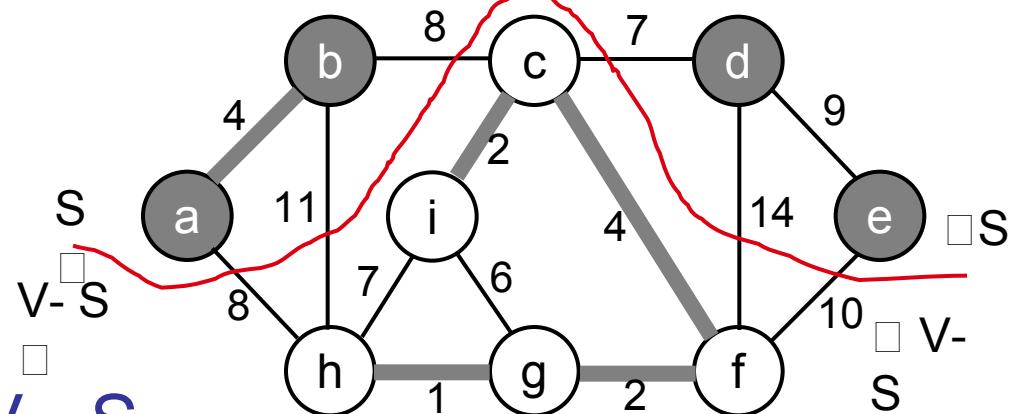
Finding Safe Edges

- Let's look at edge (h, g)
 - Is it safe for A initially?
- Later on:
 - Let $S \subset V$ be any set of vertices that includes h but not g (so that g is in $V - S$)
 - In any MST, there has to be one edge (at least) that connects S with $V - S$
 - Why not choose the edge with **minimum weight** (h,g) ?



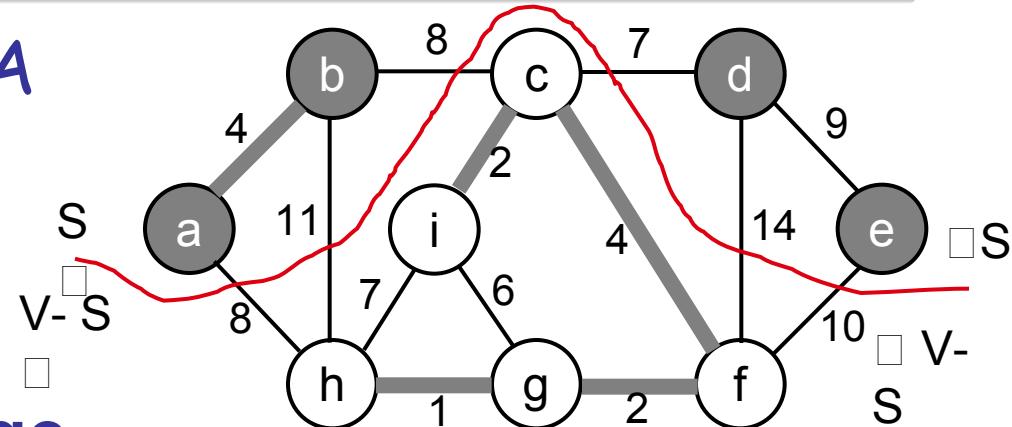
Definitions

- A **cut** $(S, V - S)$ is a partition of vertices into disjoint sets S and $V - S$
- An edge **crosses** the cut $(S, V - S)$ if one endpoint is in S and the other in $V - S$



Definitions (cont'd)

- A cut **respects** a set A of edges \Leftrightarrow no edge in A crosses the cut
- An edge is a **light edge** crossing a cut \Leftrightarrow its weight is minimum over all edges crossing the cut
 - Note that for a given cut, there can be > 1 light edges crossing it

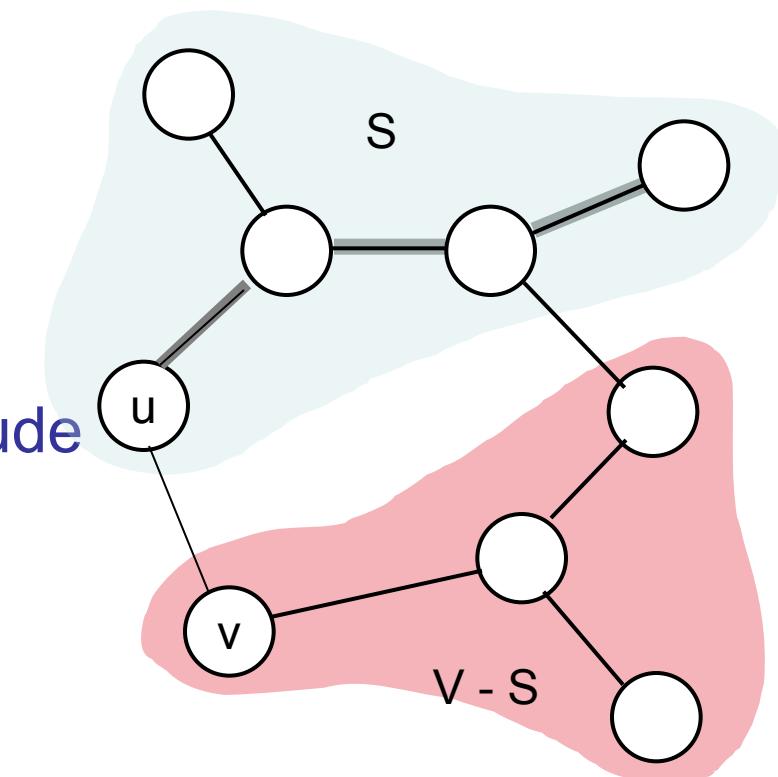


Theorem

- Let A be a subset of some MST (i.e., T), $(S, V - S)$ be a **cut** that respects A , and (u, v) be a **light edge** crossing $(S, V - S)$. Then (u, v) is safe for A .

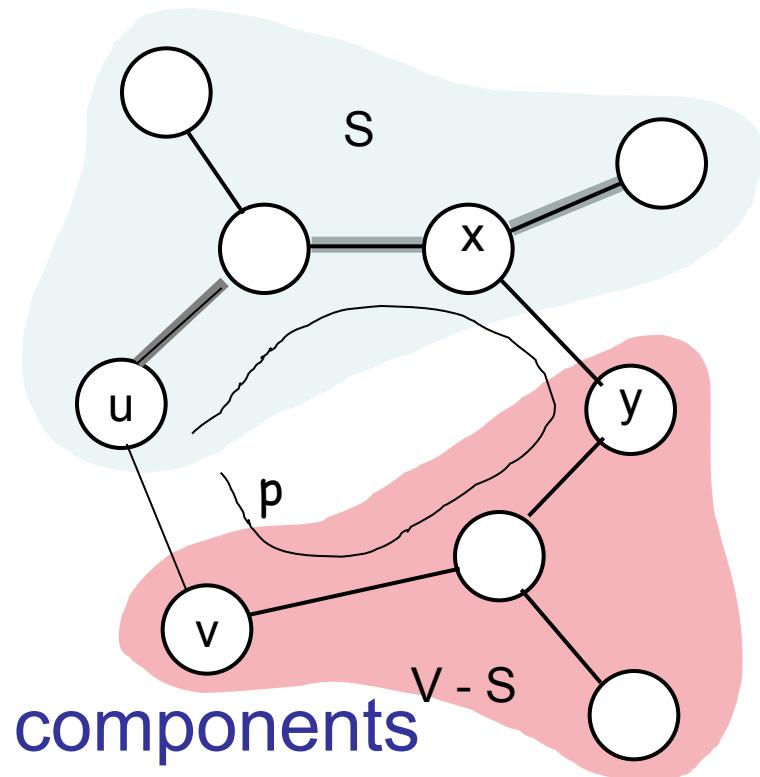
Proof:

- Let T be an MST that includes A
 - edges in A are shaded
- Case1: If T includes (u, v) , then it would be safe for A
- Case2: Suppose T does not include the edge (u, v)
- Idea:** construct another MST T' that includes $A \cup \{(u, v)\}$



Theorem - Proof

- T contains a unique path p between u and v
- Path p must cross the cut $(S, V - S)$ at least once: let (x, y) be that edge
- Let's remove $(x, y) \Rightarrow$ breaks T into two components.
- Adding (u, v) reconnects the components
$$T' = T - \{(x, y)\} \cup \{(u, v)\}$$

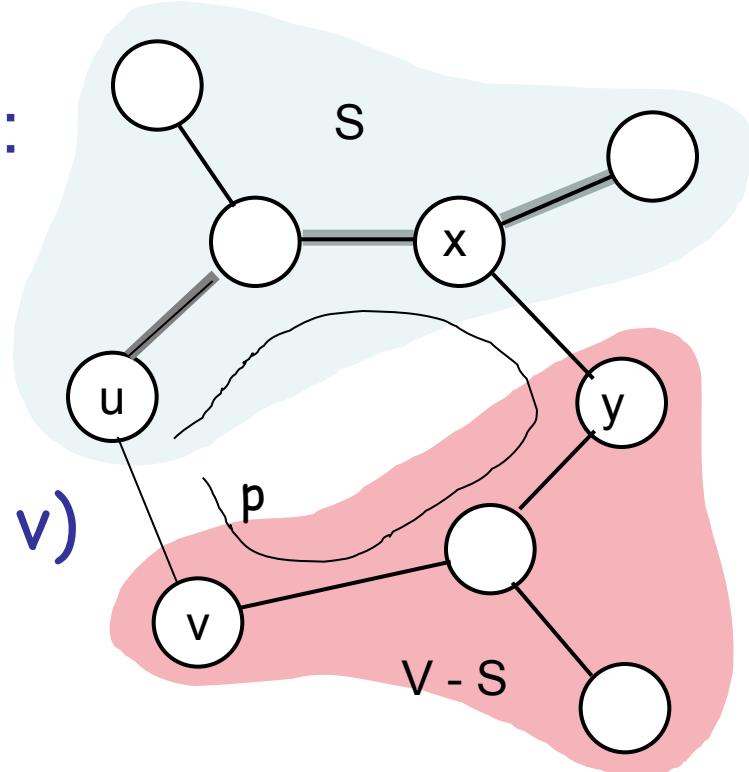


Theorem – Proof (cont.)

$$T' = T - \{(x, y)\} \cup \{(u, v)\}$$

Have to show that T' is an MST:

- (u, v) is a light edge
 $\Rightarrow w(u, v) \leq w(x, y)$
- $w(T') = w(T) - w(x, y) + w(u, v)$
 $\leq w(T)$
- Since T is a spanning tree
 $w(T) \leq w(T') \Rightarrow T'$ must be an MST as well



Theorem – Proof (cont.)

Need to show that (u, v) is safe for A :

i.e., (u, v) can be a part of an MST

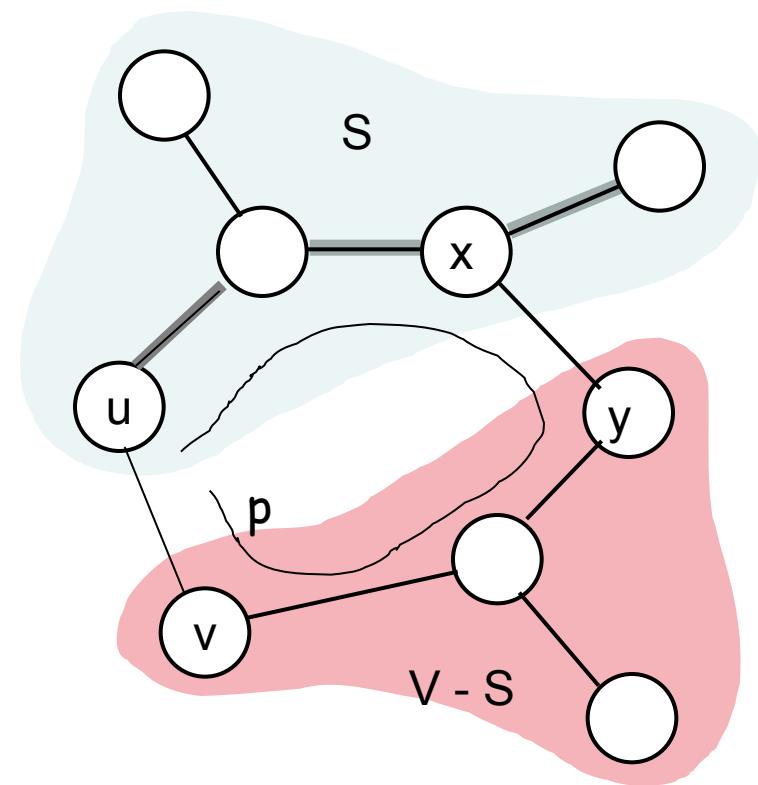
- $A \subseteq T$ and $(x, y) \notin T \Rightarrow$

$$(x, y) \notin A \Rightarrow A \subseteq T'$$

- $A \cup \{(u, v)\} \subseteq T'$

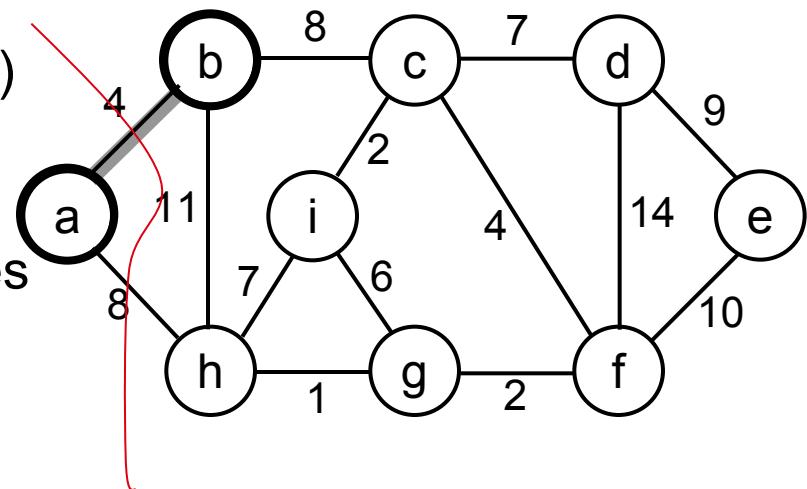
- Since T' is an MST

$\Rightarrow (u, v)$ is safe for A



Prim's Algorithm

- The edges in set A always form a single tree
- Starts from an arbitrary “root”: $V_A = \{a\}$
- At each step:
 - Find a light edge crossing $(V_A, V - V_A)$
 - Add this edge to A
 - Repeat until the tree spans all vertices



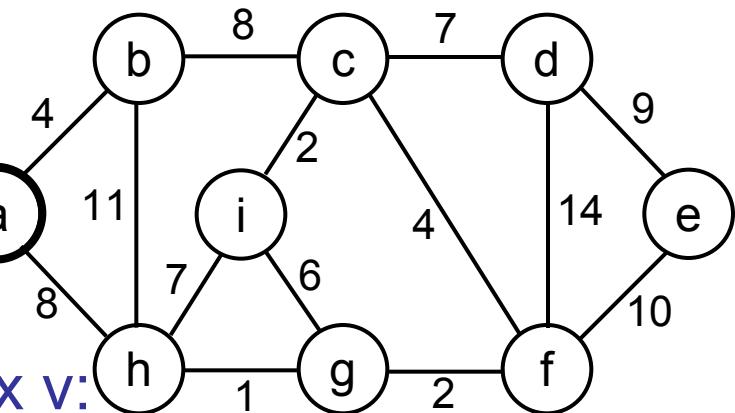
How to Find Light Edges Quickly?

Use a priority queue Q:

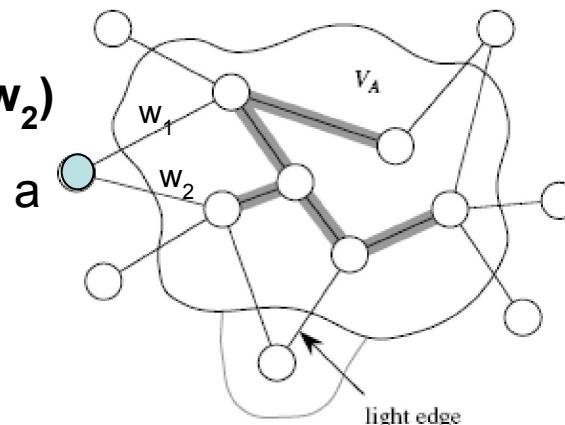
- Contains vertices not yet included in the tree, i.e., $(V - V_A)$
 - $V_A = \{a\}$, $Q = \{b, c, d, e, f, g, h, i\}$

- We associate a key with each vertex v :
 $\text{key}[v] = \text{minimum weight of any edge } (u, v)$

connecting v to V_A

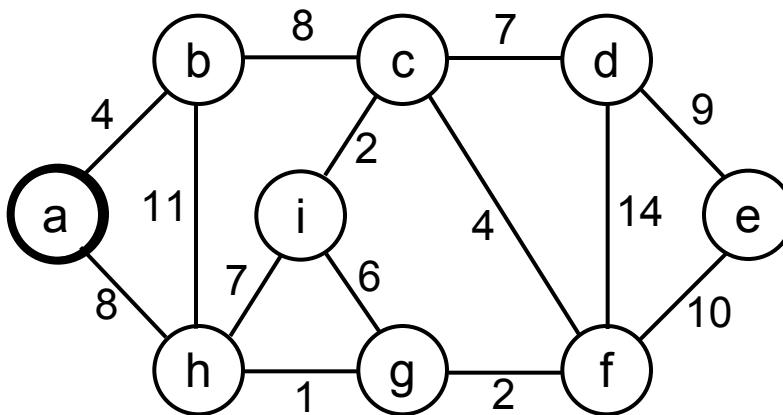


$$\text{Key}[a] = \min(w_1, w_2)$$

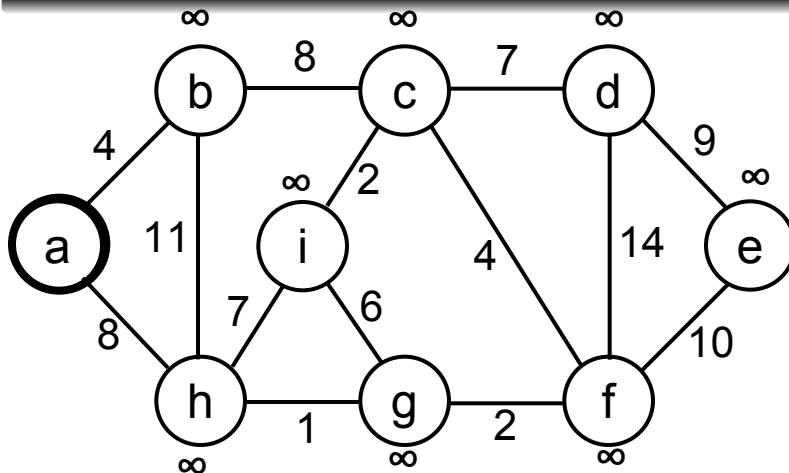


How to Find Light Edges Quickly? (cont.)

- After adding a new node to V_A we update the weights of all the nodes adjacent to it
e.g., after adding a to the tree, $k[b]=4$ and $k[h]=8$
- Key of v is ∞ if v is not adjacent to any vertices in V_A



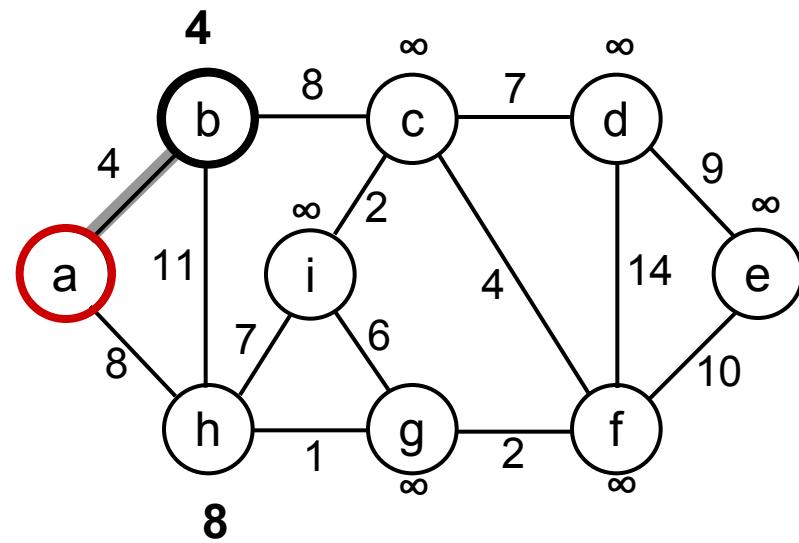
Example



0 $\infty \infty \infty \infty \infty \infty \infty \infty$

$Q = \{a, b, c, d, e, f, g, h, i\}$
 $V_A = \emptyset$

Extract-MIN(Q) $\Rightarrow a$

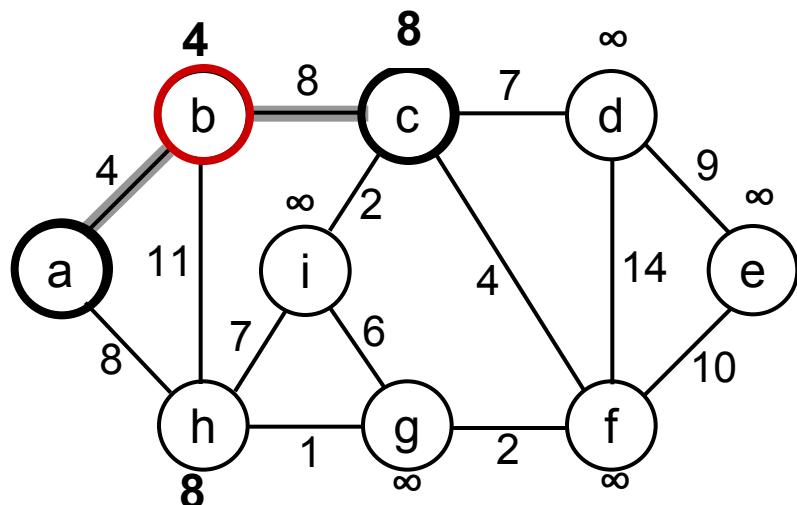


key [b] = 4 π [b] = a
key [h] = 8 π [h] = a

4 $\infty \infty \infty \infty \infty$ 8 ∞

$Q = \{b, c, d, e, f, g, h, i\}$ $V_A = \{a\}$
Extract-MIN(Q) $\Rightarrow b$

Example



$\text{key [c]} = 8 \quad \pi [c] = b$
 $\text{key [h]} = 8 \quad \pi [h] = a - \text{unchanged}$

8 ∞ ∞ ∞ ∞ 8 ∞

$Q = \{c, d, e, f, g, h, i\} \quad V_A = \{a, b\}$
 $\text{Extract-MIN}(Q) \Rightarrow c$

$\text{key [d]} = 7 \quad \pi [d] = c$

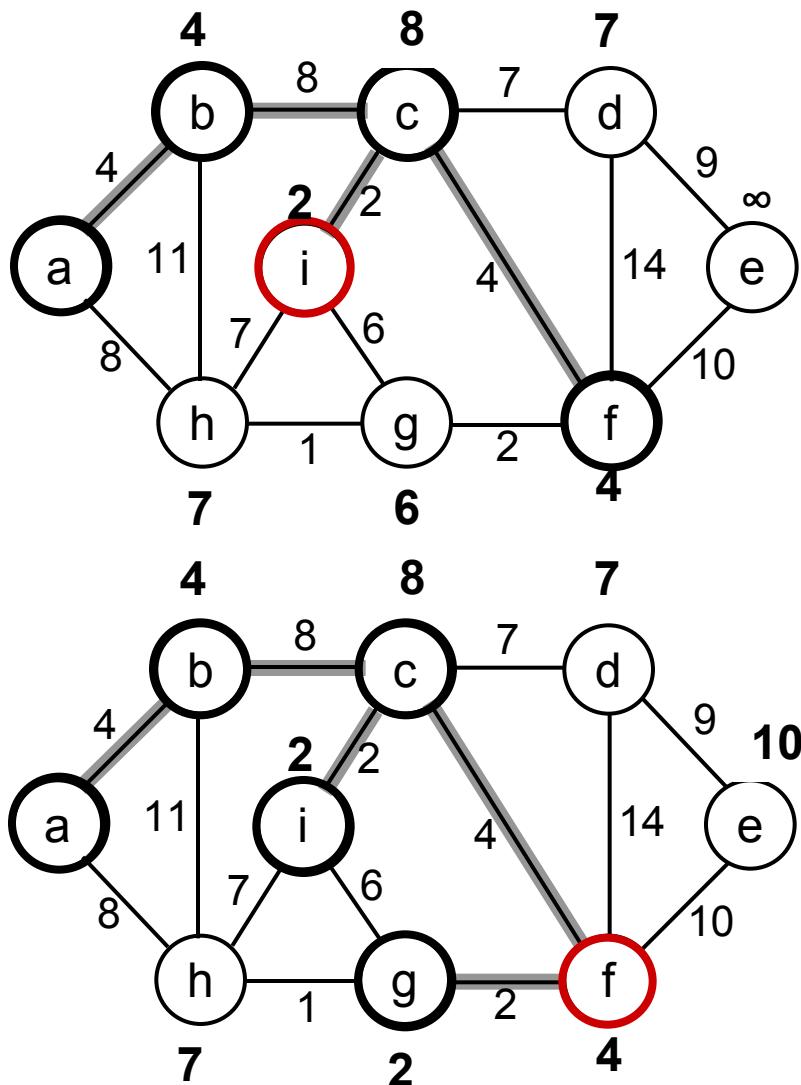
$\text{key [f]} = 4 \quad \pi [f] = c$

$\text{key [i]} = 2 \quad \pi [i] = c$

7 ∞ 4 ∞ 8 2

$Q = \{d, e, f, g, h, i\} \quad V_A = \{a, b, c\}$
 $\text{Extract-MIN}(Q) \Rightarrow i$

Example



key [h] = 7 π [h] = i

key [g] = 6 π [g] = i

7 ∞ 4 6 8

$Q = \{d, e, f, g, h\}$ $V_A = \{a, b, c, i\}$

Extract-MIN(Q) \Rightarrow f

key [g] = 2 π [g] = f

key [d] = 7 π [d] = c unchanged

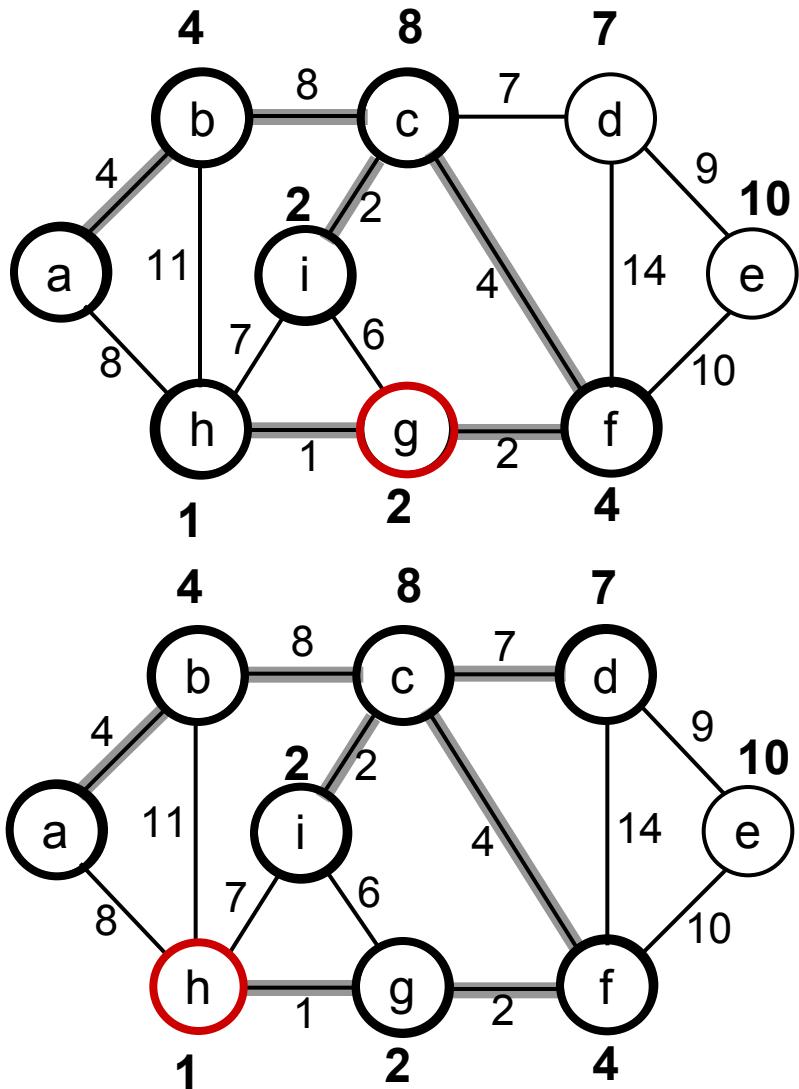
key [e] = 10 π [e] = f

7 10 2 8

$Q = \{d, e, g, h\}$ $V_A = \{a, b, c, i, f\}$

Extract-MIN(Q) \Rightarrow g

Example



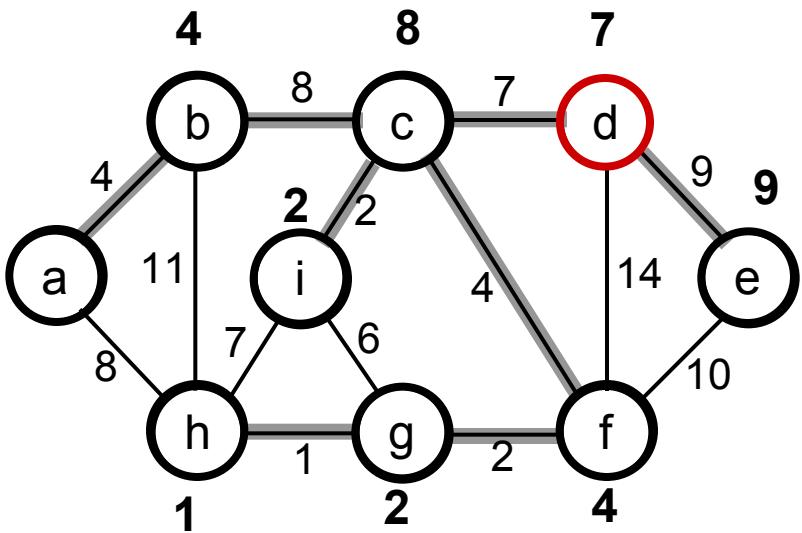
key [h] = 1 π [h] = g
7 10 1

$Q = \{d, e, h\}$ $V_A = \{a, b, c, i, f, g\}$
Extract-MIN(Q) $\Rightarrow h$

7 10

$Q = \{d, e\}$ $V_A = \{a, b, c, i, f, g, h\}$
Extract-MIN(Q) $\Rightarrow d$

Example



key [e] = 9 π [e] = f
9

$Q = \{e\}$ $V_A = \{a, b, c, i, f, g, h, d\}$

Extract-MIN(Q) $\Rightarrow e$

$Q = \emptyset$ $V_A = \{a, b, c, i, f, g, h, d, e\}$

PRIM(V, E, w, r)

```

1.    $Q \leftarrow \emptyset$ 
2.   for each  $u \in V$ 
3.       do  $\text{key}[u] \leftarrow \infty$ 
4.        $\pi[u] \leftarrow \text{NIL}$ 
5.        $\text{INSERT}(Q, u)$ 
6.    $\text{DECREASE-KEY}(Q, r, 0)$             $\triangleright \text{key}[r] \leftarrow 0$             $\leftarrow O(\lg V)$ 
7.   while  $Q \neq \emptyset$             $\leftarrow$  Executed  $|V|$  times
8.       do  $u \leftarrow \text{EXTRACT-MIN}(Q)$             $\leftarrow$  Takes  $O(\lg V)$             $\left[ \begin{array}{l} \text{Min-heap} \\ \text{operations:} \\ O(V\lg V) \end{array} \right]$ 
9.           for each  $v \in \text{Adj}[u]$             $\leftarrow$  Executed  $O(E)$  times total
10.              do if  $v \in Q$  and  $w(u, v) < \text{key}[v]$             $\leftarrow$  Constant
11.                  then  $\pi[v] \leftarrow u$ 
12.                   $\text{DECREASE-KEY}(Q, v, w(u, v))$             $\leftarrow$  Takes  $O(\lg V)$ 

```

Total time: $O(V\lg V + E\lg V) = O(E\lg V)$

$O(V)$ if Q is implemented as a min-heap

Using Fibonacci Heaps

- Depending on the heap implementation, running time could be improved!

	EXTRACT-MIN	DECREASE-KEY	Total
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$	$O(1)$	$O(V \lg V + E)$

Prim's Algorithm

- Prim's algorithm is a “**greedy**” algorithm
 - Greedy algorithms find solutions based on a sequence of choices which are “**locally**” optimal at each step.
- Nevertheless, Prim's greedy strategy produces a globally optimum solution!
 - See proof for generic approach (i.e., slides 12-15)

Problem 4

- **(Exercise 23.2-2, page 573)** Analyze Prim's algorithm assuming:

(a) an adjacency-list representation of G

$$O(E \lg V)$$

(b) an adjacency-matrix representation of G

$$O(E \lg V + V^2)$$

PRIM(V, E, w, r)

```

1.    $Q \leftarrow \emptyset$ 
2.   for each  $u \in V$ 
3.       do  $\text{key}[u] \leftarrow \infty$ 
4.        $\pi[u] \leftarrow \text{NIL}$ 
5.        $\text{INSERT}(Q, u)$ 
6.    $\text{DECREASE-KEY}(Q, r, 0)$             $\triangleright \text{key}[r] \leftarrow 0$             $\leftarrow O(\lg V)$ 
7.   while  $Q \neq \emptyset$             $\leftarrow$  Executed  $|V|$  times
8.       do  $u \leftarrow \text{EXTRACT-MIN}(Q)$             $\leftarrow$  Takes  $O(\lg V)$            Min-heap
9.           for each  $v \in \text{Adj}[u]$             $\leftarrow$  Executed  $O(E)$  times
10.          do if  $v \in Q$  and  $w(u, v) < \text{key}[v]$             $\leftarrow$  Constant
11.             then  $\pi[v] \leftarrow u$ 
12.              $\text{DECREASE-KEY}(Q, v, w(u, v))$             $\leftarrow$  Takes  $O(\lg V)$ 

```

Total time: $O(V\lg V + E\lg V) = O(E\lg V)$

$O(V)$ if Q is implemented as a min-heap

$O(\lg V)$

$O(V\lg V)$

$O(E\lg V)$

PRIM(V, E, w, r)

```

1.    $Q \leftarrow \emptyset$ 
2.   for each  $u \in V$ 
3.       do  $\text{key}[u] \leftarrow \infty$ 
4.        $\pi[u] \leftarrow \text{NIL}$ 
5.        $\text{INSERT}(Q, u)$ 
6.    $\text{DECREASE-KEY}(Q, r, 0)$        $\triangleright \text{key}[r] \leftarrow 0$        $\leftarrow O(\lg V)$ 
7.   while  $Q \neq \emptyset$             $\leftarrow$  Executed  $|V|$  times
8.       do  $u \leftarrow \text{EXTRACT-MIN}(Q)$        $\leftarrow$  Takes  $O(\lg V)$        $\left[ \begin{array}{l} \text{Min-heap} \\ \text{operations:} \\ O(V\lg V) \end{array} \right]$ 
9.           for ( $j=0; j < |V|; j++$ )       $\leftarrow$  Executed  $O(V^2)$  times total
10.          if ( $A[u][j]=1$ )       $\leftarrow$  Constant
11.              if  $v \in Q$  and  $w(u, v) < \text{key}[v]$ 
12.                  then  $\pi[v] \leftarrow u$ 
13.                   $\text{DECREASE-KEY}(Q, v, w(u, v))$        $\leftarrow$  Takes  $O(\lg V)$        $\left[ \begin{array}{l} \\ \\ O(E\lg V) \end{array} \right]$ 

```

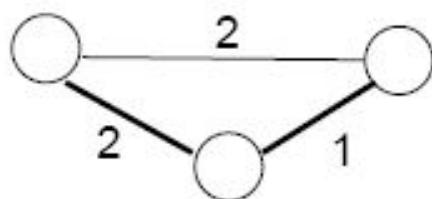
Total time: $O(V\lg V + E\lg V + V^2) = O(E\lg V + V^2)$

Problem 5

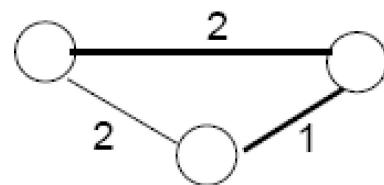
- Find an algorithm for the “maximum” spanning tree. That is, given an undirected weighted graph G , find a spanning tree of G of maximum cost.
Prove the correctness of your algorithm.
 - Consider choosing the “heaviest” edge (i.e., the edge associated with the largest weight) in a cut. The generic proof can be modified easily to show that this approach will work.
 - Alternatively, multiply the weights by -1 and apply either Prim’s or Kruskal’s algorithms without any modification at all!

Problem 6

- **(Exercise 23.1-8, page 567)** Let T be a MST of a graph G , and let L be the sorted list of the edge weights of T . Show that for any other MST T' of G , the list L is also the sorted list of the edge weights of T'



$T, L=\{1,2\}$



$T', L=\{1,2\}$



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.
18CSC201J

DATA STRUCTURES AND ALGORITHMS

Unit- V





SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

MINIMUM SPANNING TREE KRUSKAL'S ALGORITHM

Minimum Spanning Tree

Spanning Tree

Given an undirected and connected graph $G=(V,E)$, a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

Minimum Spanning Tree

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum Spanning Tree

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

1. form a tree that includes every vertex
2. has the minimum sum of weights among all the trees that can be formed from the graph

Algorithm

Input : Graph with V edges

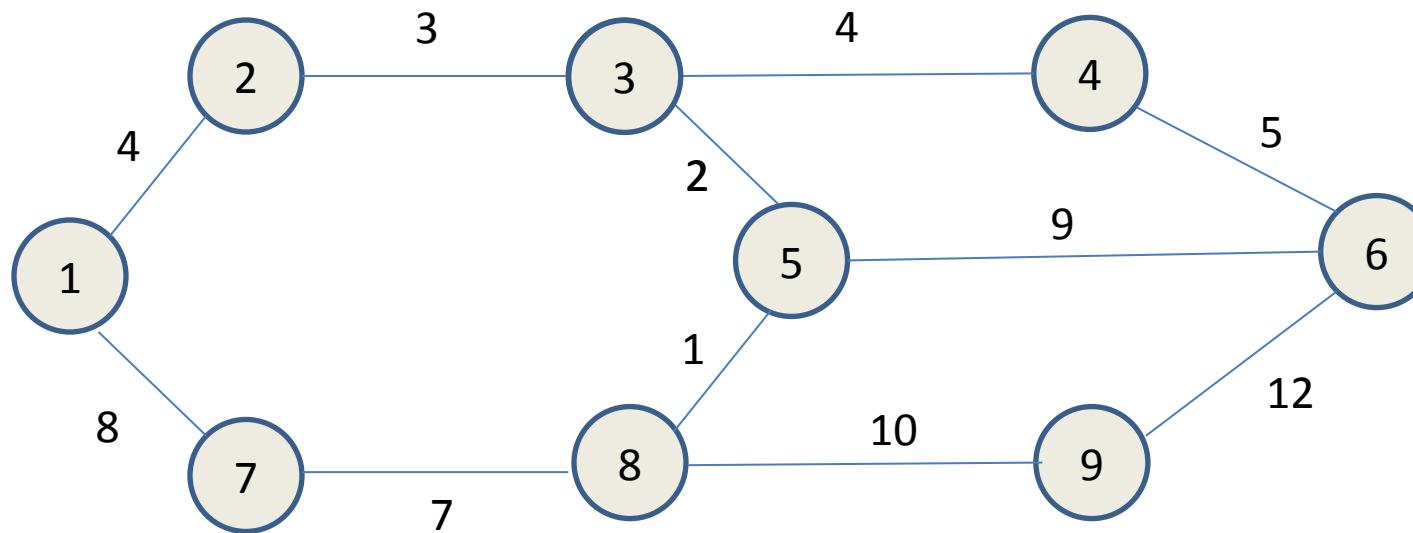
Output : Minimum Spanning tree with $V-1$ edges

Step 1. Sort all the edges in non-decreasing order of their weight.

Step 2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

Step 3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

Example



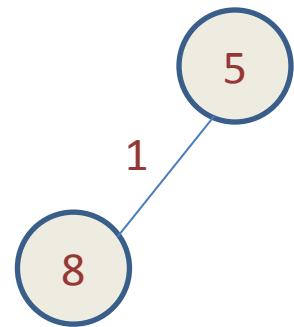
The graph contains 9 vertices and 11 edges. So, the minimum spanning tree formed will be having $(9 - 1) = 8$ edges.

Sort the weights of the graph

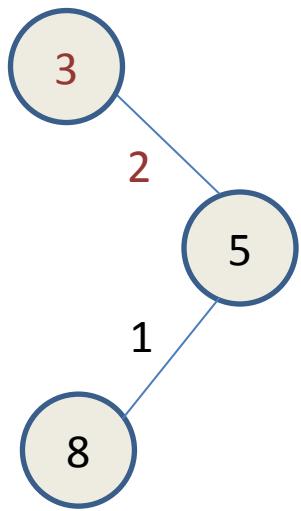
Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

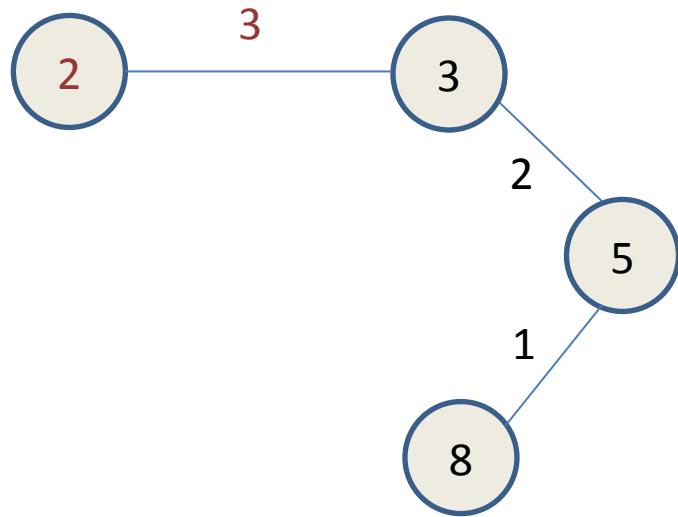
Now pick all edges one by one from sorted list of edges

Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

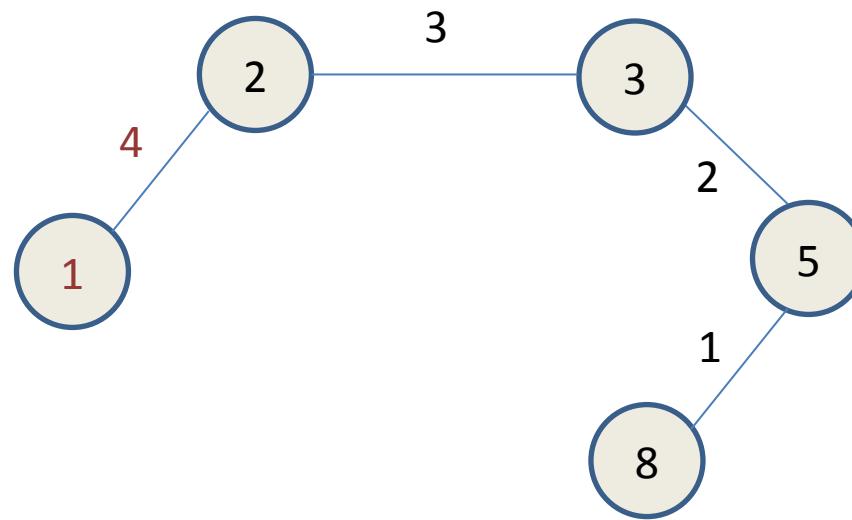


Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

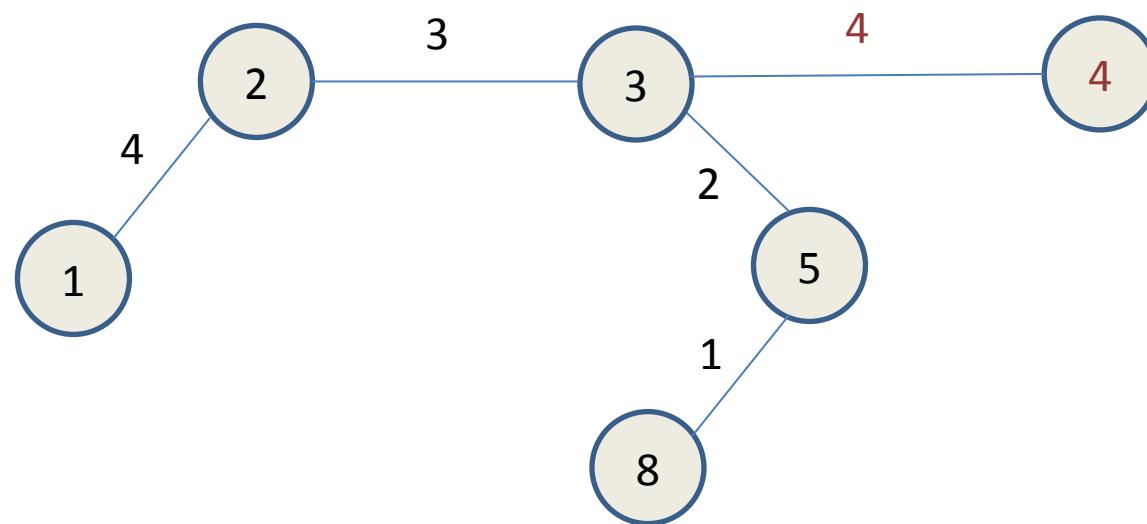




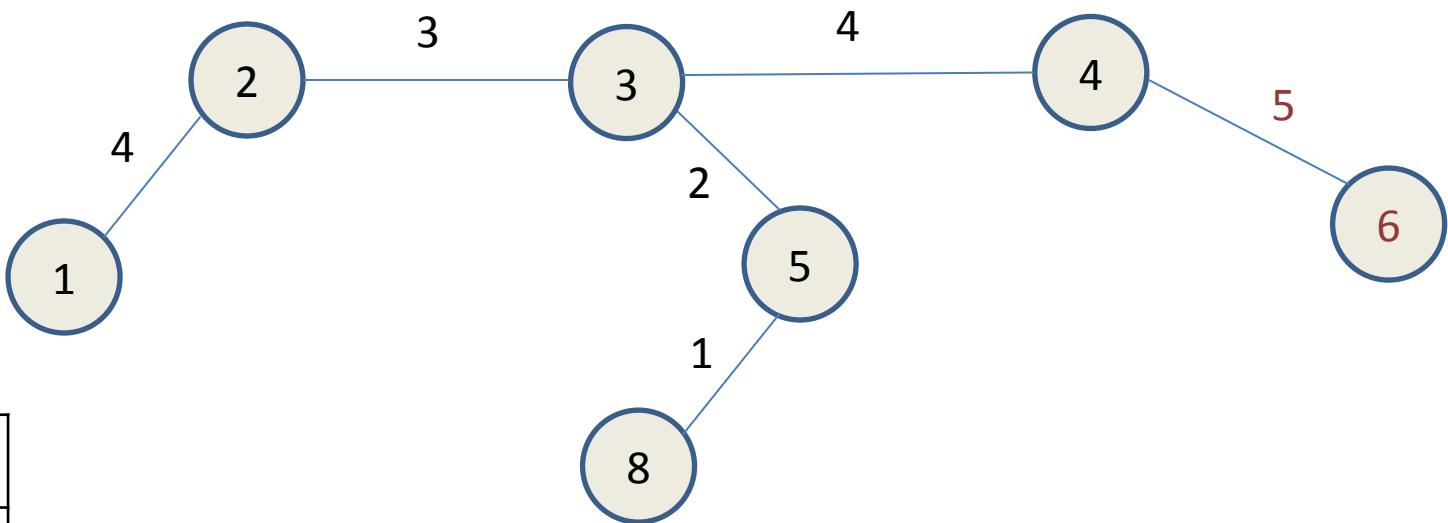
Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6



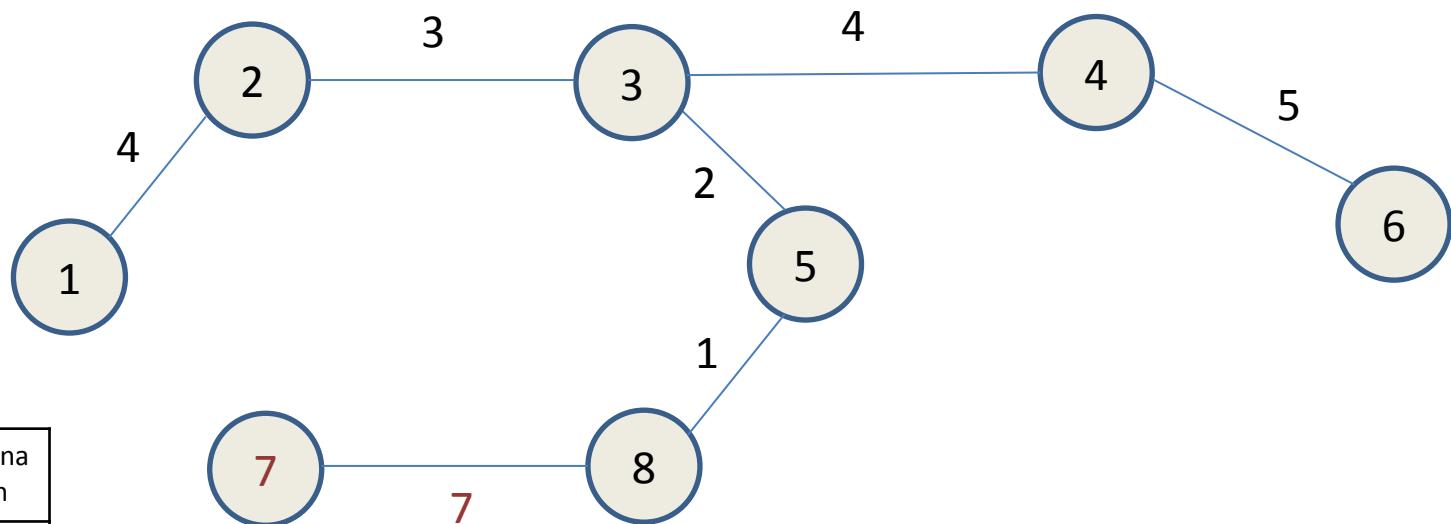
Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6



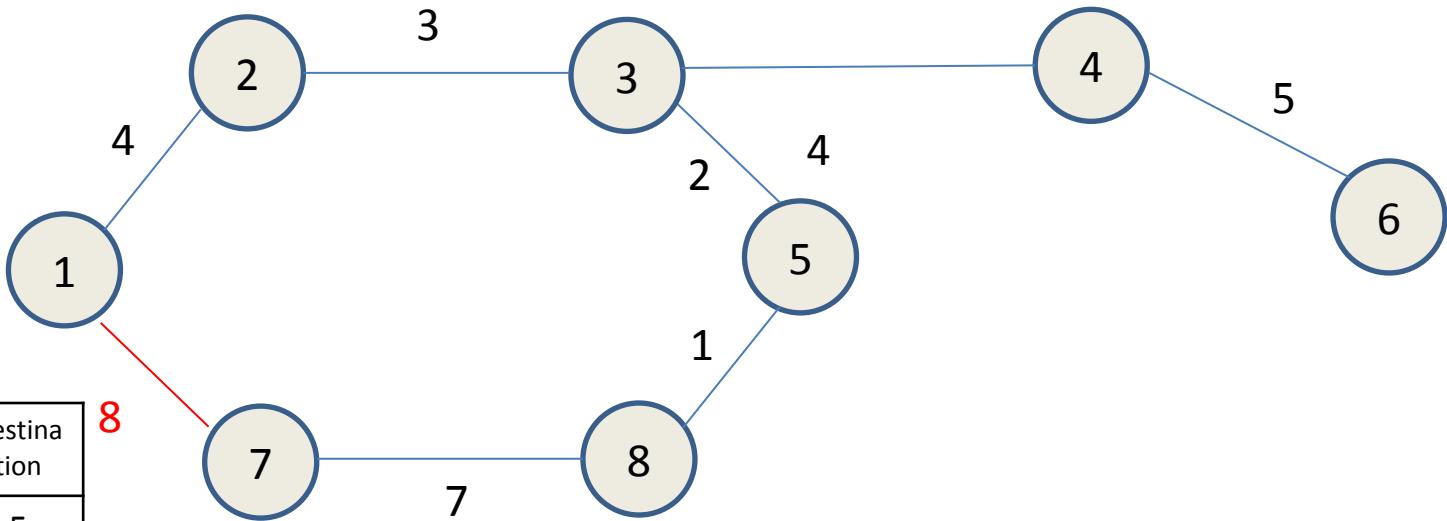
Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6



Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

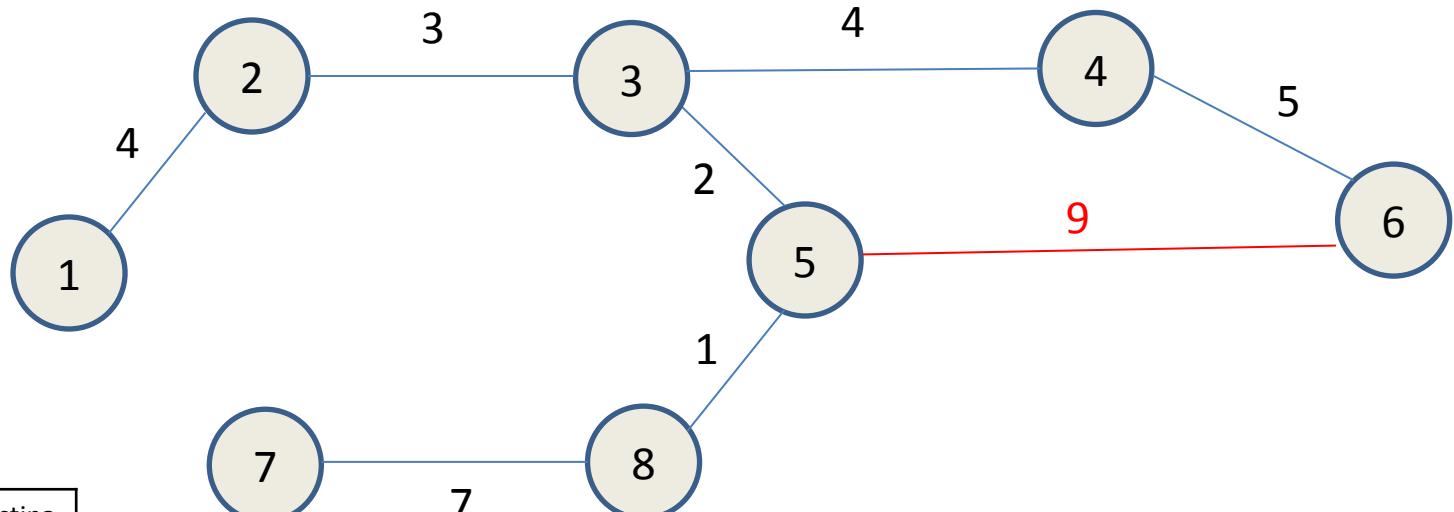


Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6



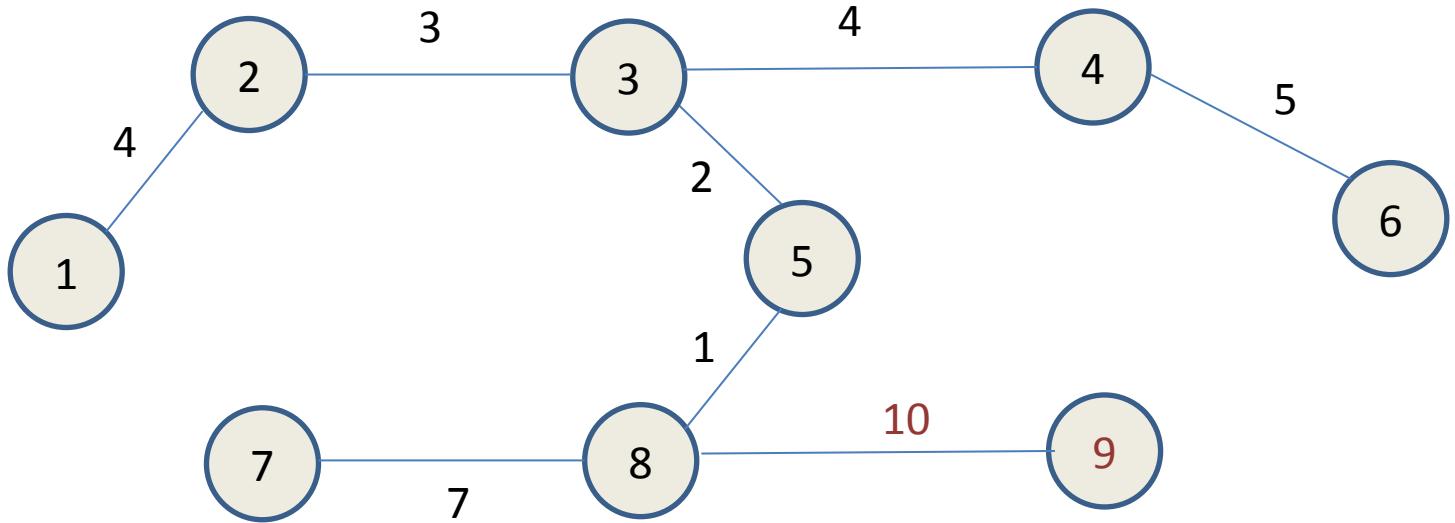
Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Cycle is formed because of weight 8. hence it is discarded.



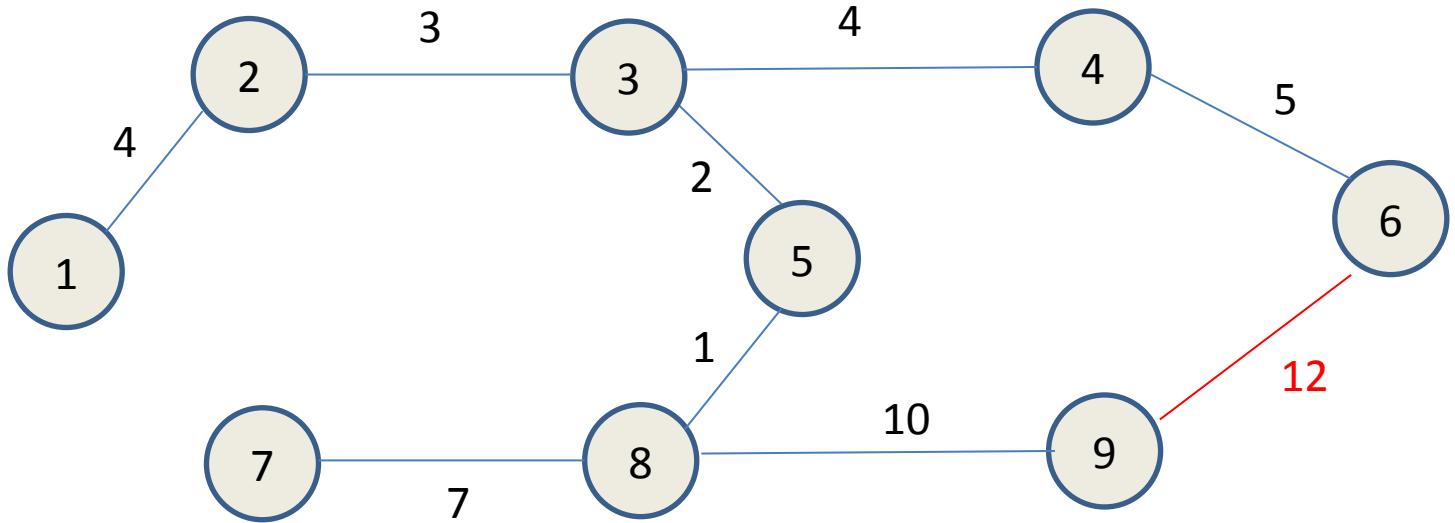
Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Cycle is formed because of weight 9. hence it is discarded.



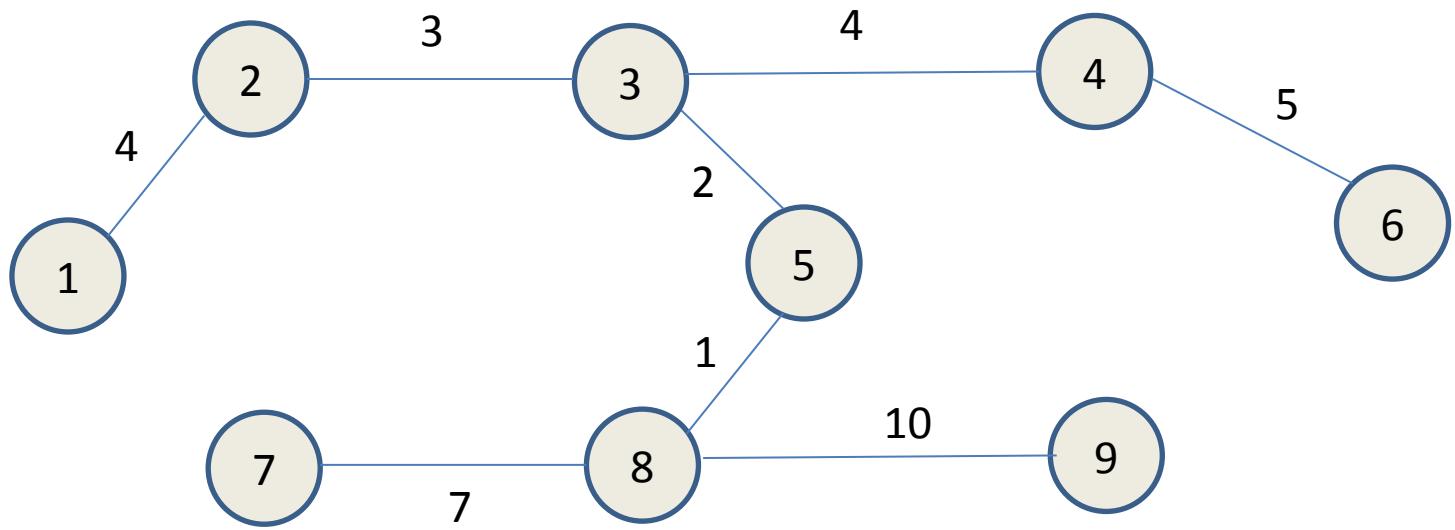
Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Cycle is formed because of weight 12. hence it is discarded.



Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Cycle is formed because of weight 12. hence it is discarded.



Minimum Spanning tree generated by Kruskal's algorithm

References:

1. A.V.Aho, J.E Hopcroft , J.D.Ullman, Data structures and Algorithms, Pearson Education, 2003
2. Reema Thareja, Data Structures Using C, 1st ed., Oxford Higher Education, 2011

Thank
You



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.
18CSC201J

DATA STRUCTURES AND ALGORITHMS

Unit- V



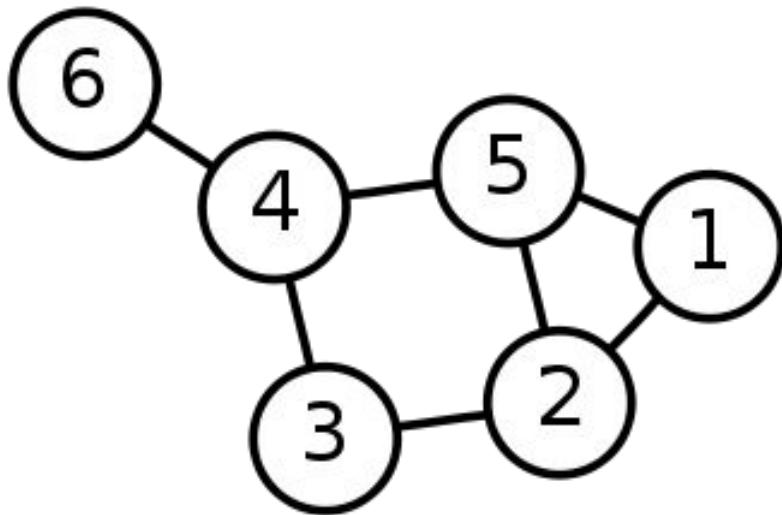


SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Dijkstra's algorithm

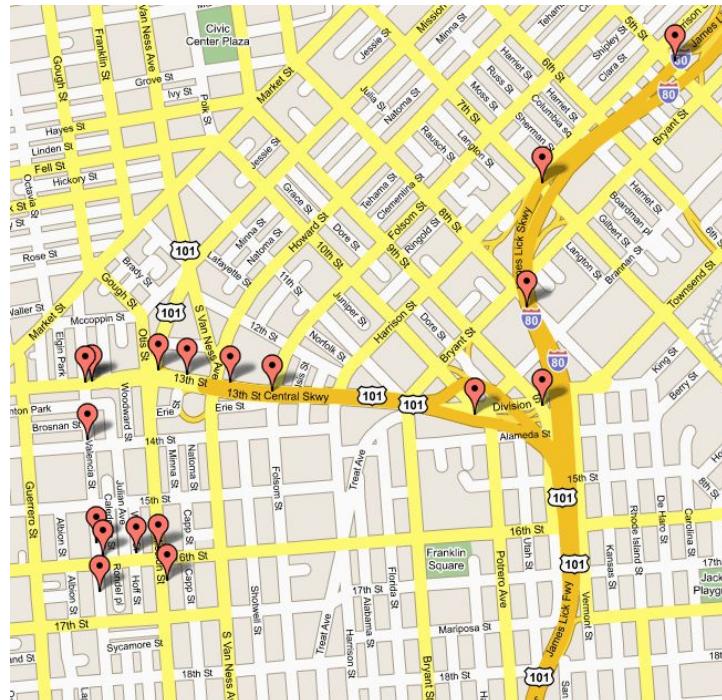
Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex v to all other vertices in the graph.



Applications

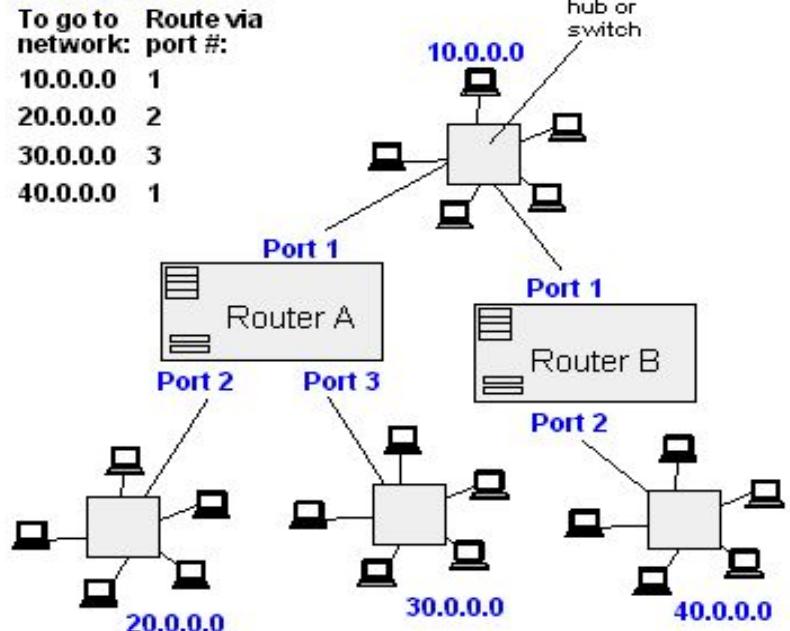
- Maps (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

Router A Routing Table

To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph $G = \{E, V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Approach

1. The algorithm computes for each vertex u the **distance** to u from the start vertex v , that is, the weight of a shortest path between v and u .
2. The algorithm keeps track of the set of vertices for which the distance has been computed, called the **cloud** C
3. Every vertex has a label D associated with it. For any vertex u , $D[u]$ stores an approximation of the distance between v and u . The algorithm will update a $D[u]$ value when it finds a shorter path from v to u .
4. When a vertex u is added to the cloud, its label $D[u]$ is equal to the actual (final) distance between the starting vertex v and vertex u .

Dijkstra pseudocode

Dijkstra(v_1, v_2):

// Initialization

for each vertex v :

v 's distance := infinity.

v 's previous := none.

v_1 's distance := 0.

List := {all vertices}.

while List is not empty:

v := remove List vertex with minimum distance.

mark v as known.

for each unknown neighbor n of v :

dist := v 's distance + edge (v, n) 's weight.

if dist is smaller than n 's distance:

n 's distance := dist.

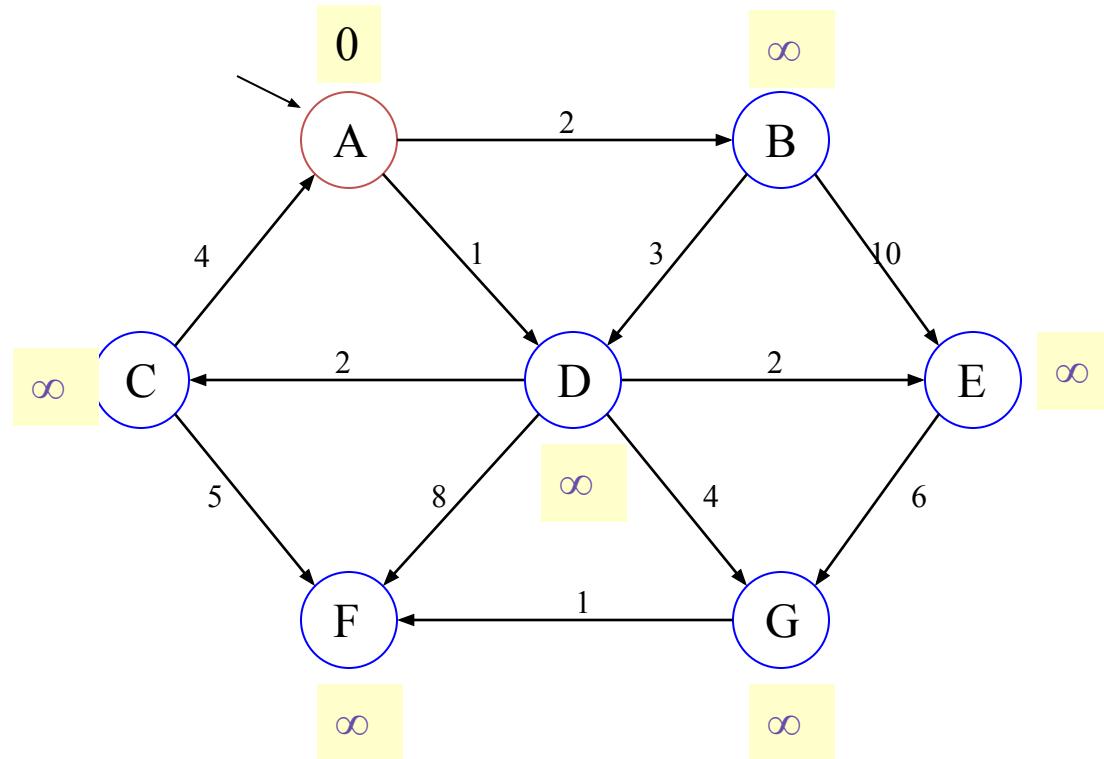
n 's previous := v .

*reconstruct path from v_2 back to v_1 ,
following previous pointers.*

Example: Initialization

Let us consider A as the source node. Distance of all vertices except source is ∞

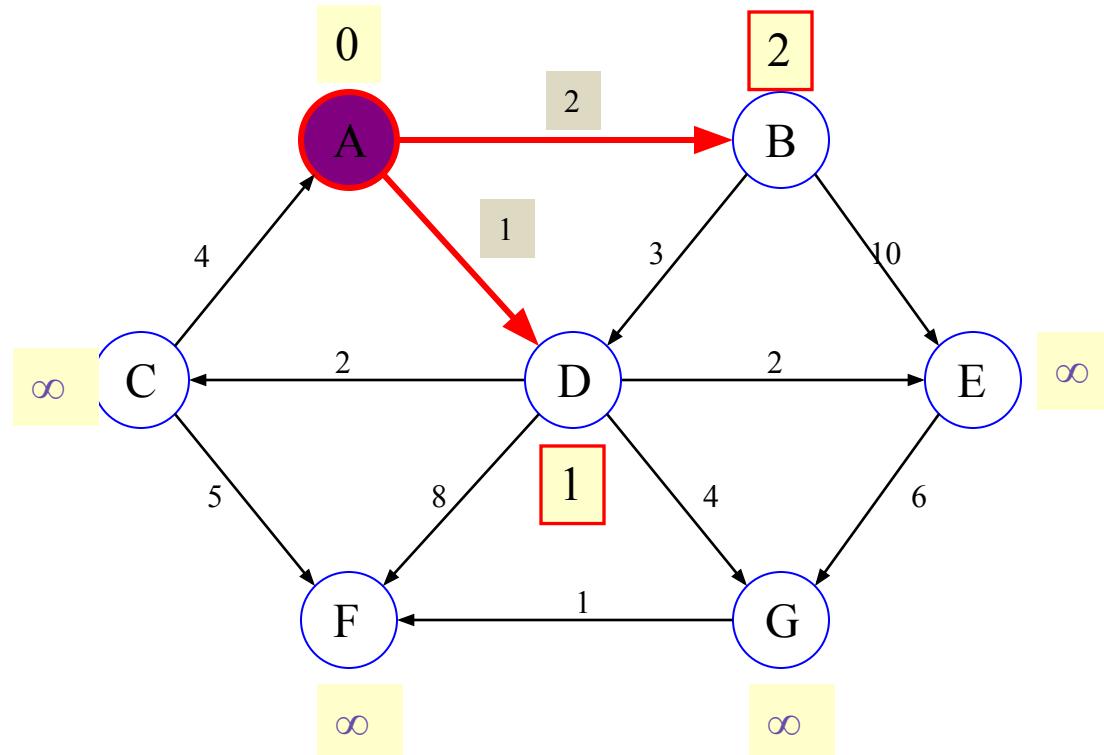
$$\text{Distance}(\text{source}) = 0$$



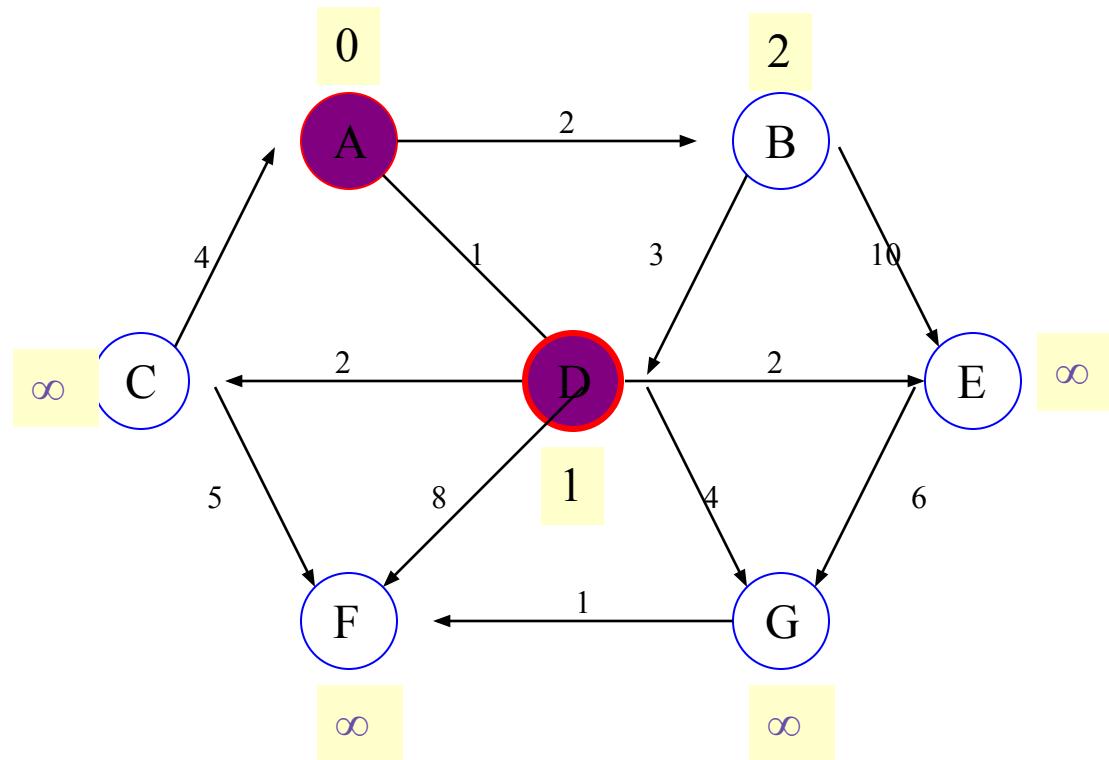
Pick vertex in List with minimum distance.

Update neighbour's distance

Select the source node A and check the neighbour nodes it reaches. Update the weights of B and D as 2 and 1.



Now, select the node with minimum distance. Weights are 2 and 1 hence 1 is selected i.e., D is chosen



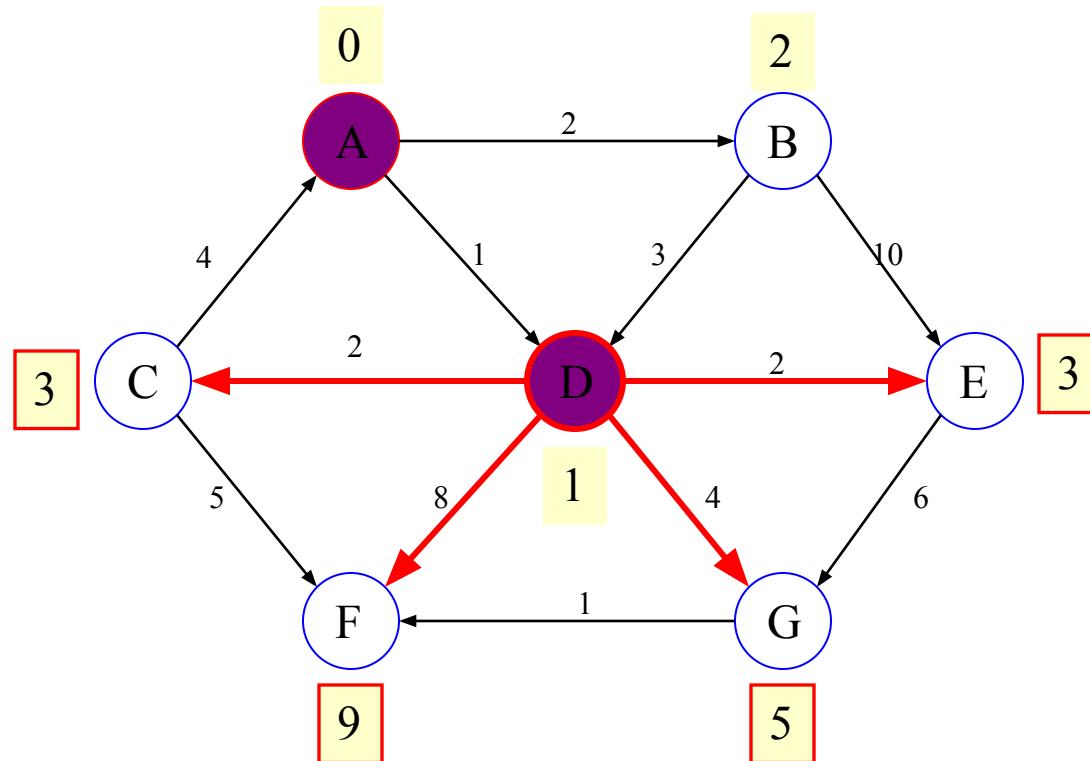
Neighbors of D are selected and their distances are updated.

Distance of C = Distance of D + weight of D to C = $1+2=3$

Distance of E = Distance of D + weight of D to E = $1+2=3$

Distance of F = Distance of D + weight of D to F = $1+8=9$

Distance of G = Distance of D + weight of D to G = $1+4=5$



Pick vertex in List with minimum distance and update neighbors.

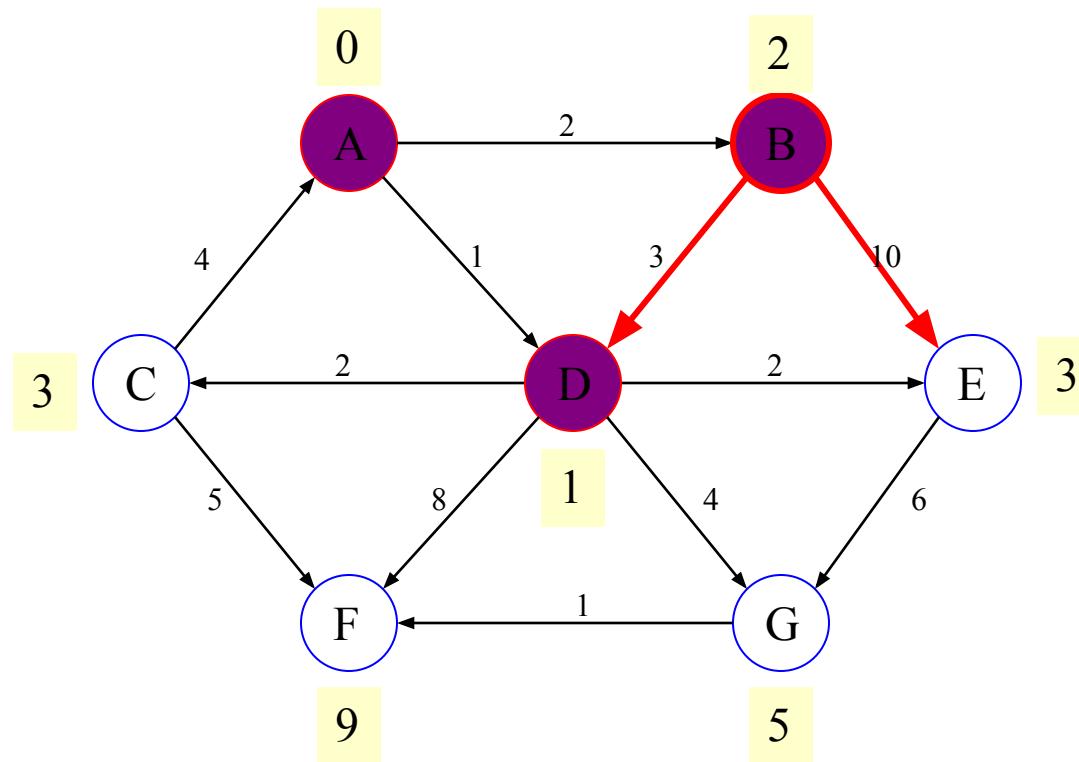
B has the minimum distance. Neighbours of B are D and E.

Distance of D is not updated since D is already visited.

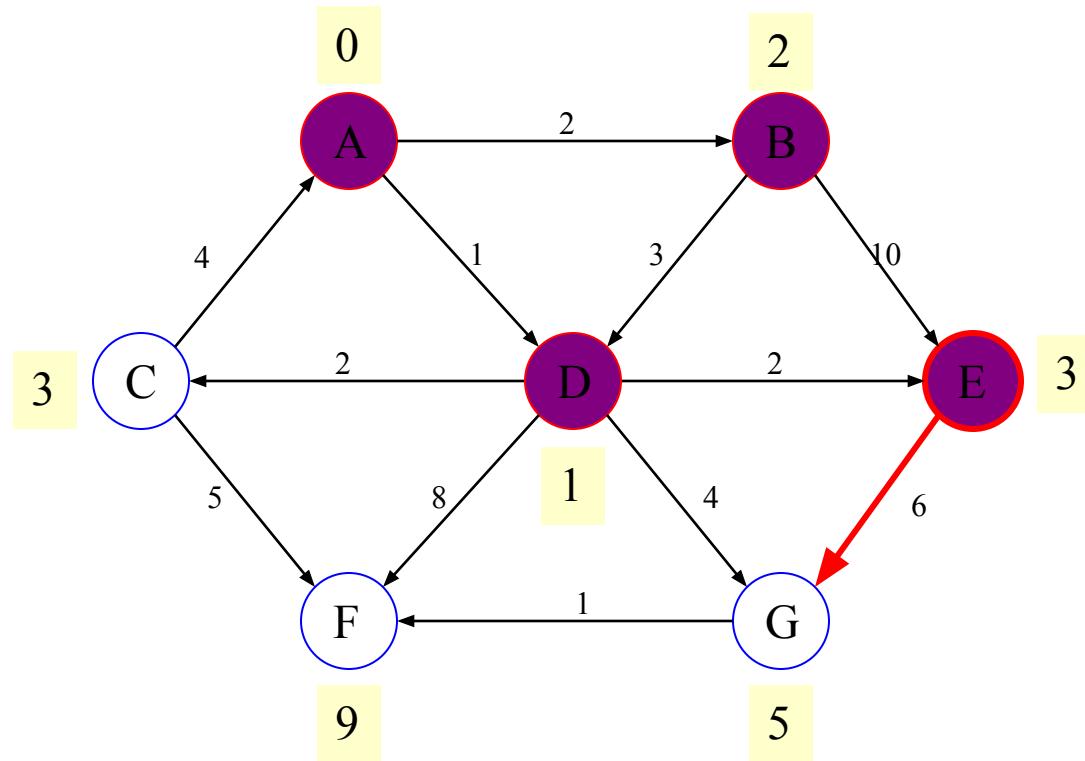
Previous distance of E = 3

New distance of E = $2+10=12$

Distance of E not updated since it is larger than previously computed.



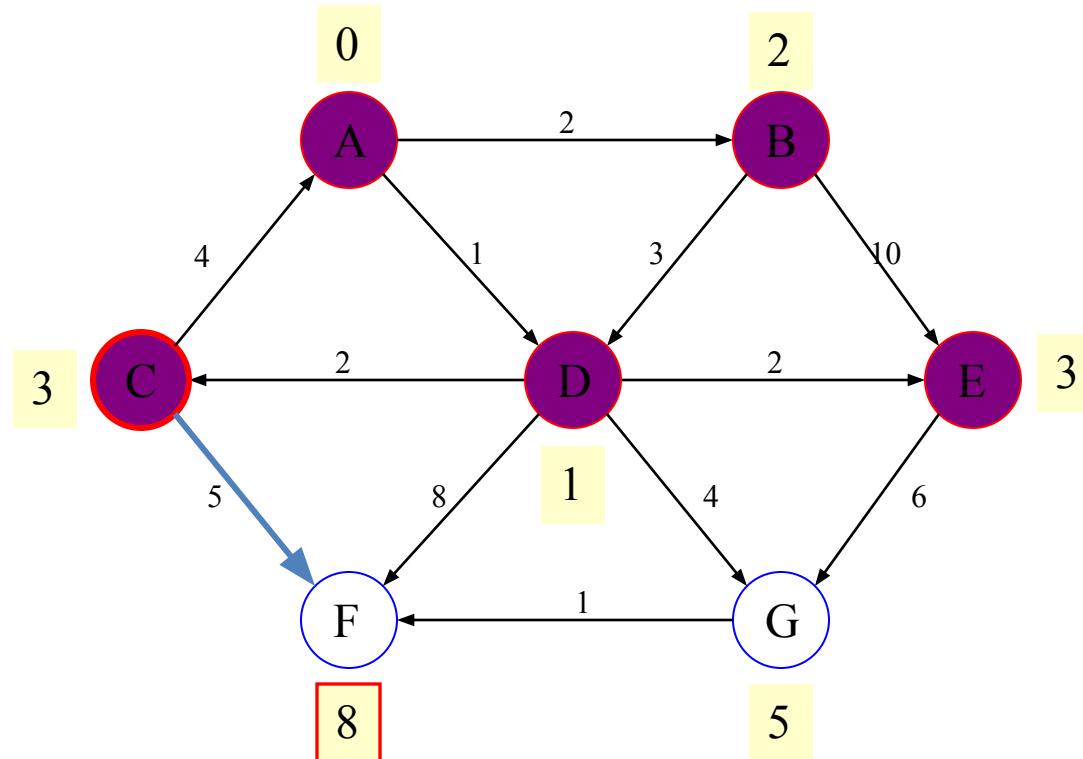
Pick vertex List with minimum distance and update neighbors. E is selected.
Distance of G not updated since it is larger than previously computed.



Pick vertex List with minimum distance and update neighbors.

C is selected. Neighbour is F.

Distance of F = Distance of C + weight of C to F = $3 + 5 = 8$



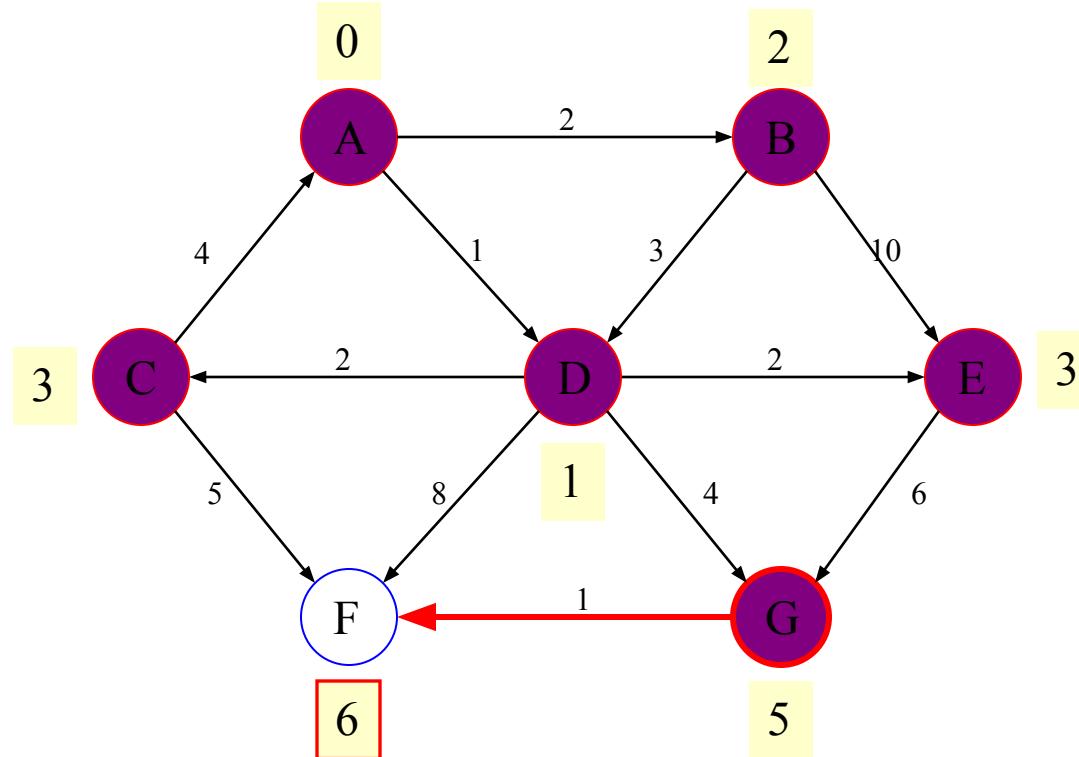
Pick vertex List with minimum distance and update neighbors

G is selected. F is the neighbour.

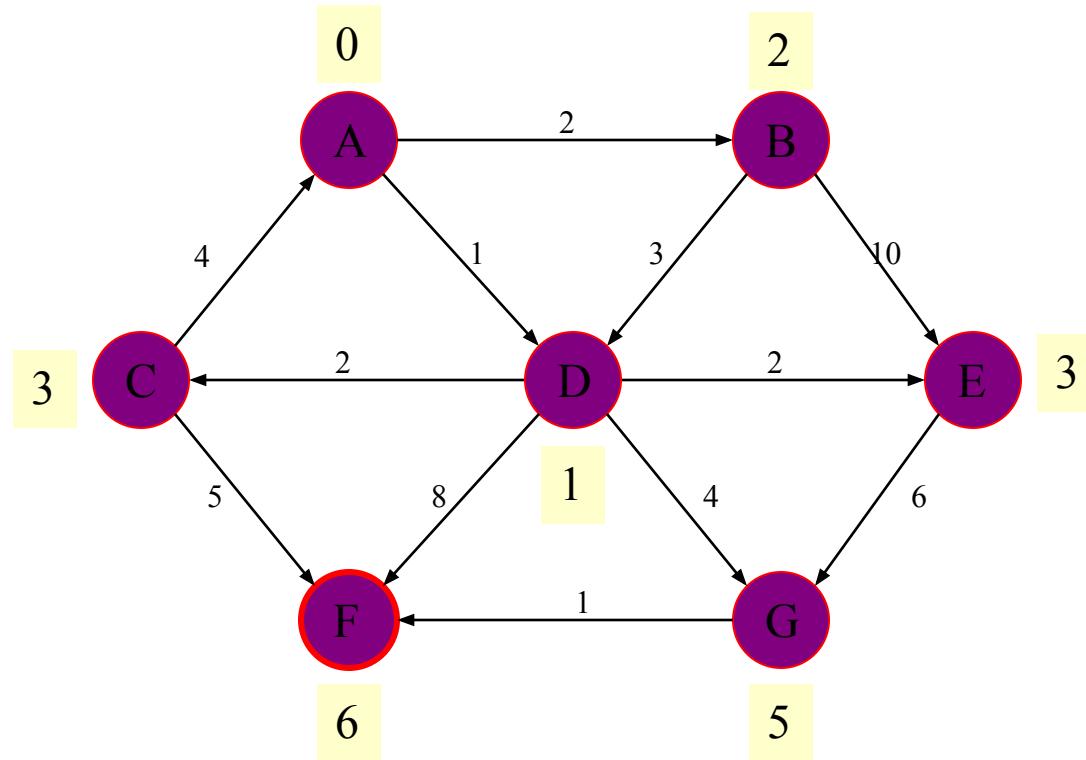
Previous distance of F is 8.

Distance of F = Distance of G+weight from G to F = $5+1$

Update the distance of F as 6.



F is selected and there are no neighbors to F. This is resultant the shortest path graph



References:

1. A.V.Aho, J.E Hopcroft , J.D.Ullman, Data structures and Algorithms, Pearson Education, 2003
2. Reema Thareja, Data Structures Using C, 1st ed., Oxford Higher Education, 2011

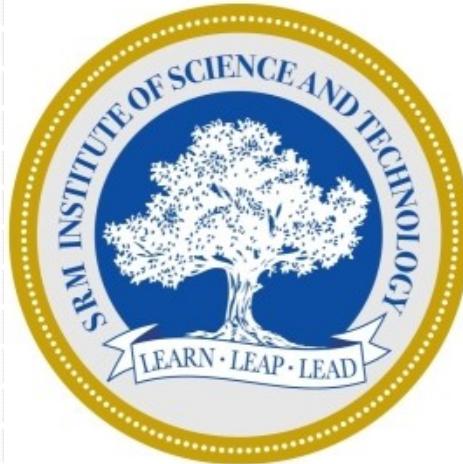
Thank
You



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.
18CSC201J

DATA STRUCTURES AND ALGORITHMS

Unit- V





SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Hashing and Collision

Introduction

In hashing, large keys are converted into small keys by using hash functions.

- The values are then stored in a data structure called hash table.
- The idea of hashing is to distribute entries (key/value pairs) uniformly across an array.
- Each element is assigned a key (converted key).
- By using that key you can access the element in $O(1)$ time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

- An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
- The element is stored in the hash table where it can be quickly retrieved using hashed key. • $\text{hash} = \text{hashfunc(key)}$ $\text{index} = \text{hash \% array_size}$.

INTRODUCTION

In this case we can directly access the record of any employee, once we know his Emp_ID, because array index is same as that of Emp_ID number. But practically, this implementation is hardly feasible.

Let us assume that the same company use a five digit Emp_ID number as the primary key. In this case, key values will range from 00000 to 99999. If we want to use the same technique as above, we will need an array of size 100,000, of which only 100 elements will be used.

HASH FUNCTION

- A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table
- The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.
- A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table.
- The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.
- To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:
 - Easy to compute
 - Uniform distribution
 - Less Collision

HASH FUNCTION

Division Method

- Division method is the most simple method of hashing an integer x . The method divides x by M and then use the remainder thus obtained. In this case, the hash function can be given as

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

- The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M .

HASH FUNCTION

- For example, M is an even number, then $h(x)$ is even if x is even; and $h(x)$ is odd if x is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread hashed values uniformly.
- Generally, it is best to choose M to be a prime number because making M a prime increases the likelihood that the keys are mapped with a uniformity in the output range of values. Then M should also be not too close to exact powers of 2. if we have,
$$h(k) = x \bmod 2^k$$
- then the function will simply extract the lowest k bits of the binary representation of x

HASH FUNCTION

- For example, M is an even number, then $h(x)$ is even if x is even; and $h(x)$ is odd if x is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread hashed values uniformly.
- Generally, it is best to choose M to be a prime number because making M a prime increases the likelihood that the keys are mapped with a uniformity in the output range of values. Then M should also be not too close to exact powers of 2. if we have,
$$h(k) = x \bmod 2^k$$
- then the function will simply extract the lowest k bits of the binary representation of x

HASH FUNCTION

- A potential drawback of the division method is that using this method, consecutive keys map to consecutive hash values. While on one hand this is good as it ensures that consecutive keys do not collide, but on the other hand it also means that consecutive array locations will be occupied. This may lead to degradation in performance.
- Example: Calculate hash values of keys 1234 and 5462.
Setting $m = 97$, hash values can be calculated as
$$h(1234) = 1234 \% 97 = 70$$
$$h(5642) = 5642 \% 97 = 16$$

HASH FUNCTION

Multiplication Method

The steps involved in the multiplication method can be given as below:

Step 1: Choose a constant A such that $0 < A < 1$.

Step 2: Multiply the key k by A

Step 3: Extract the fractional part of kA

Step 4: Multiply the result of Step 3 by m and take the floor.

Hence, the hash function can be given as,

$$h(x) = \lfloor m(kA \bmod 1) \rfloor$$

where, $kA \bmod 1$ gives the fractional part of kA and m is the total number of indices in the hash table

The greatest advantage of the multiplication method is that it works practically with any value of A . Although the algorithm works better with some values than the others but the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is

» **(sqrt5 - 1) / 2 = 0.6180339887**

HASH FUNCTION

Example: Calculate the hash value for keys 1234 and 5642 using the mid square method. The hash table has 100 memory locations.

Note the hash table has 100 memory locations whose indices vary from 0-99. this means, only two digits are needed to map the key to a location in the hash table, so $r = 2$.

When $k = 1234$, $k^2 = 1522756$, $h(k) = 27$

When $k = 5642$, $k^2 = 31832164$, $h(k) = 21$

Observe that 3rd and 4th digits starting from the right are chosen.

HASH FUNCTION

Folding Method

The folding method works in two steps.

Step 1: Divide the key value into a number of parts. That is divide k into parts, k_1, k_2, \dots, k_n , where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is obtain the sum of $k_1 + k_2 + \dots + k_n$. Hash value is produced by ignoring the last carry, if any. Note that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000. Then it means there are 1000 locations in the hash table. To address these 1000 locations, we will need at least three digits, therefore, each part of the key must have three digits except the last part which may have lesser digits.

HASH TABLE

- A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element is stored. By using a good hash function, hashing can work well.
- Under reasonable assumptions, the average time required to search for an element in a hash table is $O(1)$.
- Hash Table is a data structure in which keys are mapped to array positions by a hash function.
- A value stored in the Hash Table can be searched in $O(1)$ time using a hash function to generate an address from the key (by producing the index of the array where the value is stored).

HASH TABLE

- When the set K of keys that are actually used is much smaller than that of U , a hash table consumes much less storage space. The storage requirement for a hash table is just $O(k)$, where k is the number of keys actually used.
- In a hash table, an element with key k is stored at index $h(k)$ not k . This means, a hash function h is used to calculate the index at which the element with key k will be stored. Thus, the process of mapping keys to appropriate locations (or indexes) in a hash table is called ***hashing***.

COLLISION

- If x_1 and x_2 are two different keys, it is possible that $h(x_1) = h(x_2)$. This is called a collision.
- Collision resolution is the most important issue in hash table implementations.
- Choosing a hash function that minimizes the number of collisions and also hashes uniformly is another critical issue.
- Separate chaining (open hashing)
- Linear probing (open addressing or closed hashing)
- Quadratic Probing
- Double hashing

Collision Resolution by Open Addressing

- Once a collision takes place, open addressing computes new position is using a probe sequence and the next record is stored in that position. In this technique of collision resolution, all the values are stored in the hash table. The hash table will contain two types of values- either sentinel value (for example, -1) or a data value. The presence of sentinel value indicates that the location contains no data value at present but can be used to hold a value.
- The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using- linear probing, quadratic probing and double hashing. We will discuss all these techniques in this section.

SEPARATE CHAINING

- Separate chaining is one of the most commonly used collision .
- It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list.
- To store an element in the hash table you must insert it into a specific linked list.
- If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

PROS AND CONS OF HASHING

- Advantage of hashing is that no extra space is required to store the index as in case of other data structures. In addition, a hash table provides fast data access and an added advantage of rapid updates.
- On the other hand, the primary drawback of using hashing technique for inserting and retrieving data values is that it usually lacks locality and sequential retrieval by key. This makes insertion and retrieval of data values even more random.
- All the more choosing an effective hash function is more an art than a science. It is not uncommon to (in open-addressed hash tables) to create a poor hash function.

APPLICATIONS OF HASHING

- Hash tables are widely used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information. A few typical examples where hashing is used are given below.
- Hashing is used for database indexing. Some DBMSs store a separate file known as indexes. When data has to be retrieved from a file, the key information is first found in the appropriate index file which references the exact record location of the data in the database file. This key information in the index file is often stored as a hashed value.
- Hashing is used as symbol tables, for example, in Fortran language to store variable names. Hash tables speeds up execution of the program as the references to variables can be looked up quickly.

APPLICATIONS OF HASHING

- In many database systems, File and Directory hashing is used in high performance file systems. Such systems use two complementary techniques to improve the performance of file access. While one of these techniques is caching which saves information in memory, the other is hashing which makes looking up the file location in memory much quicker than most other methods.
- Hash tables can be used to store massive amount of information for example, to store driver's license records. Given the driver's license number, hash tables help to quickly get information about the driver (i.e. name, address, age)
- Hashing technique is for compiler symbol tables in C++. The compiler uses a symbol table to keep a record of the user-defined symbols in a C++ program. Hashing facilitates the compiler to quickly look up variable names and other attributes associated with symbols .



SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.
18CSC201J

DATA STRUCTURES AND ALGORITHMS

Unit- V





SR
**INSTITUTE OF SCIENCE AND
TECHNOLOGY,**
CHENNAI.

Hashing

Topics covered: quadratic, double, rehashing and extensible hashing

Open Addressing (Probing)

Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets). If the index given by the hash function is occupied, then increment the table position by some number.

There are three schemes commonly used for probing:

Linear Probing: The linear probing algorithm is detailed below:

Index := hash(key)

While Table(Index) Is Full do

index := (index + 1) MOD Table_Size

if (index = hash(key))

return table_full

else

Table(Index) := Entry

Quadratic Probing: increment the position computed by the hash function in

quadratic fashion i.e. increment by 1, 4, 9, 16,

Double Hash: compute the index as a function of two different hash functions.

Quadratic Probing

Quadratic probing eliminates primary clusters.

$c(i)$ is a quadratic function in i of the form $c(i) = a*i^2 + b*i$. Usually $c(i)$ is chosen as:

$$c(i) = i^2 \text{ for } i = 0, 1, \dots, \text{tableSize} - 1$$

Or

$$c(i) = \pm i^2 \text{ for } i = 0, 1, \dots, (\text{tableSize} - 1) / 2$$

The probe sequences are then given by:

$$h_i(\text{key}) = [\text{h(key)} + i^2] \% \text{tableSize} \text{ for } i = 0, 1, \dots, \text{tableSize} - 1$$

Or

$$h_i(\text{key}) = [\text{h(key)} \pm i^2] \% \text{tableSize} \text{ for } i = 0, 1, \dots, (\text{tableSize} - 1) / 2$$

↓ ↓
Probe number key Auxiliary hash function

Properties of quadratic hashing

- Hashtable size should not be an even number; otherwise Property 2 will not be satisfied.
- Ideally, table size should be a prime of the form $4j+3$, where j is an integer. This choice of table size guarantees Property 2.

Example:Quadratic Probing

- ✓ Load the keys 23, 13, 21, 14, 7, 8, and 15, in this order, in a hash table of size 7 using quadratic probing with $c(i) = \pm i^2$ and the hash function:

$$h(\text{key}) = \text{key \% 7}$$

- ✓ The required probe sequences are given by:

$$h_i (\text{key}) = (h(\text{key}) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$$

Computations

$$hi(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$$

$$h_0(23) = (23 \% 7) \% 7 = 2$$

$$h_0(13) = (13 \% 7) \% 7 = 6$$

$$h_0(21) = (21 \% 7) \% 7 = 0$$

$$h_0(14) = (14 \% 7) \% 7 = 0 \text{ collision}$$

$$h_1(14) = (0 + 1_2) \% 7 = 1$$

$$h_0(7) = (7 \% 7) \% 7 = 0 \text{ collision}$$

$$h_1(7) = (0 + 1_2) \% 7 = 1 \text{ collision}$$

$$h_{-1}(7) = (0 - 1_2) \% 7 = -1$$

$$\text{NORMALIZE: } (-1 + 7) \% 7 = 6 \text{ collision}$$

$$h_2(7) = (0 + 2_2) \% 7 = 4$$

$$h_0(8) = (8 \% 7) \% 7 = 1 \text{ collision}$$

$$h_1(8) = (1 + 1_2) \% 7 = 2 \text{ collision}$$

$$h_{-1}(8) = (1 - 1_2) \% 7 = 0 \text{ collision}$$

$$h_2(8) = (1 + 2_2) \% 7 = 5$$

$$h_0(15) = (15 \% 7) \% 7 = 1 \text{ collision}$$

$$h_1(15) = (1 + 1_2) \% 7 = 2 \text{ collision}$$

$$h_{-1}(15) = (1 - 1_2) \% 7 = 0 \text{ collision}$$

$$h_2(15) = (1 + 2_2) \% 7 = 5 \text{ collision}$$

$$h_{-2}(15) = (1 - 2_2) \% 7 = -3$$

$$\text{NORMALIZE: } (-3 + 7) \% 7 = 4 \text{ collision}$$

0	O	21
1	O	14
2	O	23
3	O	15
4	O	7
5	O	8
6	O	13

Quadratic probing is better than linear probing because it eliminates primary clustering.

Double Hashing

Double hashing achieves this by having two hash functions that both depend on the hash key.

$$c(i) = i * h_p(\text{key}) \text{ for } i = 0, 1, \dots, \text{tableSize} - 1$$

where h_p (or h_2) is another hash function.

The probing sequence is:

$$h_i(\text{key}) = [h(\text{key}) + i * h_p(\text{key})] \% \text{tableSize} \text{ for } i = 0, 1, \dots, \text{tableSize} - 1$$

The function $c(i) = i * h_p(r)$ satisfies Property 2 provided $h_p(r)$ and tableSize are relatively prime.

Common definitions for h_p are :

$$h_p(\text{key}) = 1 + \text{key \% (tableSize - 1)}$$

$$h_p(\text{key}) = q - (\text{key \% q}) \text{ where q is a prime less than tableSize}$$

$$h_p(\text{key}) = q * (\text{key \% q}) \text{ where q is a prime less than tableSize}$$

Performance and example of Double hashing

Much better than linear or quadratic probing because it eliminates both primary and secondary clustering. BUT requires a computation of a second hash function h_p .

Example: Load the keys 18, 26, 35, 9, 64, 47, 96, 36, and 70 in this order, in an empty hash table of size 13

- (a) using double hashing with the first hash function: $h(\text{key}) = \text{key} \% 13$ and the second hash function: $h_p(\text{key}) = 1 + \text{key} \% 12$
- (b) using double hashing with the first hash function: $h(\text{key}) = \text{key} \% 13$ and the second hash function: $h_p(\text{key}) = 7 - \text{key} \% 7$

Computation

$$h_0(18) = (18\%13)\%13 = 5$$

$$h_0(26) = (26\%13)\%13 = 0$$

$$h_0(35) = (35\%13)\%13 = 9$$

$$h_0(9) = (9\%13)\%13 = 9 \text{ collision}$$

$$h_p(9) = 1 + 9\%12 = 10$$

$$h_1(9) = (9 + 1*10)\%13 = 6$$

$$h_0(64) = (64\%13)\%13 = 12$$

$$h_0(47) = (47\%13)\%13 = 8$$

$$h_0(96) = (96\%13)\%13 = 5 \text{ collision}$$

$$h_p(96) = 1 + 96\%12 = 1$$

$$h_1(96) = (5 + 1*1)\%13 = 6 \text{ collision}$$

$$h_2(96) = (5 + 2*1)\%13 = 7$$

$$h_0(36) = (36\%13)\%13 = 10$$

$$h_0(70) = (70\%13)\%13 = 5 \text{ collision}$$

$$h_p(70) = 1 + 70\%12 = 11$$

$$h_1(70) = (5 + 1*11)\%13 = 3$$

$$h_i(\text{key}) = [h(\text{key}) + i*h_p(\text{key})]\% 13$$

$$h(\text{key}) = \text{key \% 13}$$

$$h_p(\text{key}) = 1 + \text{key \% 12}$$

0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64

Rehashing

- Rehashing is with respect to closed hashing. When we try to store the record with Key1 at bucket Hash(Key1) position and find that it already holds a record, it is collision situation
- To handle collision, we use strategy to choose a sequence of alternative locations Hash1(Key1), Hash2(Key1), ... within the bucket table so as to place the record with Key1

If 23 is inserted, the table is over 70 percent full.

0	6
1	15
2	23
3	24
4	
5	
6	13



A new table is created

17 is the first prime twice as large as the old one; so

$$H_{\text{new}}(X) = X \bmod 17$$

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Extendible hashing

- Extendible hashing is based on a radix-2 trie. The idea is to hash the key, yielding a long two-bit number. Then, use as many bits as desired to create a radix-2 trie.
- But, instead of building a tree, just collapse the tree, interpreting the 0/1 sequence along each branch as a binary number, used as the index into the bucket array.
- When collision occurs, more bits can be used to divide the buckets into a larger (by powers of 2) address space.

Illustration of Extendible hashing

An empty Extendible Hash Table w/One Bucket and an initial address space of 2 bits.

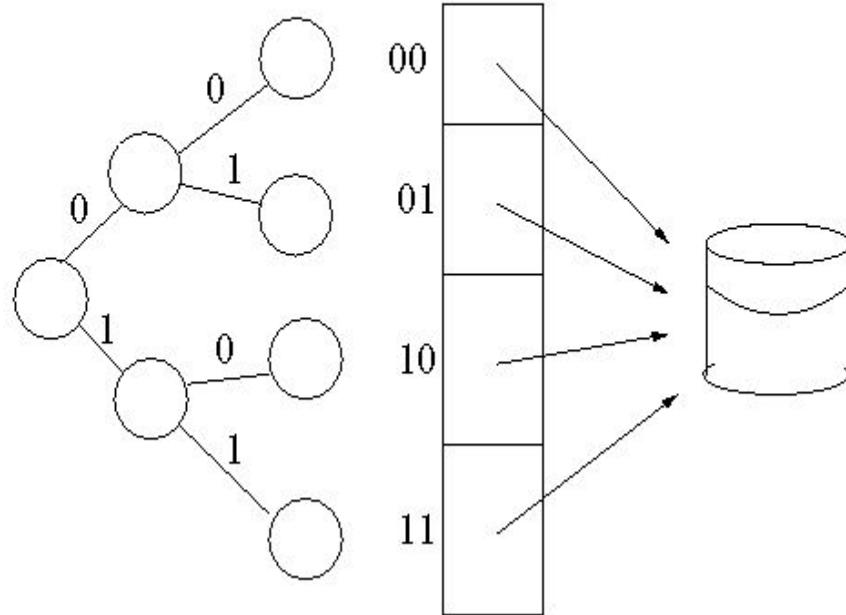


Illustration of Extendible hashing

The address space splits in response to the addition of two keys, one with prefix 00, the other 01

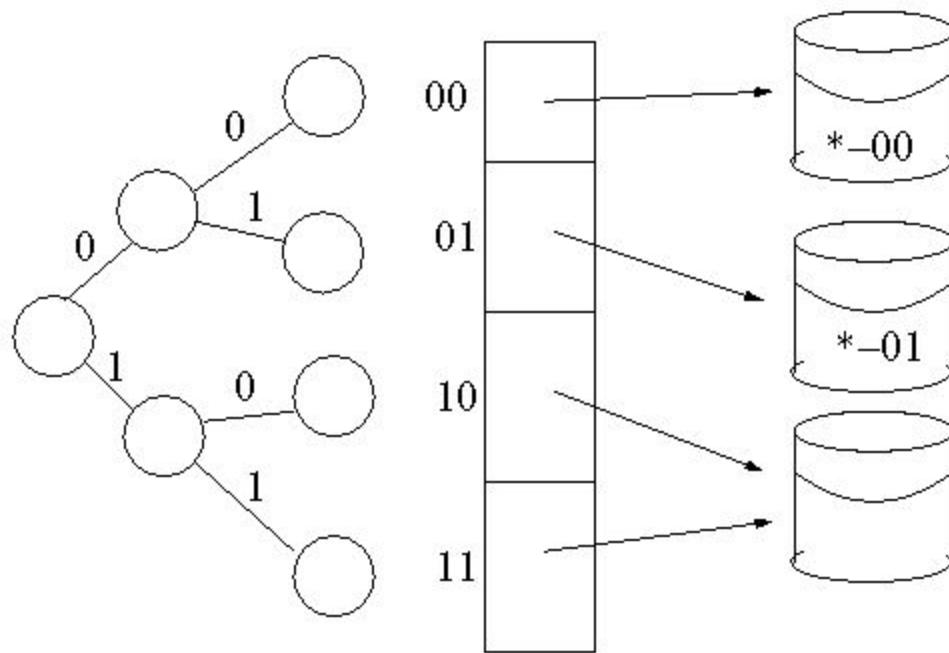
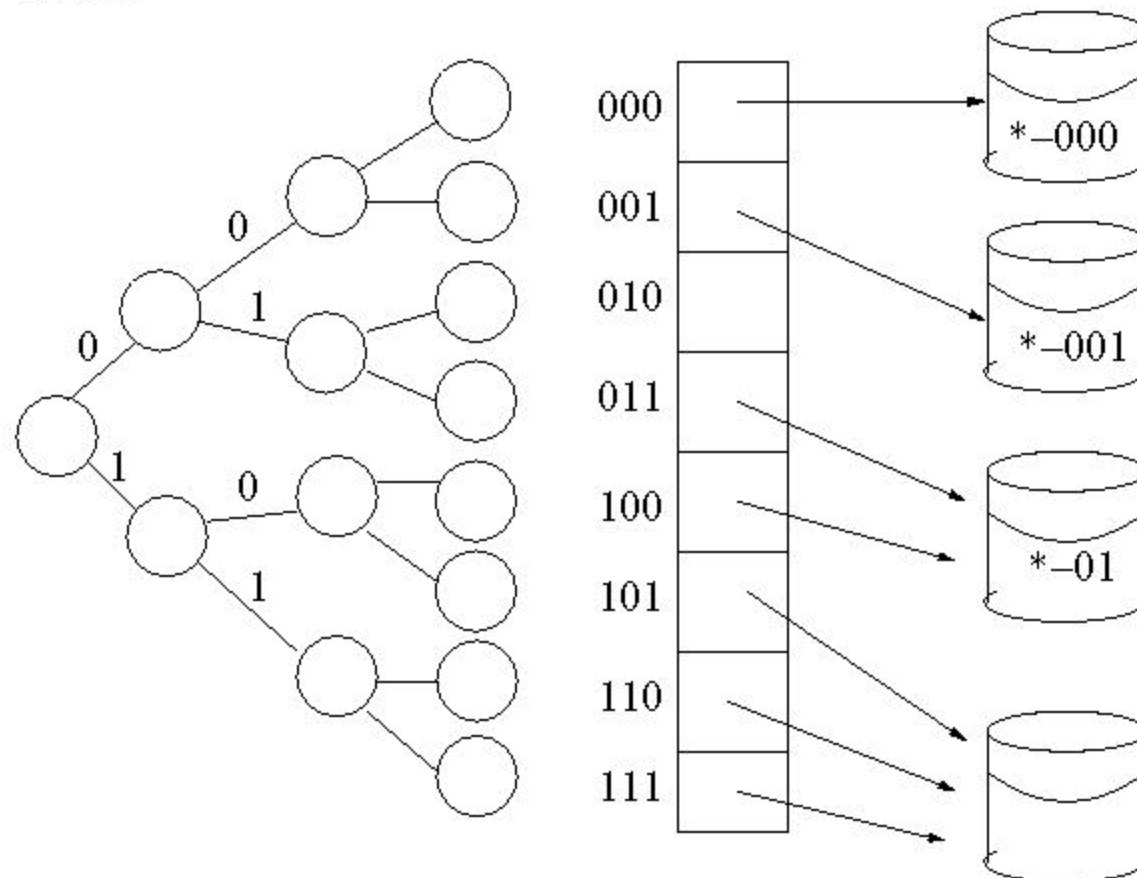


Illustration of Extendible hashing

The address space splits again, this time keys need to be distinguished by three bits:
000, 001, and 010



References

*Reema Thareja, “Data Structures Using C”, Oxford Higher Education, First Edition, 2011.

<https://www.slideshare.net/HanifDurad/chapter-12-ds>

<https://www.slideshare.net/sumitbardhan/358-33-powerpointslides-15hashingcollisionchapter15>

*andrew.cmu.edu/course/15-310/applications/ln/lecture12.html