SORTING & SEARCHING.

Looking for data.

SORTING.

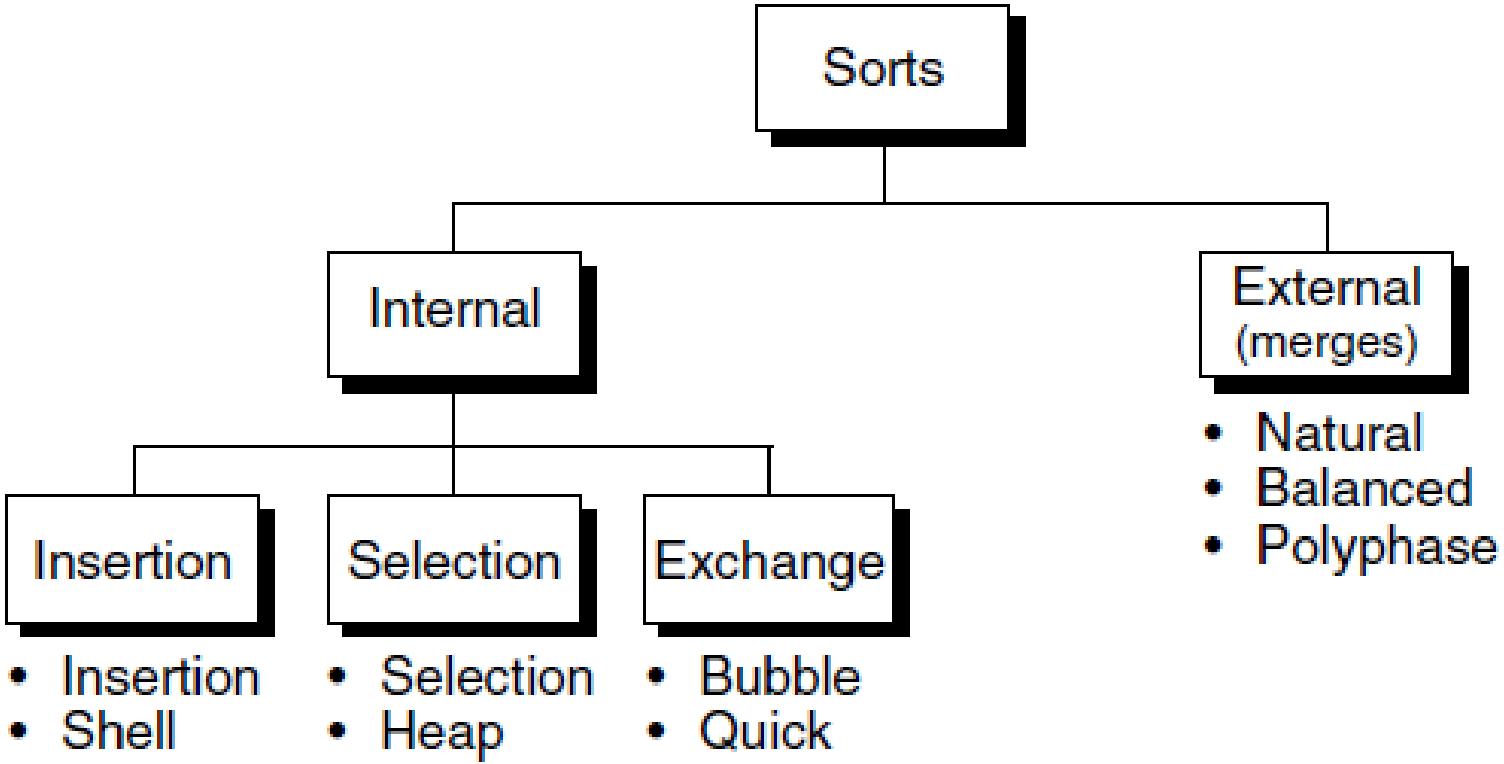
One of the most common data-processing applications.

The process through which data are arranged according to their values is called *SORTING*.

If data were not ordered, hours spent on trying to find a single piece of information.

Example: The difficulty of finding someone's telephone number in a telephone book that had no internal order.

SORT CLASSIFICATIONS.



TYPES OF SORTS.

All data are held in **primary memory** during the sorting process

Internal

Uses **primary memory** for the current data being sorted.
Secondary storage for data not fitting in primary memory

External

THREE INTERNAL SORTS.

Selection sort

Insertion sort

Bubble sort

Shell sort

Heap sort

Quick sort

SORT ORDER.

Data may be sorted in either **ascending** or **descending** sequence.

If the order of the sort is not specified, it is **assumed** to be ***ascending.***

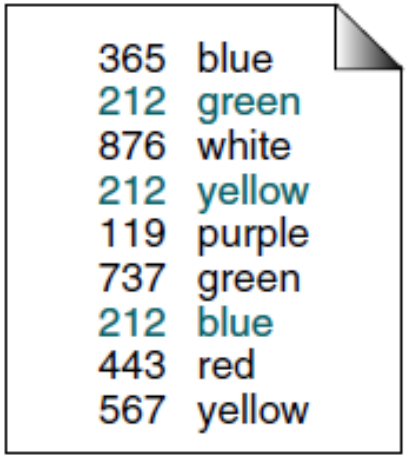
Examples of common data sorted in ascending sequence are the dictionary and the telephone book.

Examples of common descending data are percentages of games won in a sporting event such as baseball or grade-point averages for honor students.

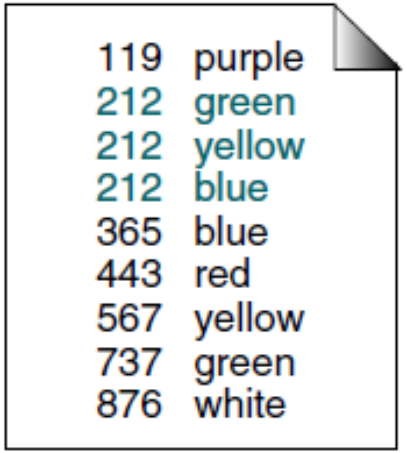
SORT STABILITY.

Is an attribute of a sort, indicating that **data with equal keys maintain their relative input order** in the **output.**

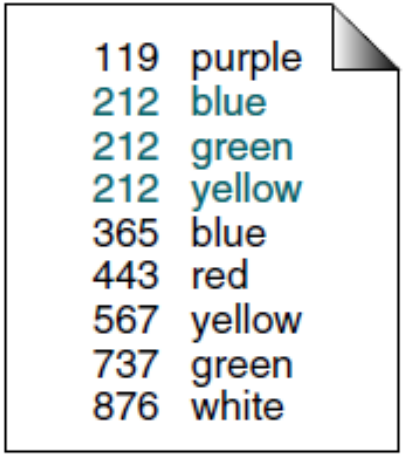
input order



(a) Unsorted data



(b) Stable sort



(c) Unstable sort

output

PASSES.

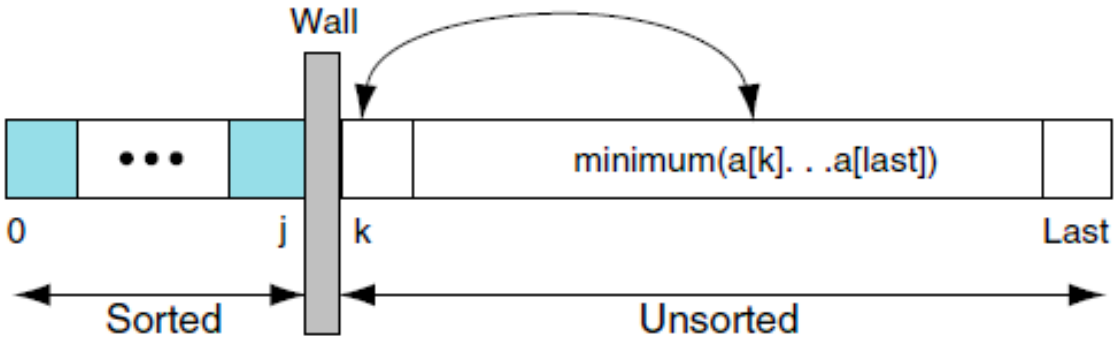
During the sorting process, the data are traversed many times. Each traversal of the data is referred to as a **sort pass.**

Depending on the algorithm, the sort pass may traverse the whole list or just a section of the list.

Also, a characteristic of a sort pass is the placement of one or more elements in a sorted list.

SELECTION SORT.

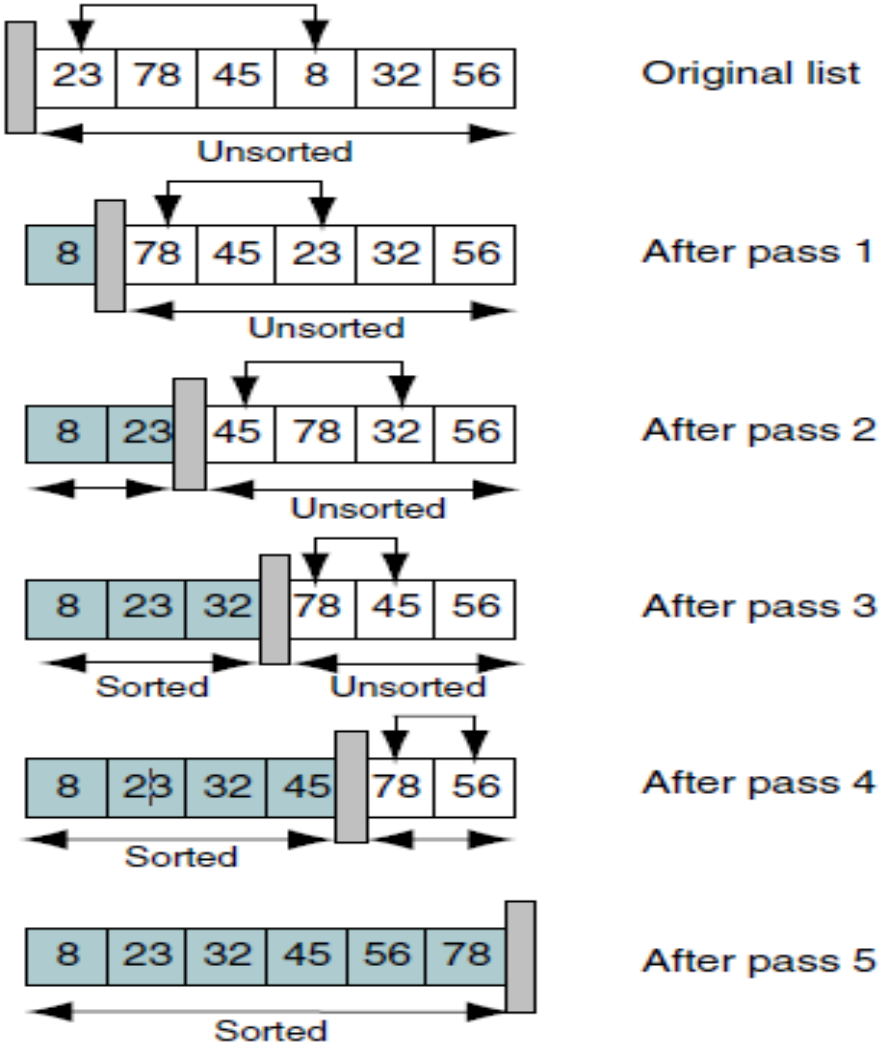
- In each pass of the selection sort, the smallest element is selected from the unsorted sublist and exchanged with the element at the beginning of the unsorted list.
- If there is a list of **n** elements, therefore, **n – 1** passes are needed to completely rearrange the data.



SELECTION SORT

5 3 4 1 2

Example 2

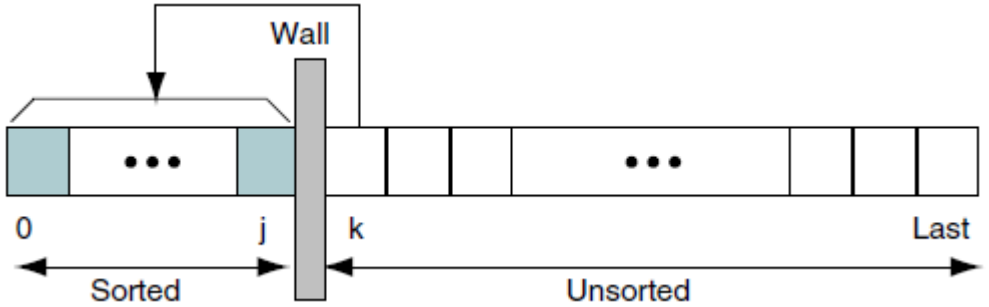


Selection Sort

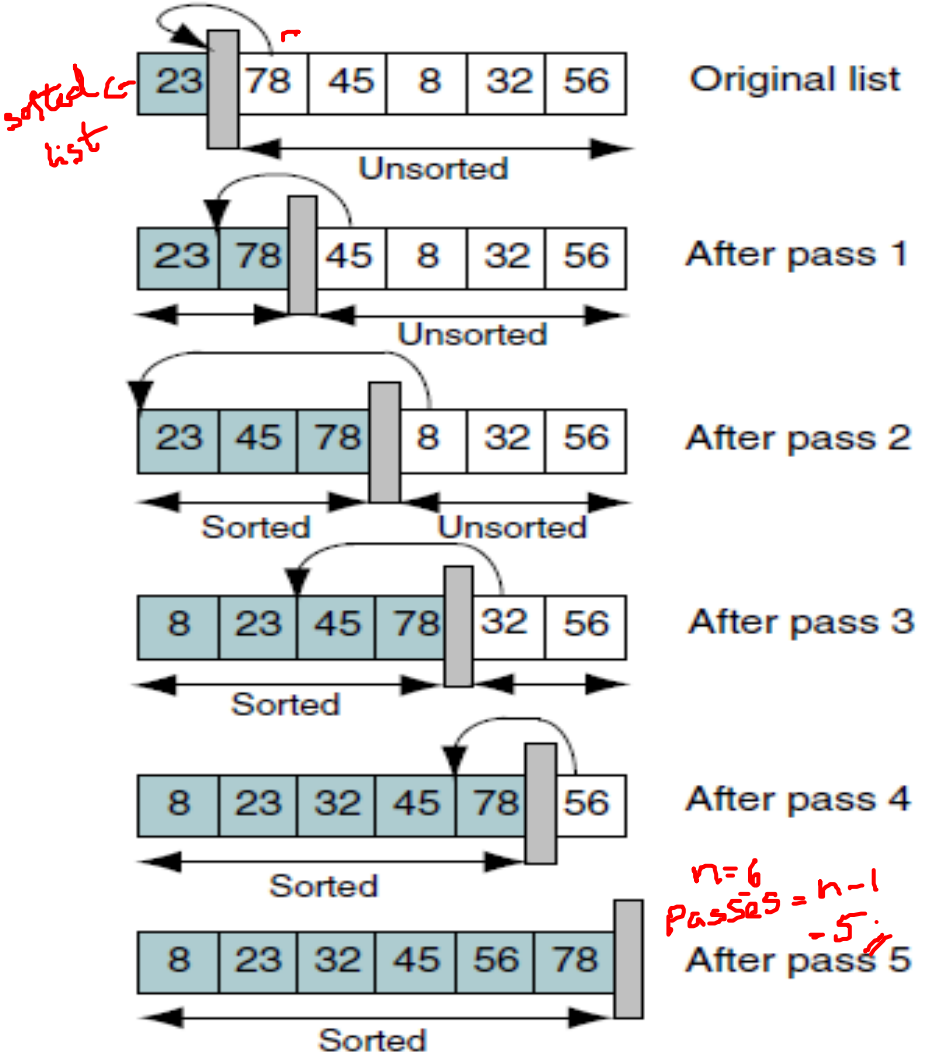
```
Algorithm selectionSort (list, last)
Sorts list array by selecting smallest element in
unsorted portion of array and exchanging it with element
at the beginning of the unsorted list.
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged smallest to largest
 1 set current to 0
 2 loop (until last element sorted)
   1   set smallest to current
   2   set walker   to current + 1
   3   loop (walker <= last)
       1   if (walker key < smallest key)
           1   set smallest to walker
       2   increment walker
   4   end loop
       Smallest selected: exchange with current element.
   5   exchange (current, smallest)
   6   increment current
 3 end loop
end selectionSort
```

INSERTION SORT.

- Given a list, it is divided into two parts: sorted and unsorted.
- In each pass the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.
- If list has **n** elements, it will take at most **n – 1** passes to sort the data.



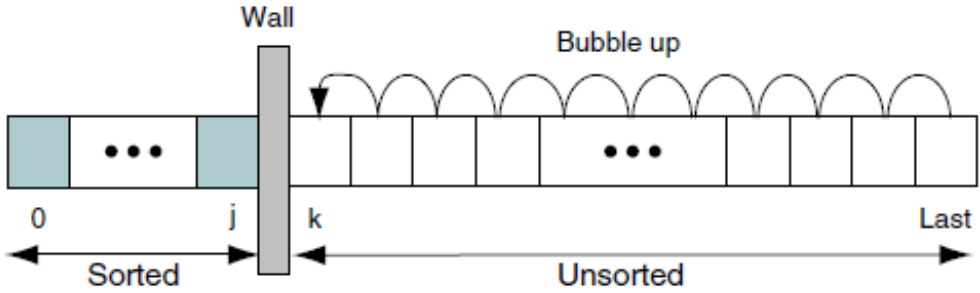
INSERTION SORT.



Straight Insertion Sort

```
Algorithm insertionSort (list, last)
Sort list array using insertion sort. The array is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list.
   Pre  list must contain at least one element
        last is an index to last element in the list
   Post list has been rearranged
 1 set current to 1
 2 loop (until last element sorted)
   1   move current element to hold
   2   set walker to current - 1
   3   loop (walker >= 0 AND hold key < walker key)
       1   move walker element right one element
       2   decrement walker
   4   end loop
   5   move hold to walker + 1 element
   6   increment current
 3 end loop
end insertionSort
```

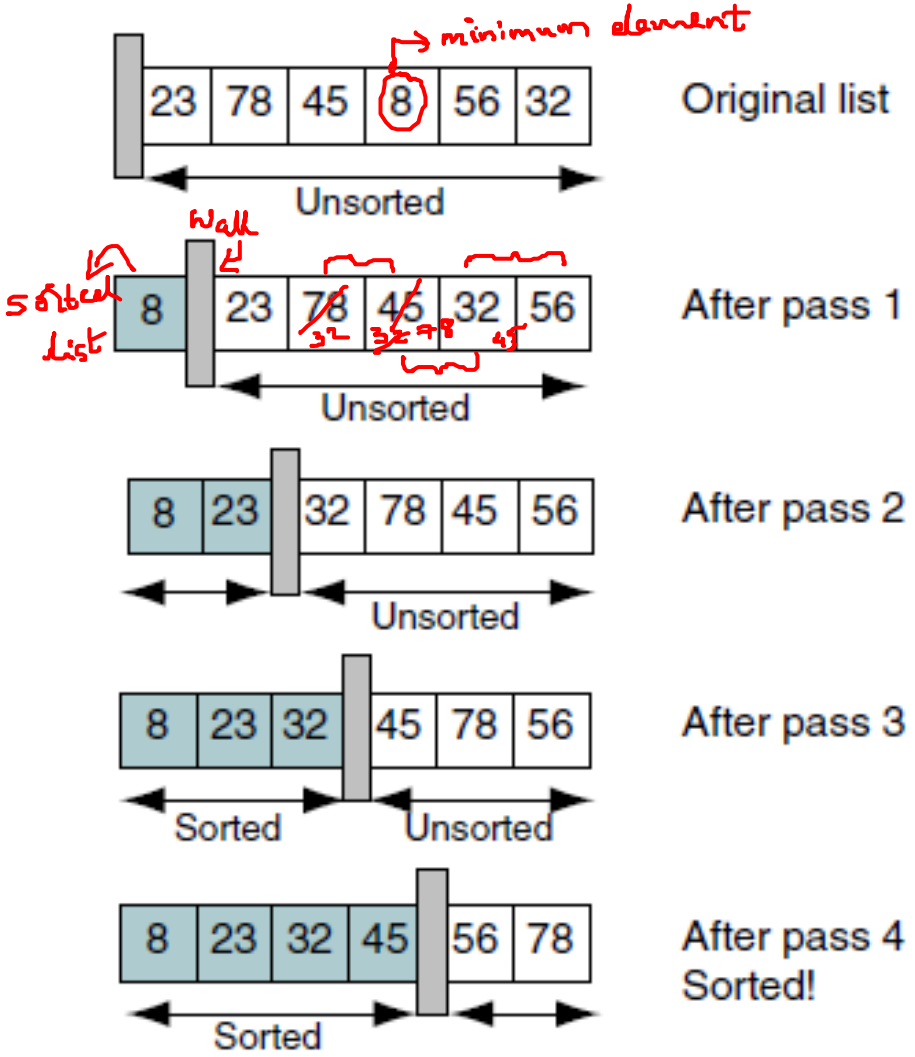
BUBBLE SORT.



- The list is divided into two sublists: sorted and unsorted.
- The smallest element is bubbled from the unsorted sublist and moved to the sorted sublist.
- After moving the smallest to the sorted list, the wall moves one element to the right, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted sublist to the sorted sublist, one sort pass is completed.
- Given a list of **n elements**, the bubble sort requires up to **n – 1 passes** to sort the data.

BUBBLE SORT.

6 5 3 1 8 7 2 4



```
Algorithm  bubbleSort (list, last)
Sort an array using bubble sort. Adjacent
elements are compared and exchanged until list is
completely ordered.

   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged in sequence low to high
 1 set current to 0
 2 set sorted  to false
 3 loop (current <= last AND sorted false)
      Each iteration is one sort pass.
   1  set walker to last
   2  set sorted to true
   3  loop (walker > current)
      1  if (walker data < walker - 1 data)
            Any exchange means list is not sorted.
         1  set sorted to false
         2  exchange (list, walker, walker - 1)
      2  end if
      3  decrement walker
   4  end loop
   5  increment current
 4 end loop
end bubbleSort
```

QUICK SORT.

- In **Quick sort**, also an exchange sort method, developed by **C. A. R. Hoare in 1962.**
- **Quick sort is an exchange sort in which a pivot key is placed in its correct position in the array while rearranging other elements widely dispersed across the list.**
- **More efficient** than the bubble sort because a typical **exchange** involves ***elements that are far apart***, so ***fewer exchanges*** are required to correctly position an element.

QUICK SORT.

- Also called **partition-exchange sort**.
- **Each iteration** of the quick sort **selects an element**, known as **pivot**, and **divides the list into three groups**:
  - **Partition of elements** whose **keys are less than the pivot's key**,
  - **Pivot element** placed in its ultimately correct location in the list,
  - **Partition of elements greater than or equal to the pivot's key**.
- **Pivot** element can be **any element from the array**, it can be the **first** element, the **last** element or any **random** element.
- Approach is **recursive.**

LOGIC OF PARTITION.

- In the array **{52, 37, 63, 14, 17, 8, 6, 25}** , 25 is taken as **pivot**.
- **First pass: {6, 8, 17, 14, 25, 63, 37, 52}**
- After the first pass, **pivot** will be **set at its position** in the final sorted array, with all the ***elements smaller*** to it on its ***left*** and all the **elements larger** than to its ***right***.
- Next, **{6 8 17 14}** and **{63 37 52}** are considered as two separate subarrays
- Same **recursive logic** will be applied on them, and keep doing this until the complete array is sorted.

HOW DOES QUICK SORT WORK?

Selection of pivot to partition the array

Pivot (key) = First element; Find a position for it

Left partition < pivot, Right partition ≥ pivot

Repeat recursively for Left and Right partitions

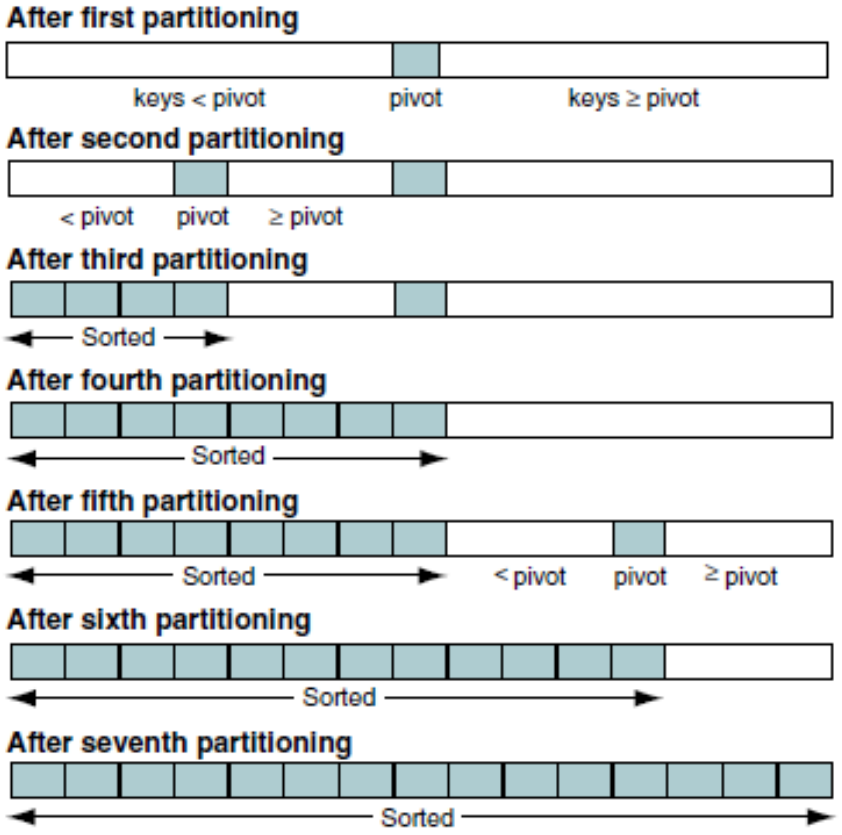
Can be used for finding the k^{th} smallest OR largest element in an array without completely sorting the array of size n.

When the pivot element is placed in $(k-1)^{th}$ position, it will be the k^{th} smallest element

When the pivot element is placed in $(n-k)^{th}$ position, it will be the k^{th} largest element

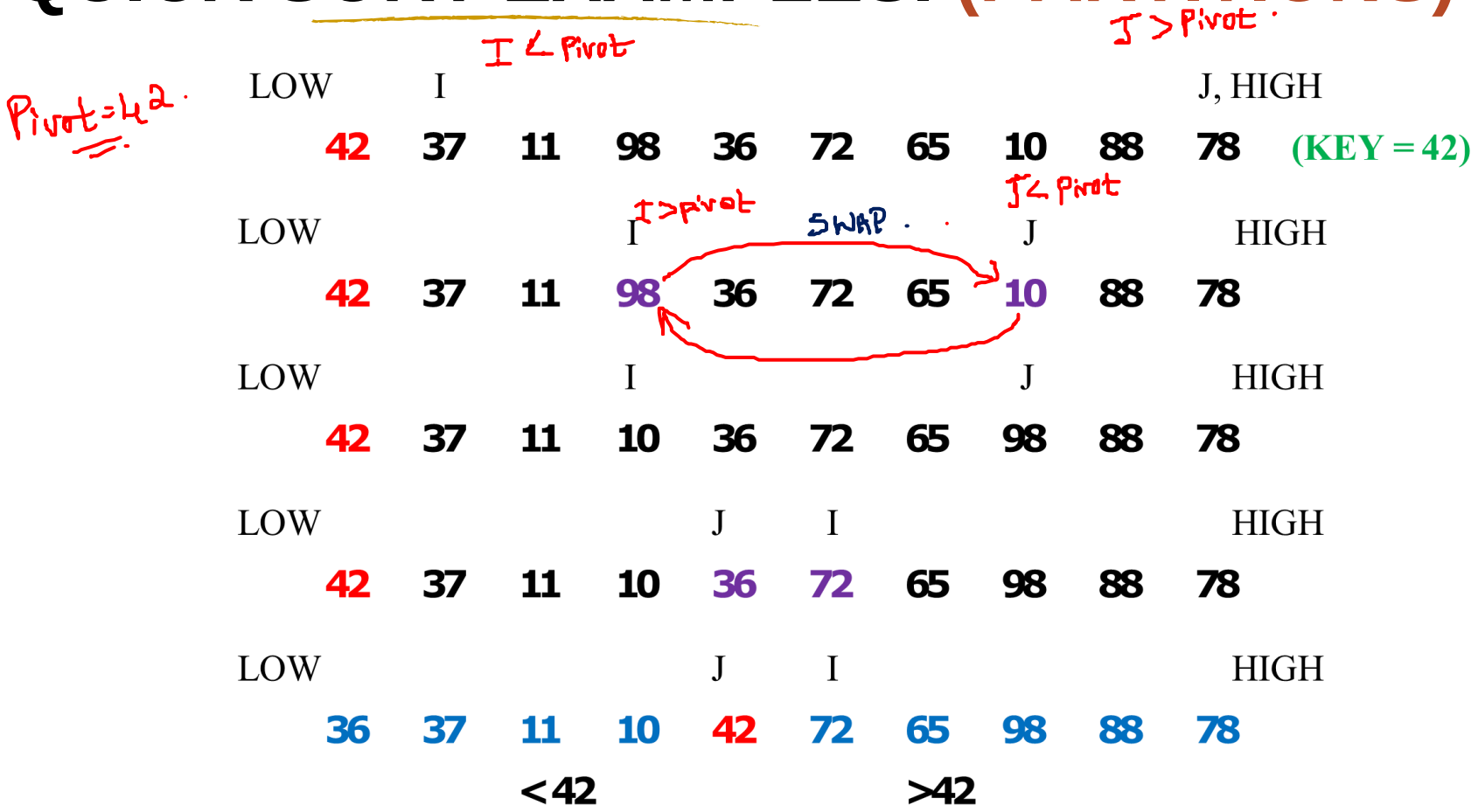
QUICK SORT ALGORITHM.

- Two Algorithms:
  - Quick Sort Recursive
  - Partition

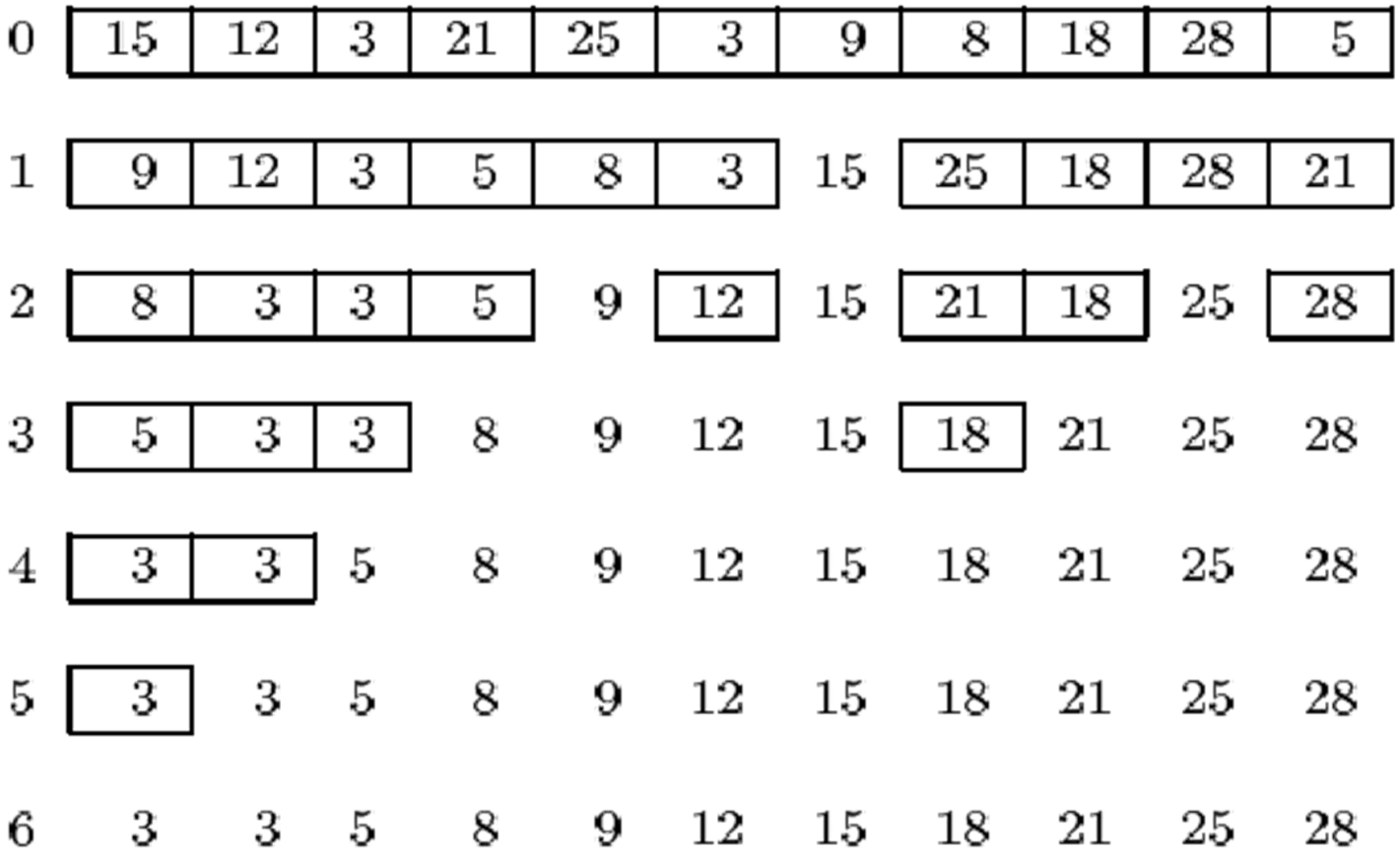


Quick Sort Partitions

QUICK SORT EXAMPLES. (PARTITIONS)

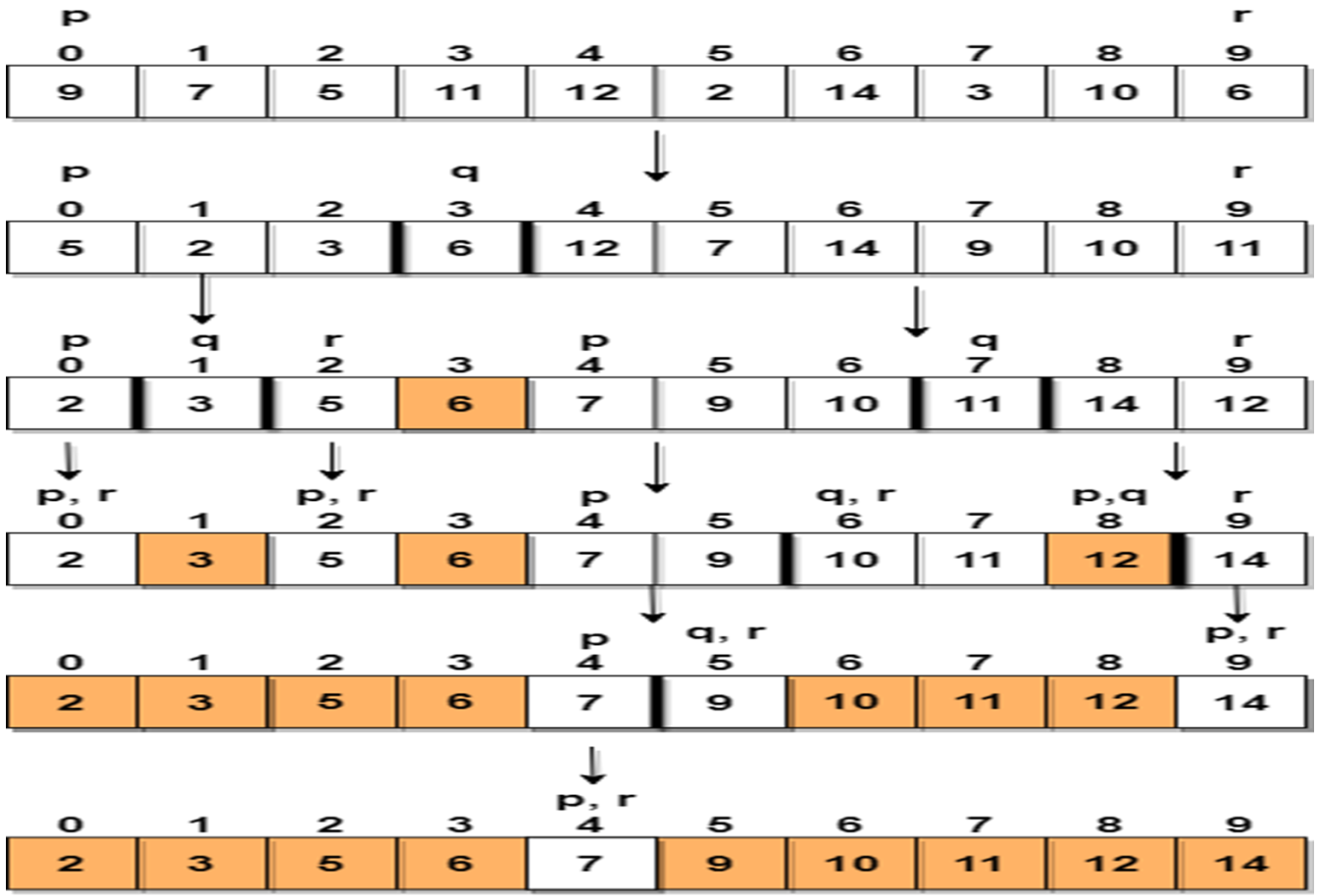


QUICK SORT EXAMPLES.



QUICK SORT EXAMPLES.

Pivot Element = 6



QUICK SORT ALGORITHM.

***Algorithm* Quicksort (Array, Low, High)**

1. If (Low < High) Then
    1. Set Mid = Partition (Array, Low, High)
    2. Quicksort (Array, Low, Mid – 1)
    3. Quicksort (Array, Mid + 1, High)
2. End

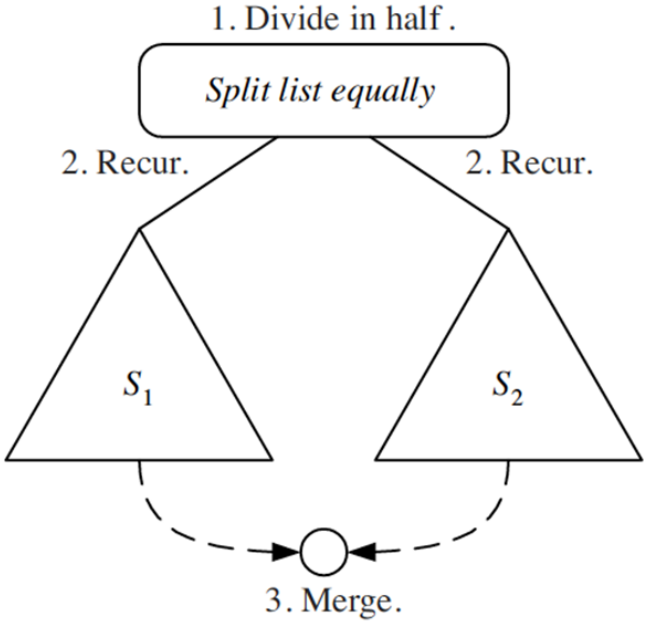
QUICK SORT ALGORITHM.

***Algorithm* Partition (Array, Low, High)**

1. Set Key = Array[low], I = Low + 1, J = High
2. Repeat Steps A through C
   A. while (I < High && Key ≥ Array[i]) **i++**
   B. while (Key < Array[j]) **j- -**
   C. if (I < J) then
      swap Array[i] with Array[j]
      else
      swap Array[low] with Array[j]
      return j *{Position For KEY}*
3. End

DIVIDE & CONQUER.

- Divide-and conquer is a general algorithm design paradigm.
- **Divide:** divide the input data S in two disjoint subsets S1 and S2.
- **Recur:** solve the subproblems associated with S1 and S2.
- **Conquer:** combine the solutions for S1 and S2 into a solution for S.
- Base case for recursion are subproblems of size 0 or 1.



MERGE SORT.

- Invented by ***John von Neumann*** in 1945, example of Divide & Conquer.
- Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.
- **Basic algorithm:**
  - Divide the list into two roughly equal halves.
  - Sort the left half.
  - Sort the right half.
  - Merge the two sorted halves into one sorted list.
- Often implemented **recursively**
- Runtime: **O(n log n).**

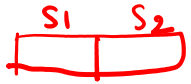
MERGE SORT ALGORITHM.

Algorithm mergeSort(S)

Input: Sequence S with n elements

Output: Sequence S sorted

```
if S.size() > 1
    (S1, S2) ← partition(S, n/2)
    mergeSort(S1)
    mergeSort(S2)
    S ← merge(S1, S2)
```



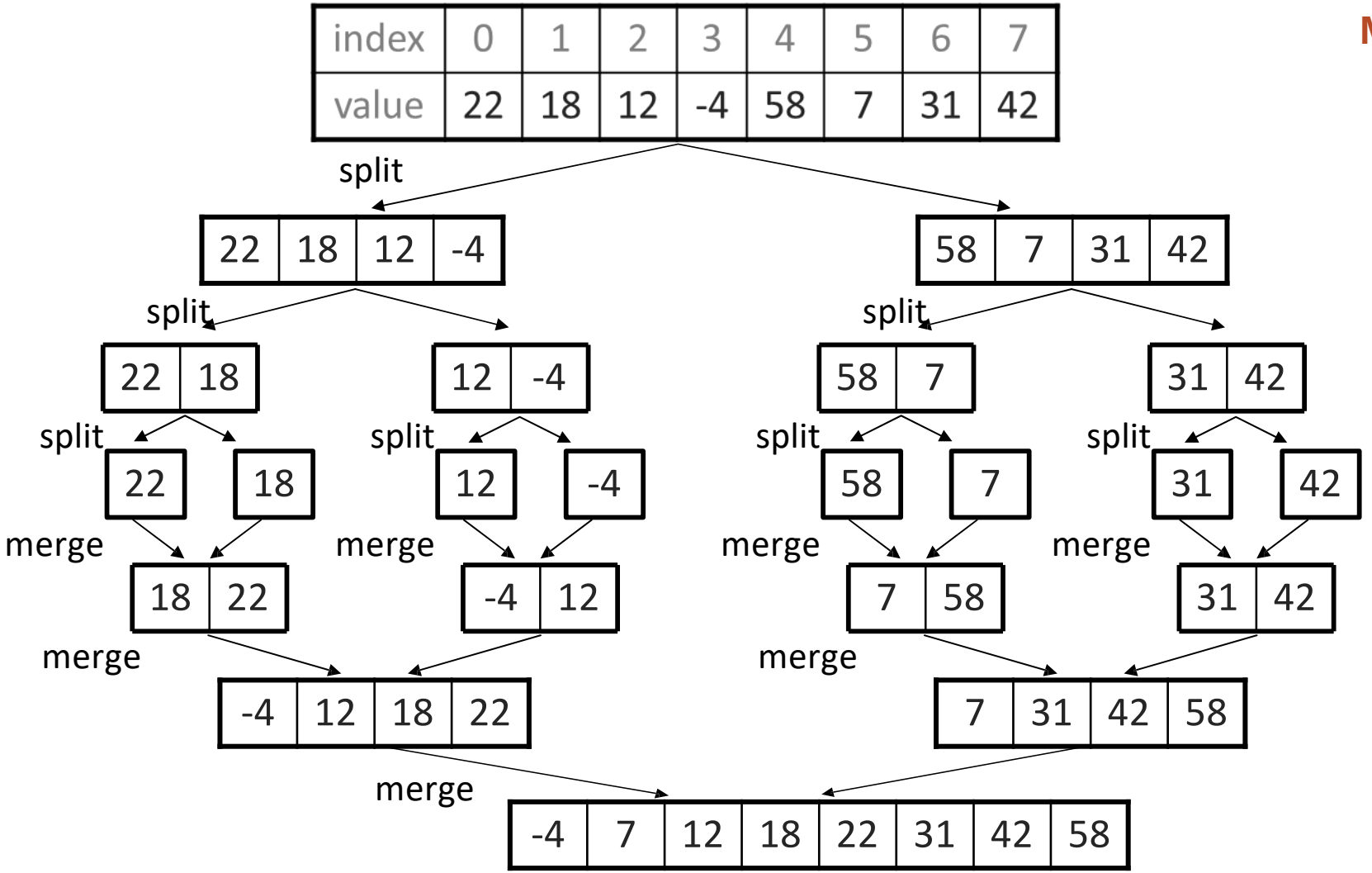
Algorithm merge(S1, S2, S)

Input: Two arrays S1 & S2 of size n1 and n2 sorted in non decreasing order and an empty array S of size at least (n1+n2)

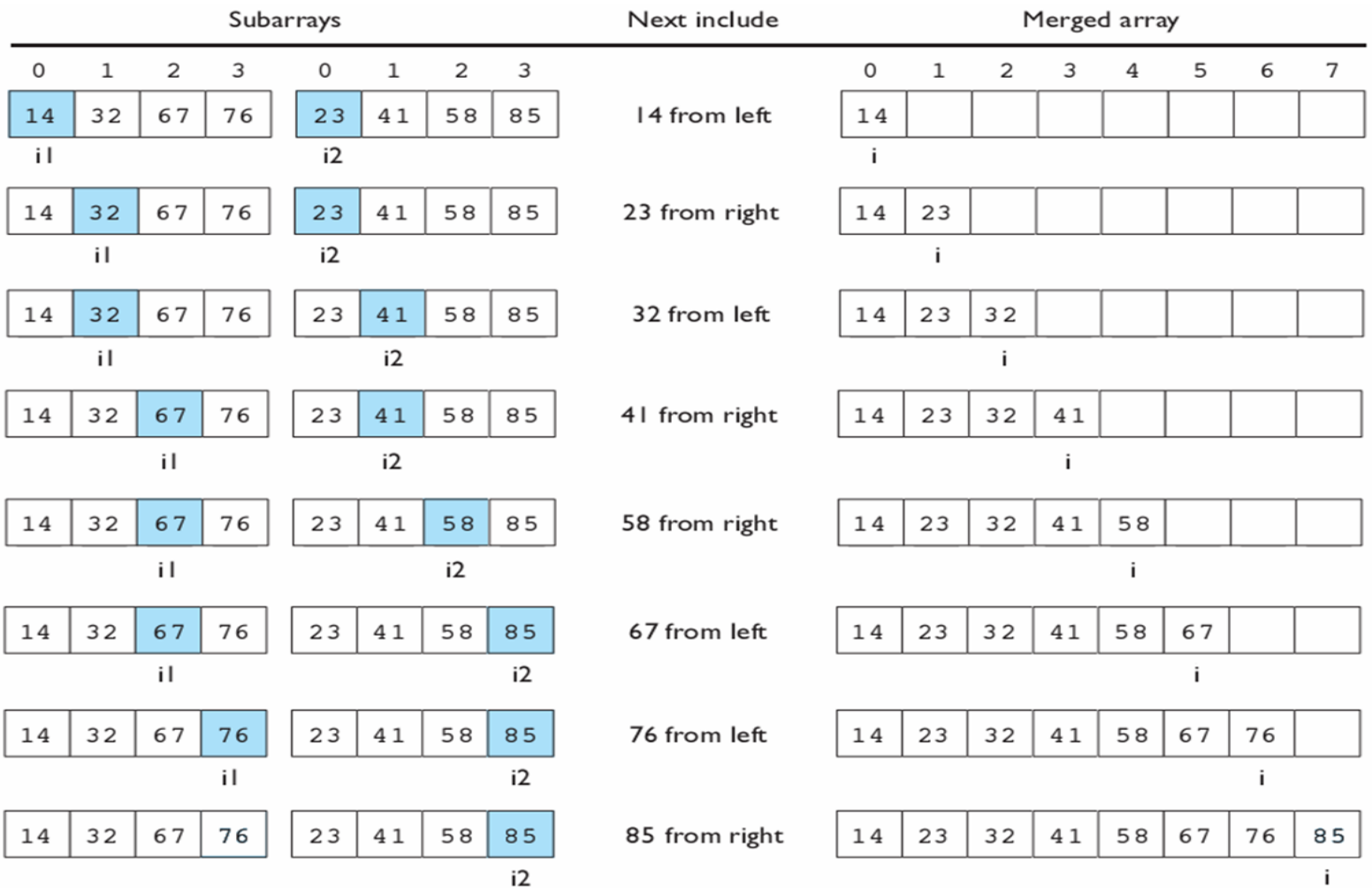
Output: S containing elements from S1 & S2 in sorted order.

```
i←1
j←1
while (i ≤ n1) and (j ≤ n2)
        if S1[i] ≤ S2[j] then
            S[i+j-1] ← Si[i]
            i ← i +1
        else
            S[i+j-1] ← Si[j]
            j ← j +1
while ((i ≤ n1) do
        S[i+j-1] ← Si[i]
        i ← i +1
while (j ≤ n2)
        S[i+j-1] ← Si[j]
        j ← j +1
```

Merge Sort example.



Merge Sort example.

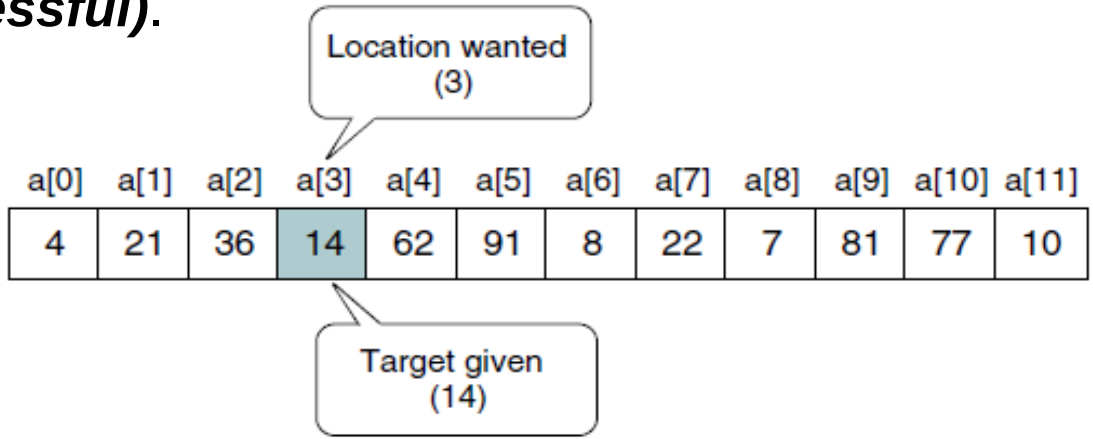


SEARCHING.

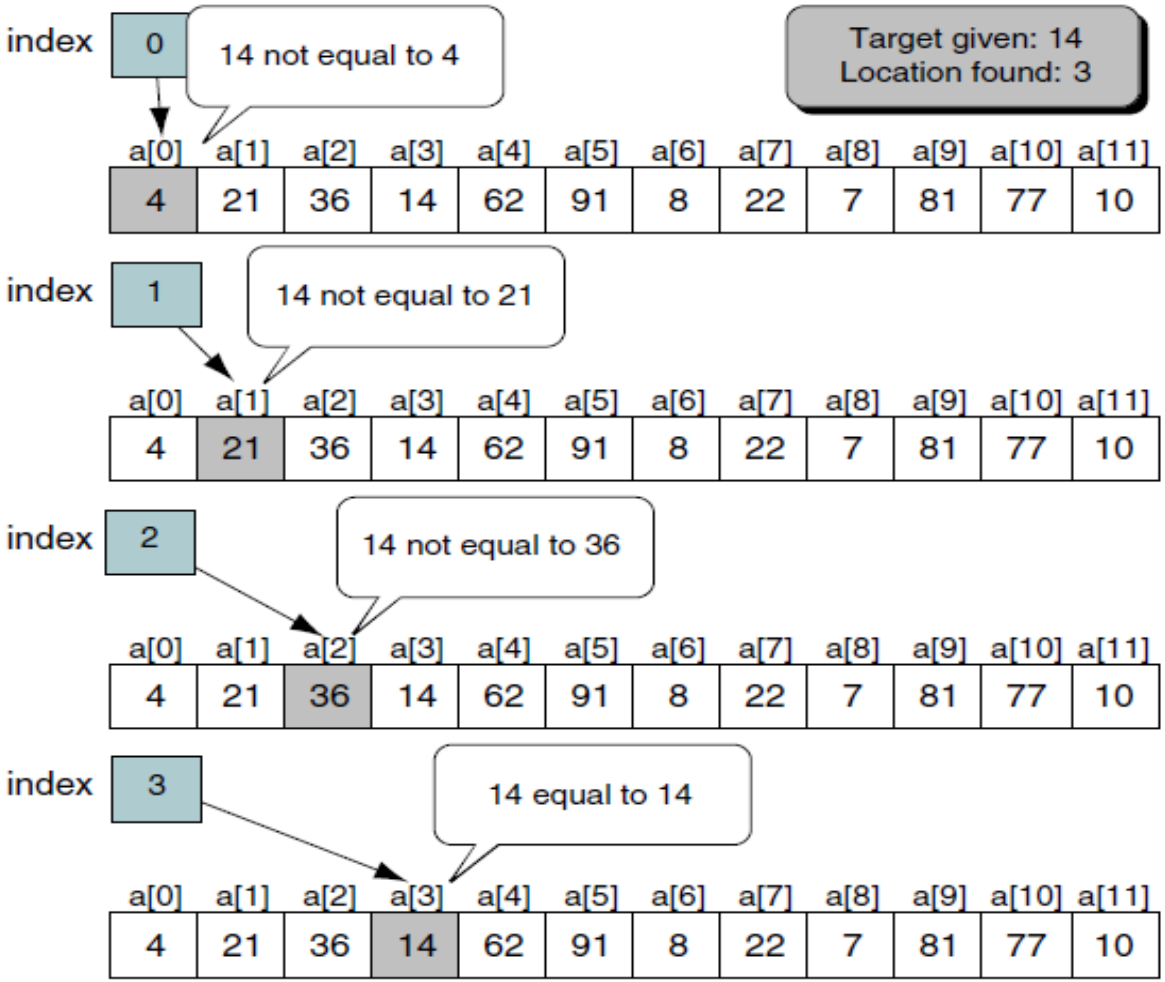
- One MORE common and time-consuming operations in computer science is searching
- The process used to find the location of a target among a list of objects.
- The two basic search algorithms:
- **Sequential search** including three interesting variations and,
- **Binary search.**

SEQUENTIAL SEARCH.

- Used whenever the list is not ordered.
- Generally, technique used only for small lists or lists that are not searched often.
- Process: Start searching for the target at the beginning of the list and continue until target found or it is not in the list.
- This approach has two possibilities: Find element ***(successful)*** or reach end of list ***(unsuccessful)***.

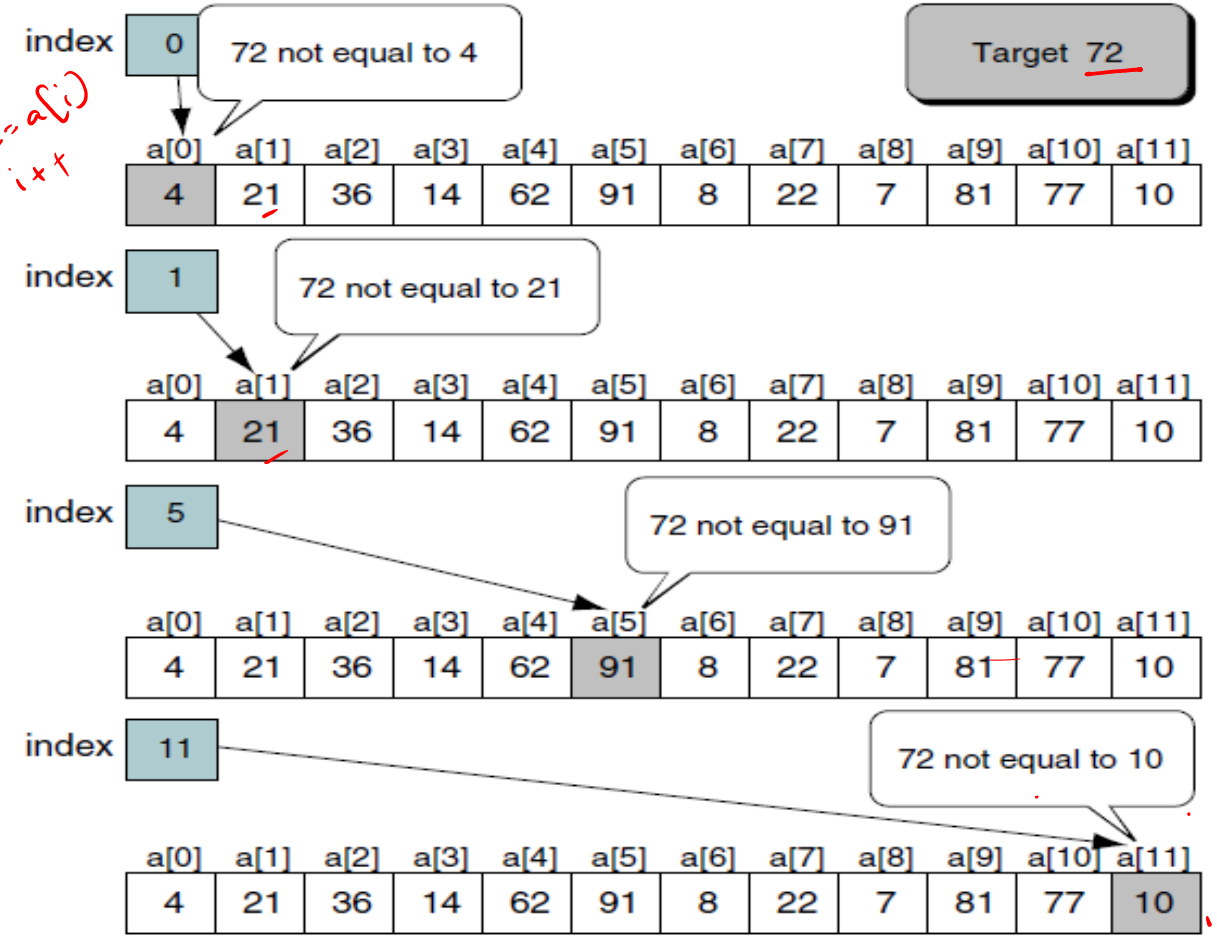


Sequential Search Example for Successful search.



Successful Search of an Unordered List

Sequential Search Example for Unsuccessful search.



Note: Not all test points are shown.

Unsuccessful Search in Unordered List

```
Algorithm seqSearch (list, last, target, locn)
Locate the target in an unordered list of elements.
   Pre    list must contain at least one element
          last is index to last element in the list
          target contains the data to be located
          locn is address of index in calling algorithm
   Post   if found: index stored in locn & found true
          if not found: last stored in locn & found false
   Return found true or false
 1 set looker to 0
 2 loop (looker < last AND target not equal list[looker])
   1   increment looker
 3 end loop
 4 set locn to looker
 5 if (target equal list[looker])
   1   set found to true
 6 else
   1   set found to false
 7 end if
 8 return found
end seqSearch
```

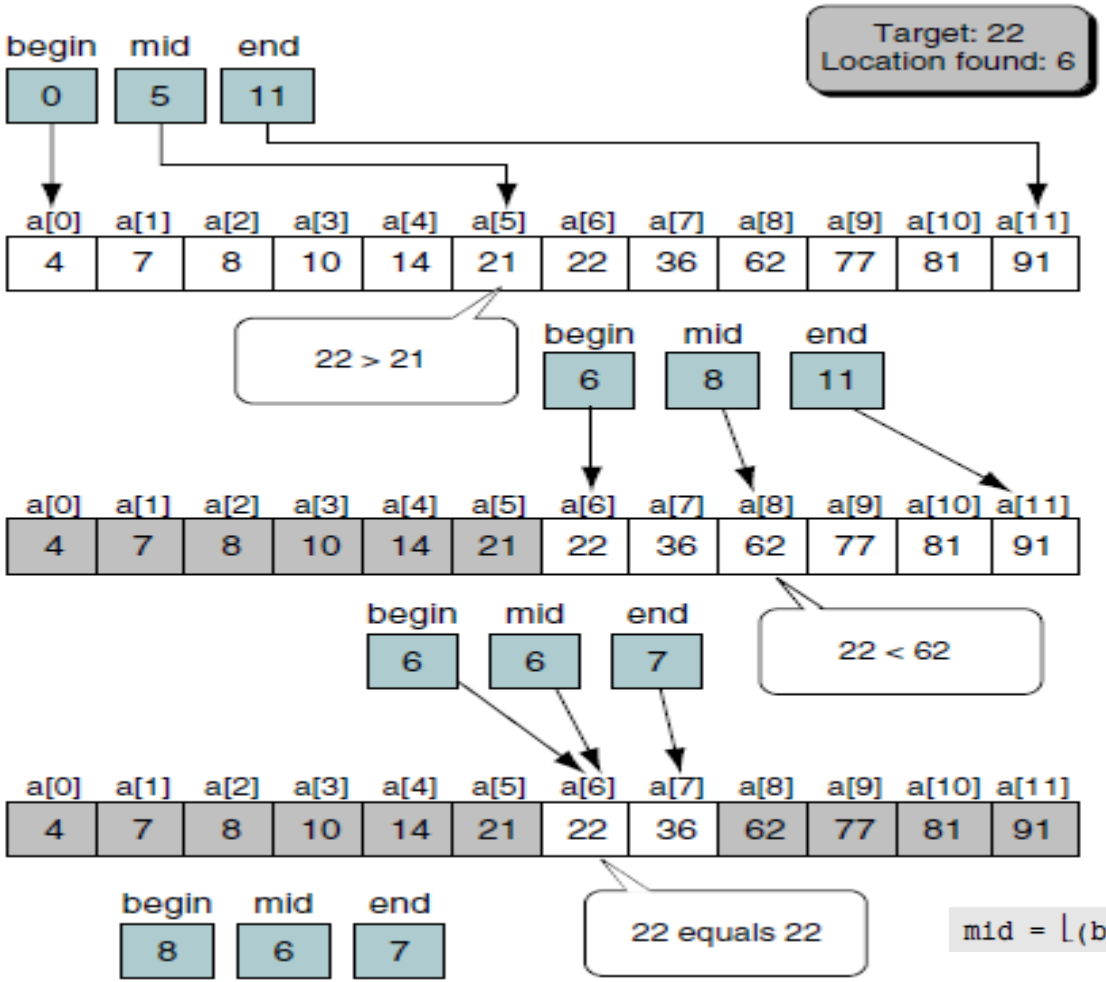
BINARY SEARCH.

- Sequential search algorithm is very slow. If an array of 1000 elements, exists, 1000 comparisons are made in worst case.
- If the array is not sorted, the sequential search is the only solution.
- However, if the array is sorted, we can use a more efficient algorithm called ***binary search***.
- Generally speaking, Binary search used whenever the list starts to become large.

BINARY SEARCH.

- Begins by testing the data in the element at the middle of the array to determine if the ***target is in the first or the second half of the list.***
- If **target in first half**, there is NO need to check the second half.
- If **target in second half**, NO need to test the first half.
- In other words, half the list is eliminated from further consideration with just one comparison.
- This process repeated, eliminating half of the remaining list with each test, until target if found or does not exist in the list.
- To find the middle of the list, three variables needed: one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list.

Binary Search Example for Successful search.

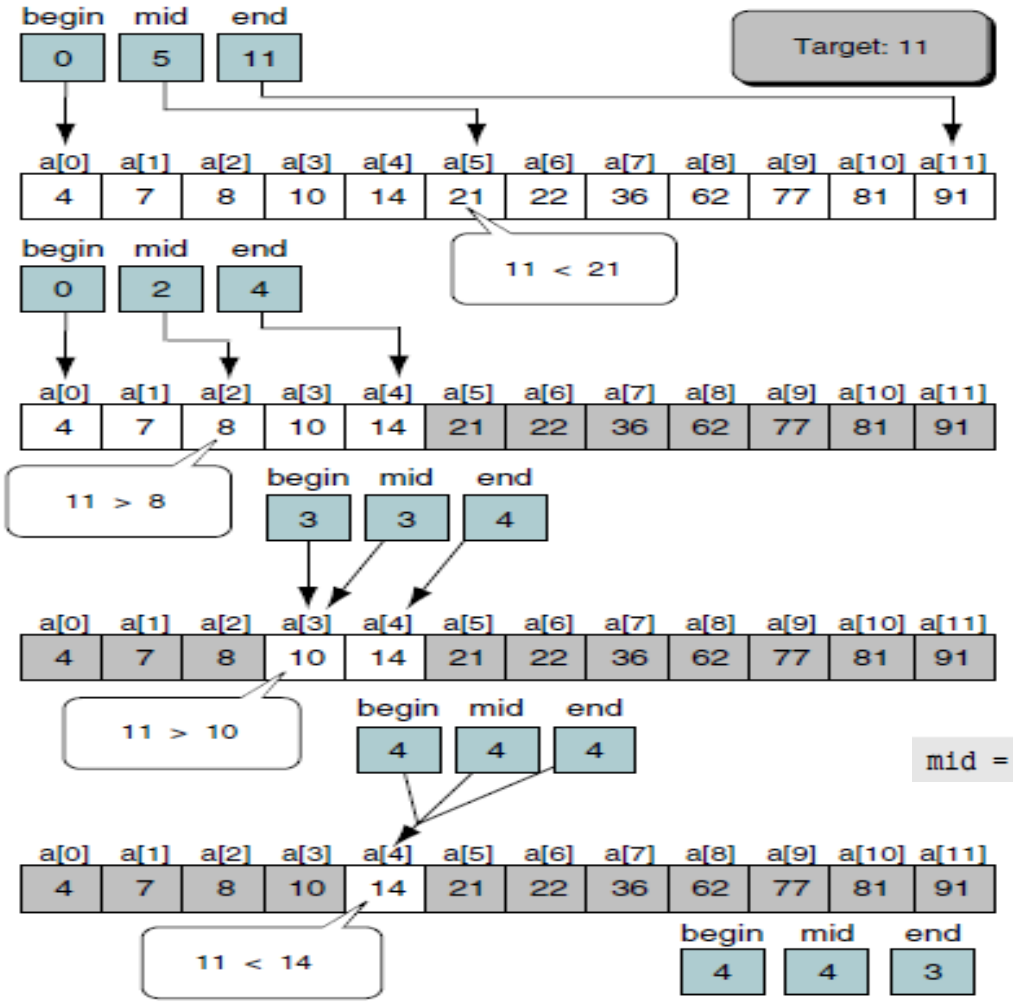


$n/2$

$$mid = \lfloor (begin + end) / 2 \rfloor = \lfloor (0 + 11) / 2 \rfloor = 5$$

Successful Binary Search Example

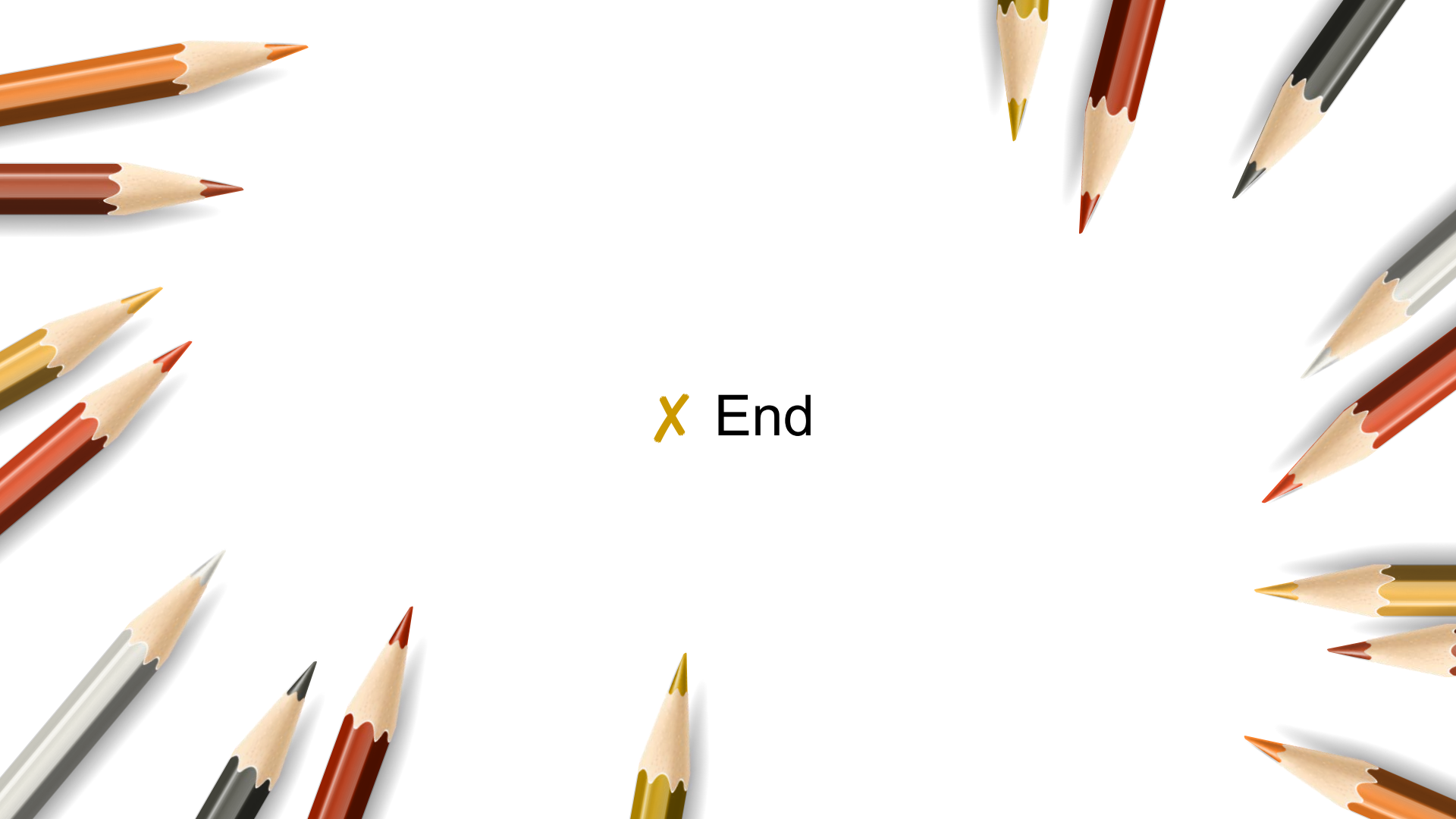
Binary Search Example for Unsuccessful search.



$$mid = \lfloor (begin + end) / 2 \rfloor = \lfloor (0 + 11) / 2 \rfloor = 5$$

Unsuccessful Binary Search Example

```
Algorithm binarySearch (list, last, target, locn)
Search an ordered list using Binary Search
   Pre     list is ordered; it must have at least 1 value
           last is index to the largest element in the list
           target is the value of element being sought
           locn is address of index in calling algorithm
   Post    FOUND: locn assigned index to target element
                  found set true
           NOT FOUND: locn = element below or above target
                      found set false
   Return found true or false
 1 set begin to 0
 2 set end to last
 3 loop (begin <= end)
   1   set mid to (begin + end) / 2
   2   if (target > list[mid])
           Look in upper half
       1   set begin to (mid + 1)
   3   else if (target < list[mid])
           Look in lower half
       1   set end to mid - 1
   4   else
           Found: force exit
       1   set begin to (end + 1)
   5   end if
 4 end loop
 5 set locn to mid
 6 if (target equal list [mid])
   1   set found to true
 7 else
   1   set found to false
 8 end if
 9 return found
end binarySearch
```

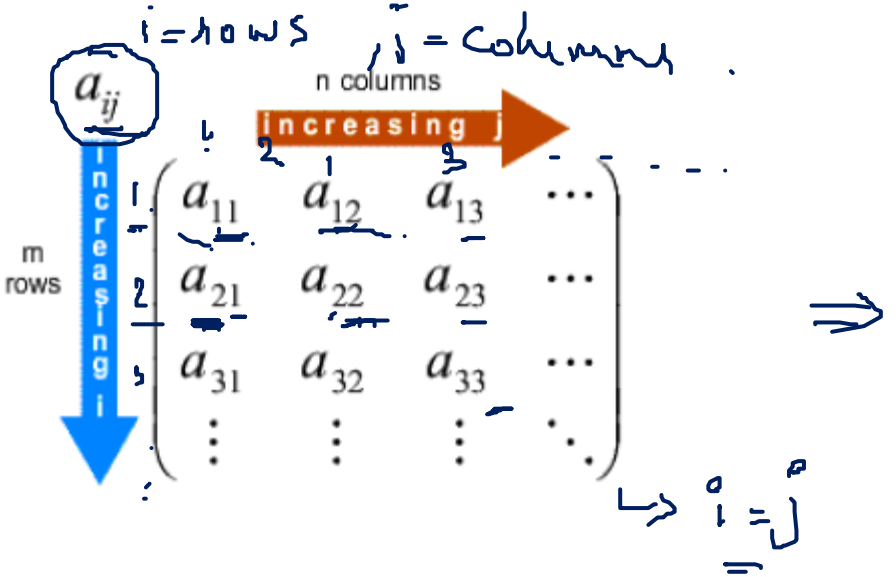

End

Special forms of square matrices- Sparse, Polynomial

2-Dimensional Arrays

Matrices

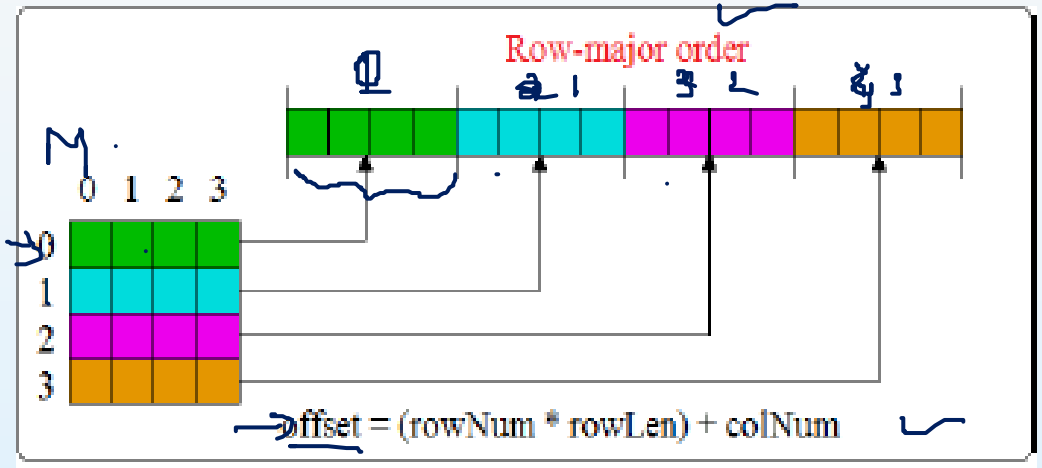
- A m x n matrix is a table with m rows and n columns (m and n are the dimensions of the matrix)

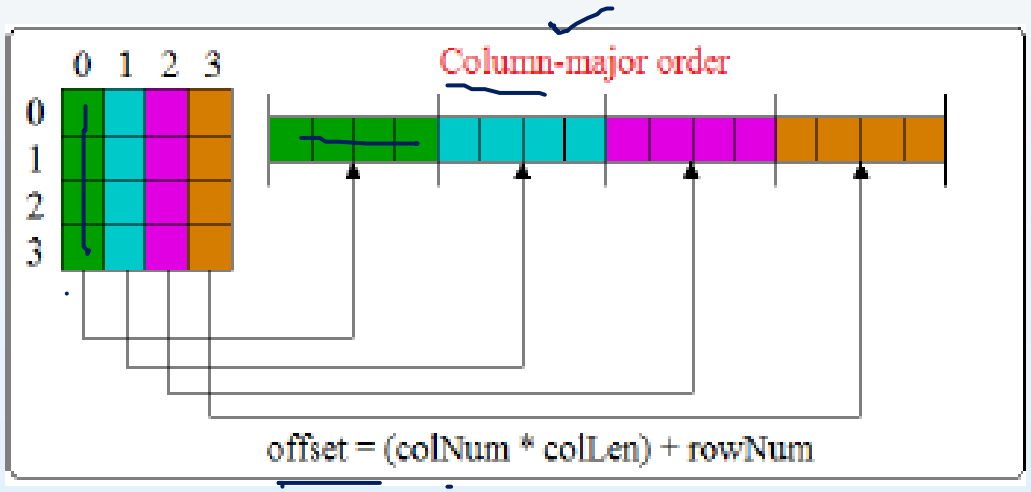


- Operations: addition, substraction, multiplication, transpose, etc.

Matrices (Continued …)

- Representations (mapped as 1-D array)





1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Row-major

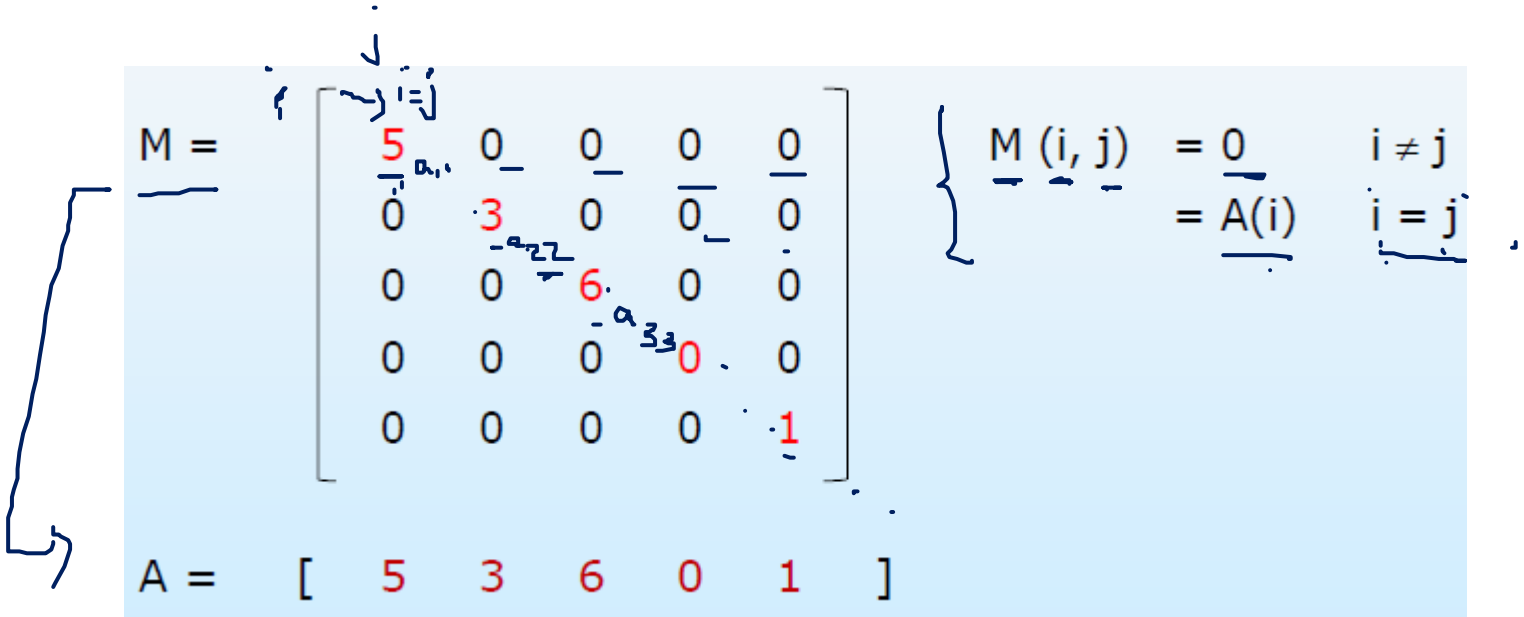
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Column-major

1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16

Special forms of square matrices (m = n)

- Diagonal: M (i, j) = 0 for i ≠ j
  - Diagonal entries = m



Special forms of square matrices (m = n)

- Tridiagonal: M (i, j) = 0 for | i –j | > 1
  - ✓ 3 diagonals: Main →i = j; Upper →i = j + 1; Lower →i= j –1
  - ✓ Number of elements on three diagonals: 3*m-2
  - ✓ Mapping: row-major, column-major or diagonals (begin with lowest)

$$M = \begin{bmatrix} 5 & 8 & 0 & 0 & 0 \\ 7 & 3 & 0 & 0 & 0 \\ 0 & 2 & 6 & 4 & 0 \\ 0 & 0 & 9 & 0 & 3 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M(i, j) = A(2*i + j) \quad |i - j| \leq 1$$

A = [5 8 7 3 0 2 6 4 9 0 3 0 1]

Special forms of square matrices (m = n)

- Triangular matrices
  - ✓ No. of elements in non-zero region: n (n + 1) / 2
  - ✓ Upper triangular: M (i, j) = 0 for i>j
  - ✓ Lower triangular: M (i, j) = 0 for i<j

$$U = \begin{bmatrix} 5 & 8 & 0 & 7 & 2 \\ 0 & 3 & 5 & 6 & 1 \\ 0 & 0 & 6 & 4 & 3 \\ 0 & 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix} \qquad L = \begin{bmatrix} 8 & 0 & 0 & 0 & 0 \\ 7 & 3 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 1 & 2 & 8 & 9 & 0 \\ 5 & 6 & 0 & 1 & 0 \end{bmatrix}$$

- **Upper**: M (i, j) = A ((n * i) + j –(i* (i+ 1) / 2)) for i ≤ j
- **Lower**: M (i, j) = A (i* (i+ 1) / 2 + j) for i ≥ j

2-Dimensional Arrays

Special forms of square matrices (m = n)

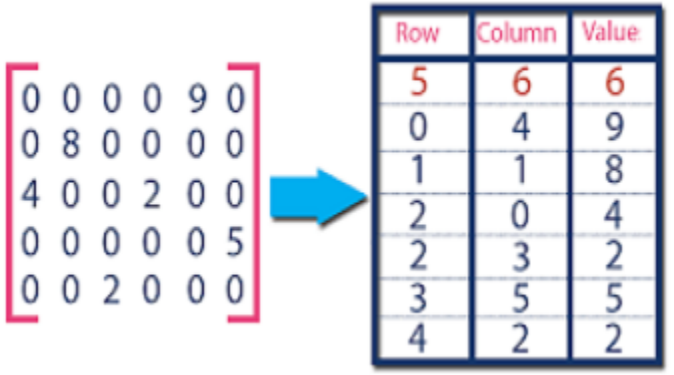
- Symmetric: M (i, j) = M (j, i) for all i and j
  - ✓ Can be stored as lower-or upper-triangular
  - ✓ E.g., Table of inter-city distance for a set of cities

$$\begin{bmatrix} 4 & 5 & 6 & 8 & 3 \\ 5 & 7 & 1 & 3 & 2 \\ 6 & 1 & 0 & 9 & 7 \\ 8 & 3 & 9 & 2 & 0 \\ 3 & 2 & 7 & 0 & 8 \end{bmatrix}$$

Sparse Matrices

Sparse matrices

- Number of non-zero elements is very less compared to total number of elements
- Represented as a 1-D array of triplets
  - ✓ Element 1 onwards -Row, Column and Value –for all non-zero elements
  - ✓ Element 0 –number of rows, columns and non-zero elements



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

Row	Column	Value
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

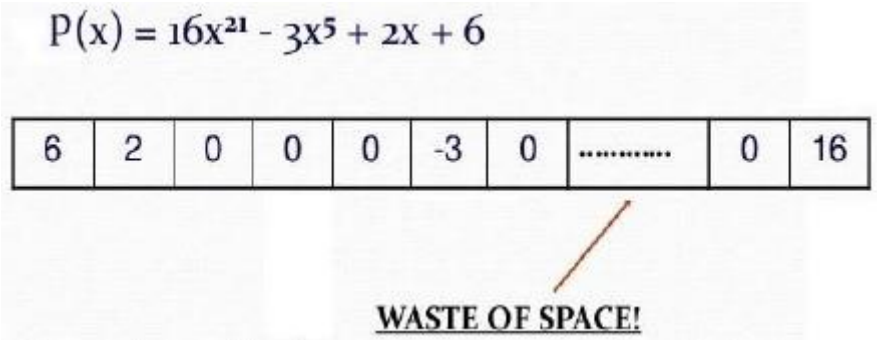
Polynomials

Polynomial

- An expression of the form

$$a_n x^n + a_{(n-1)} x^{(n-1)} + \ldots + a_1 x^1 + a_0$$

- One possible way of representation
  - ✓ Store coefficients of the terms in an array element at position equal to the exponent
  - ✓ Disadvantage: Waste of space specially in case of sparse polynomial



Polynomial(Continued …)

- Represented as a 1-D array of doublets
  - ✓ Element 1 onwards -coefficient and exponent of all terms
  - ✓ Element 0 –number of terms (either in coefficient or exponent field)

$$P(x) = 23x^9 + 18x^7 - 41x^6 + 163x^4 - 5x + 3$$

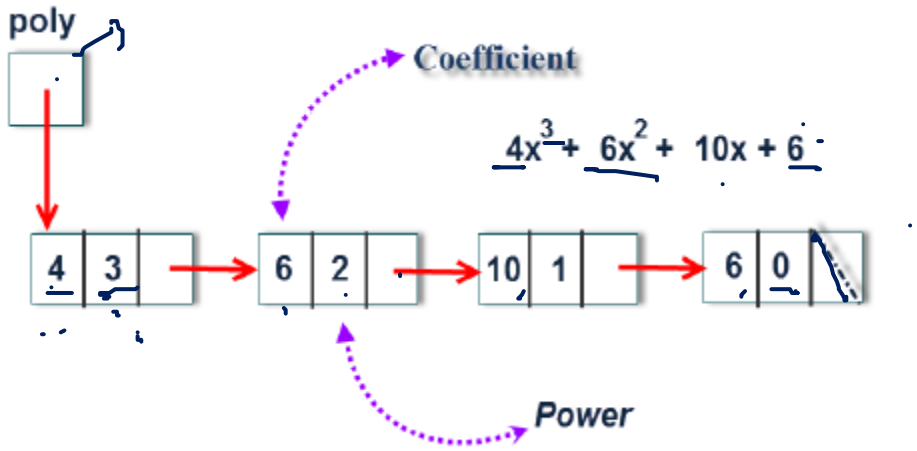
Coefficient

6	23	18	-41	163	-5	3
x	9	7	6	4	1	0
0	1	2	3	4	5	6

Exponent

Polynomial(Continued …)

- Represented as a linked list
  - ✓ Store the coefficient and exponent (power) of each term in a node of a singly linked list



```
//add polynomial as 1d array
#include <stdio.h>
typedef struct
{
	int coeff, exp;
} poly;
```

```
int main()
{
	//system("clear");
	//clrscr();
	poly p1[10],p2[10],p3[20];
	printf("1st polynomial:\n");
	getpoly(p1);
	printf("\n2nd polynomial:\n");
	getpoly(p2);
	printf("\n1st polynomial: ");
	showpoly(p1);
	printf("\n2nd polynomial: ");
	showpoly(p2);
	add(p1,p2,p3);
	printf("\nSum : ");
	showpoly(p3);
	printf("\n");
	return 0;
}
```

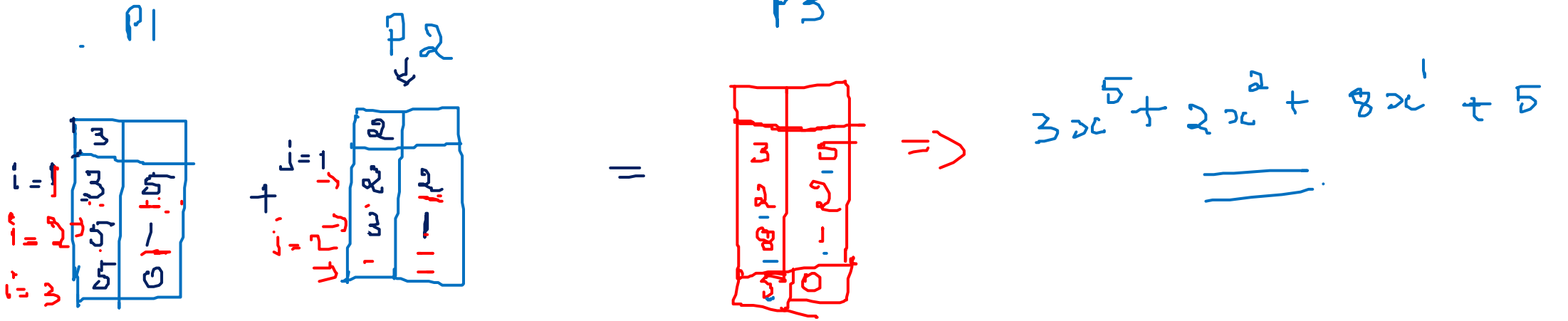
```
void getpoly(poly p[])
{
	int n,i;
	printf("Enter the number of terms: ");
	scanf("%d",&n);
	printf("Enter the terms (coefficient, exponent):\n");
	p[0].coeff=n;
	for(i=1;i<=n;i++)
		scanf("%d %d",&p[i].coeff,&p[i].exp);
}
void showpoly(poly p[])
{
	int i;
	for(i=1;i<=p[0].coeff;i++)
		printf("(%d,%d) ",p[i].coeff,p[i].exp);
}
```

$P_1 = 3x^5 + 5x^1 + 5$

$P_2 = 2x^2 + 3x^1$

P3 = result

i & j



5 > 2 ✓ => increment i

1 ≠ 2 => 4 j.

1 = 1 => add it

5 + 3 = 8

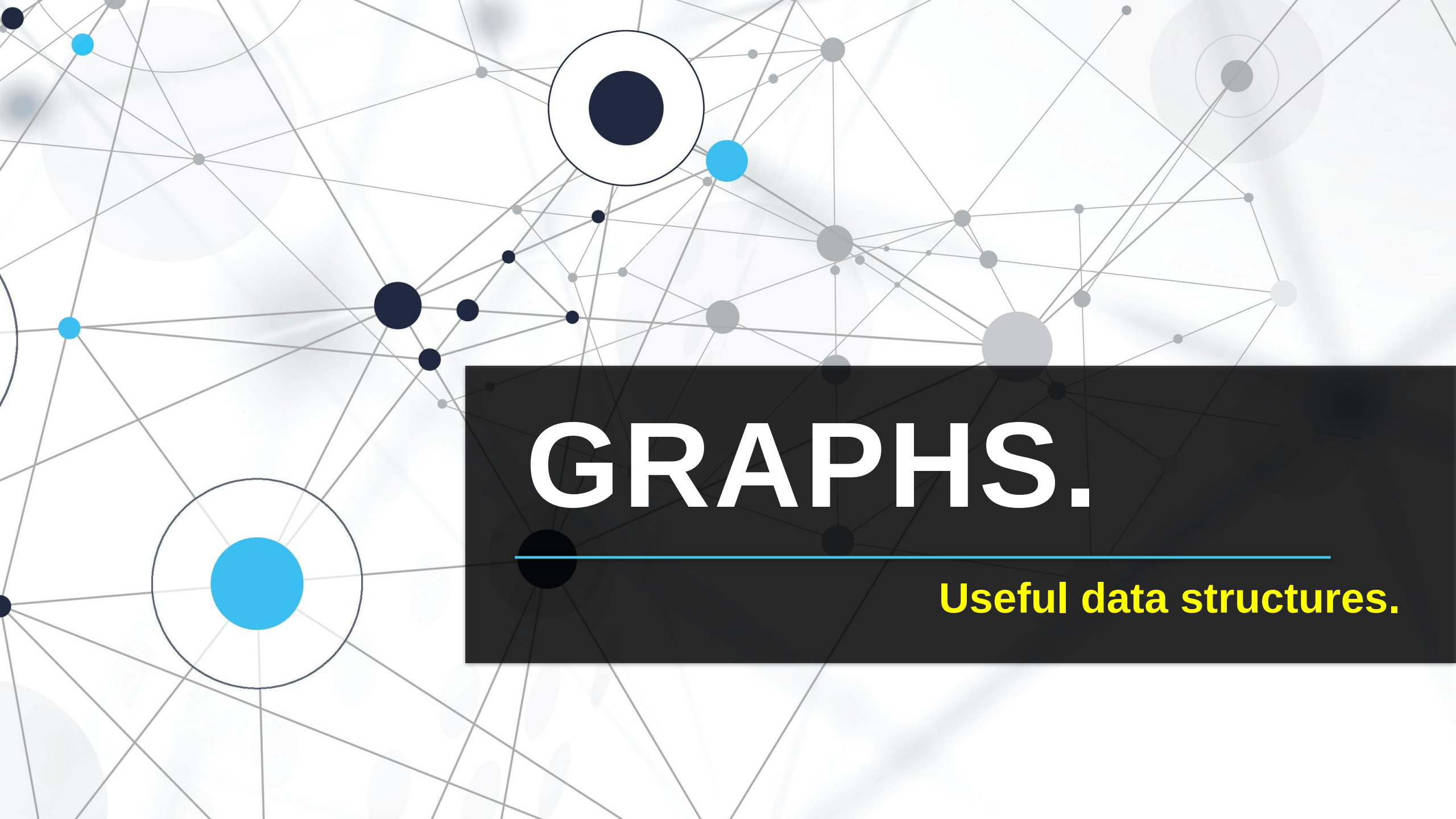
increment i & j.

```
void add(poly p1[],poly p2[],poly p3[])
{
    int i=1,j=1,n1=p1[0].coeff,
            n2=p2[0].coeff,n3=0;
      while(i<=n1 && j<=n2)
      {
            if(p1[i].exp>p2[j].exp)
            {
                  n3++;
                  p3[n3].coeff=p1[i].coeff;
                  p3[n3].exp=p1[i].exp;
                  i++;
            }
            else if(p1[i].exp<p2[j].exp)
            {
                  n3++;
                  p3[n3].coeff=p2[j].coeff;
                  p3[n3].exp=p2[j].exp;
                  j++;
            }
```

```
else  {
        int sum=p1[i].coeff+p2[j].coeff;
                  if(sum!=0)
                  {
                        n3++;
                        p3[n3].coeff=sum;
                        p3[n3].exp=p1[i].exp;
                  }
                  i++;
                  j++;
            }
      }
```

```
while(i<=n1)
      {
                  n3++;
            p3[n3].coeff=p1[i].coeff;
            p3[n3].exp=p1[i].exp;
            i++;
      }
      while(j<=n2)
      {
            n3++;
            p3[n3].coeff=p2[j].coeff;
            p3[n3].exp=p2[j].exp;
            j++;
      }
      p3[0].coeff=n3;
}
```

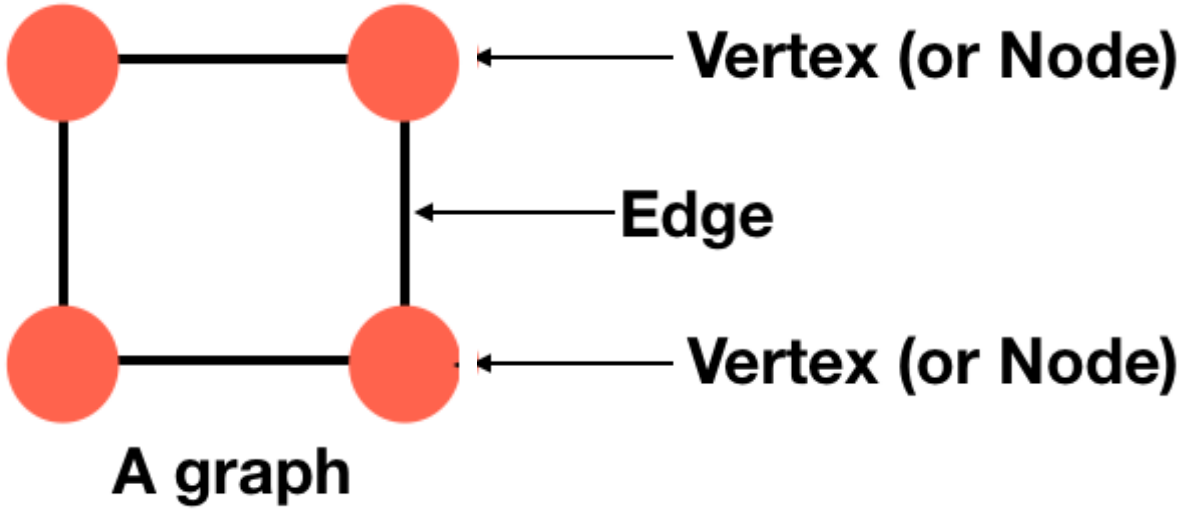
END


GRAPHS.
Useful data structures.

WHAT WE KNOW ALREADY.

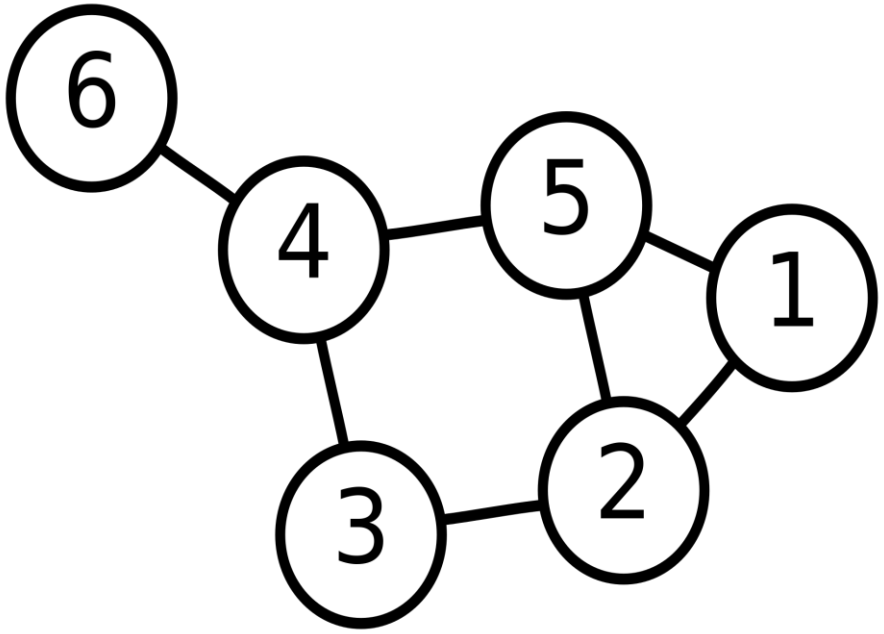
- Need for ADTs and data structures:
  - Regular Arrays (with dynamic sizing)
  - Linked Lists
  - Stacks, Queues
  - Heaps
  - Unbalanced and Balanced Search Trees
- Some algorithms like Tree traversals

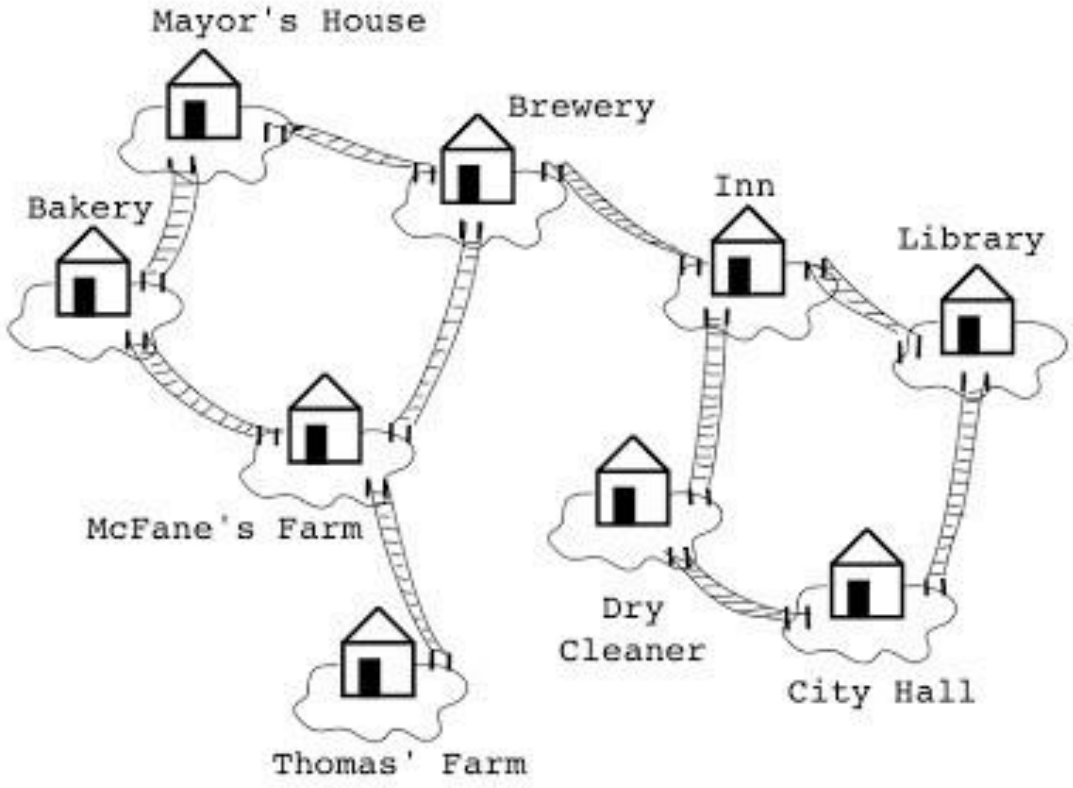
BASIC CONCEPTS: GRAPHS.



A graph is a **data structure** that consists of a **set of nodes or vertices** and a **set of edges** that *relate the nodes to each other.*

BASIC CONCEPTS: GRAPHS.





The **set of edges describes relationships among the nodes.**

MATHEMATICAL OR FORMAL DEFINITION.

- A graph G is defined as follows:

$$G=(V,E)$$

V(G): a finite, nonempty set of vertices $V = \{v_1, v_2, \ldots, v_n\}$

E(G): a set of edges (pairs of vertices) $E = \{e_1, e_2, \ldots, e_m\}$

An edge "connects" the vertices

Graphs can be directed or undirected

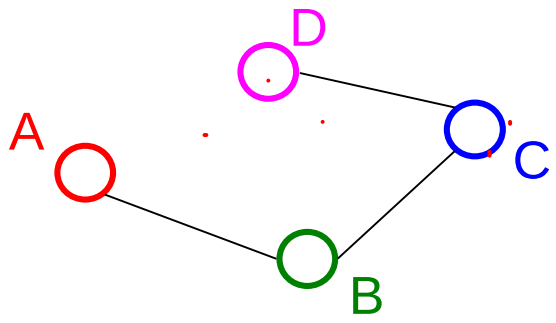
MATHEMATICAL OR FORMAL DEFINITION.

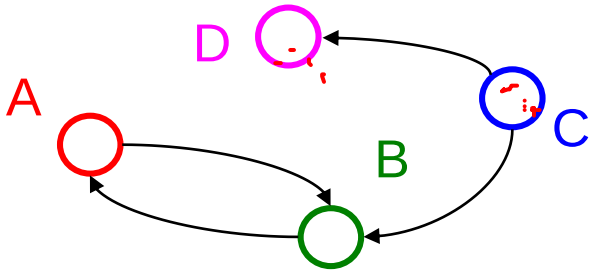
- **Undirected graphs:** edges have no specific direction.
- Edges are always "two-way“
- Thus, (u, v) ∈ E implies (v, u) ∈ E.
- Only one of these edges needs to be in the set, the other is implicit.

- **Directed graphs or digraphs:** edges have direction
- Thus, (u, v) ∈ E does not imply (v, u) ∈ E.
- Let (u, v) ∈ E mean u → v

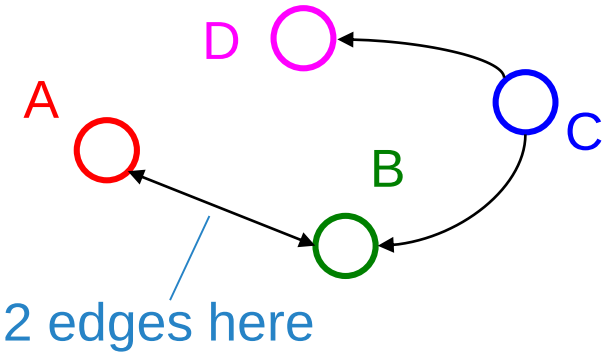
  Where u is the source, and

  v the destination



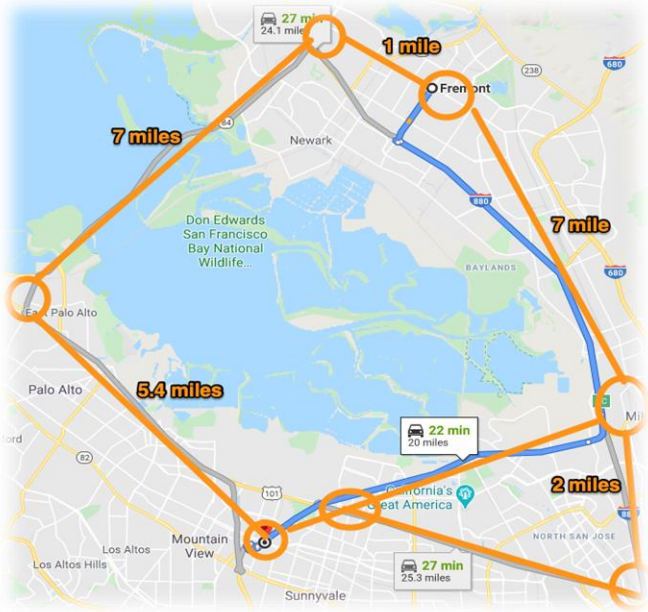


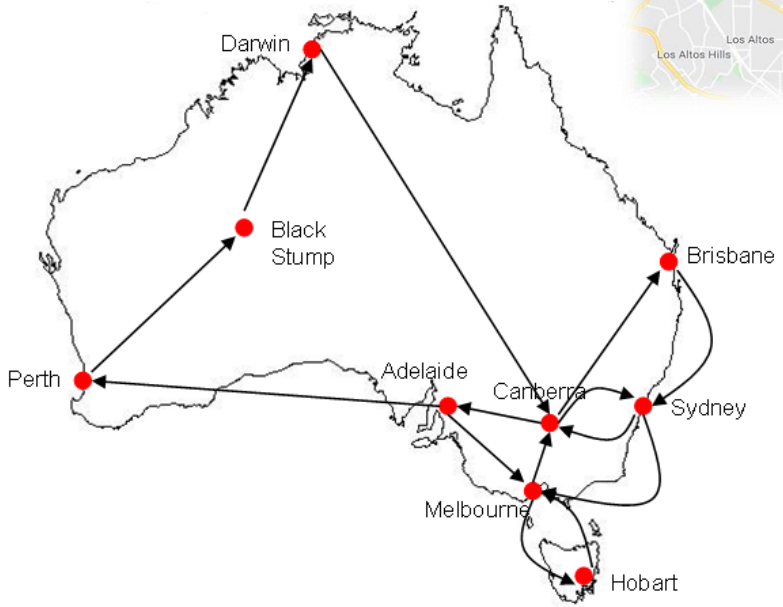
or



APPLICATIONS OF GRAPHS.

- Web pages with links
- Facebook friends
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees



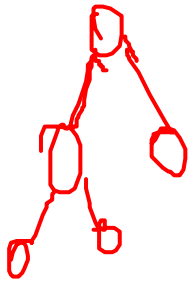


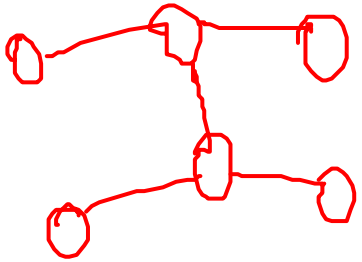
APPLICATIONS OF GRAPHS.

Graphs can be used to ***solve complex routing problems***, such as designing and routing airlines among the airports they serve.

Similarly, graphs can be used to route messages over a computer network from one node to another.

TREES & GRAPHS





In a non-linear list, each element can have more than one successor.

In a tree, an element can have only one predecessor.

In a graph, an element can have one or more predecessors.

One final point: ***A tree is a graph*** in which each vertex has only one predecessor; however, ***a graph is not a tree.***

BASIC CONCEPTS IN COMPUTER SCIENCE.

A graph is a ***collection of nodes called vertices***, and a ***collection of segments called lines***, ***connecting pairs of vertices***.

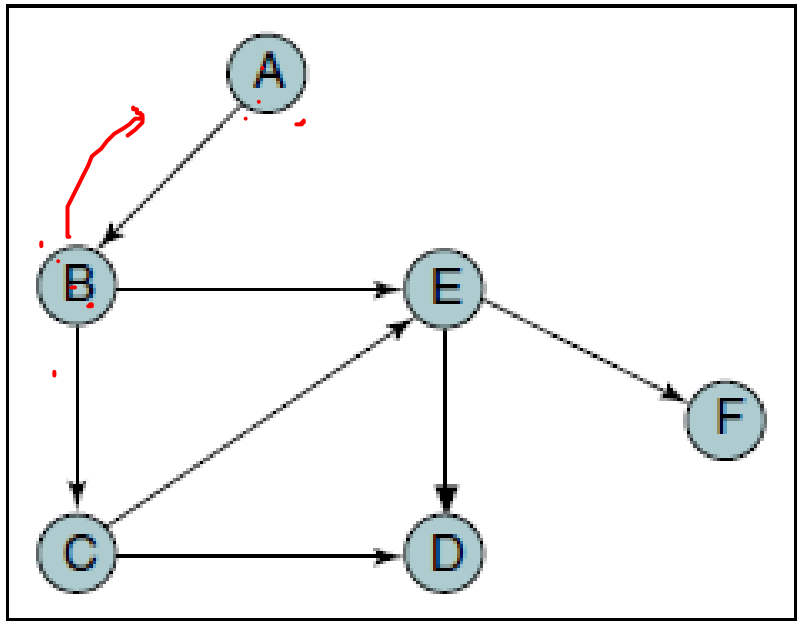


FIGURE 11-1 (a) Directed graph

So, a graph consists of two sets, a set of vertices and a set of lines *(we know that)*

A directed graph, or digraph for short, is a graph in which each line has a direction (arrow head) to its successor.

The **lines** in a directed graph are known as **arcs.** In a directed graph, the flow along the arcs between two vertices can follow only the indicated direction.

BASIC CONCEPTS.

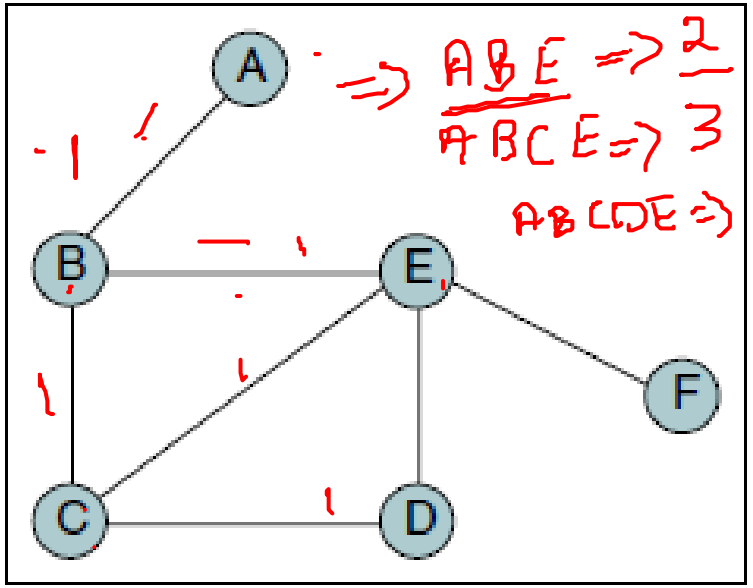


FIGURE 11-1 (b) Undirected graph

An undirected graph is a graph in which there is no direction (arrow head) on any of the lines, which are known as edges.

In an undirected graph, the flow between two vertices can go in either direction.

A path **is a sequence of vertices** in which each vertex is adjacent to the next one.

{A, B, C, E} is one path and {A, B, E, F} is another.

Two vertices in a graph are said to be adjacent vertices (or neighbors) if there is a path of length 1 connecting them.

In (a), B is adjacent to A, whereas E is not adjacent to D; on the other hand, D is adjacent to E

In (b), E and D are adjacent, but D and F are not.

CYCLES AND LOOPS.

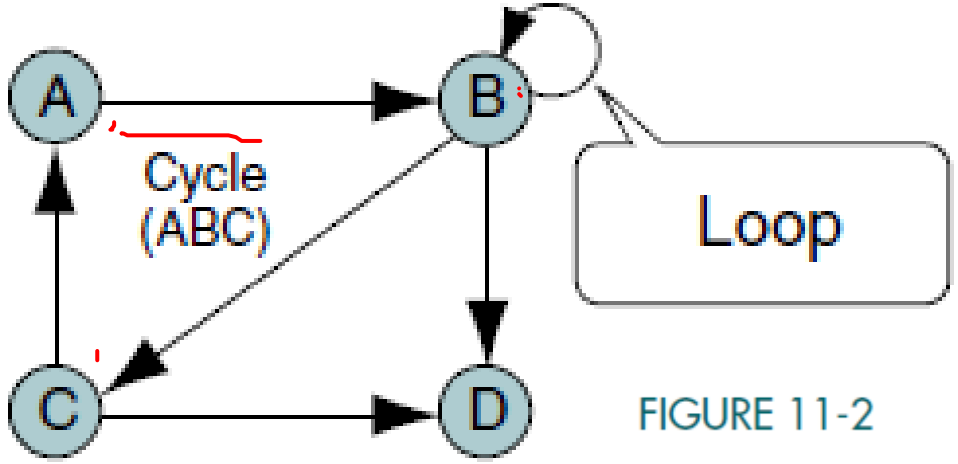


FIGURE 11-2

In Figure 11-1(a), ***same vertices do not constitute a cycle*** because in a digraph a path can follow only the direction of the arc, whereas in an undirected graph a path can move in either direction along the edge.

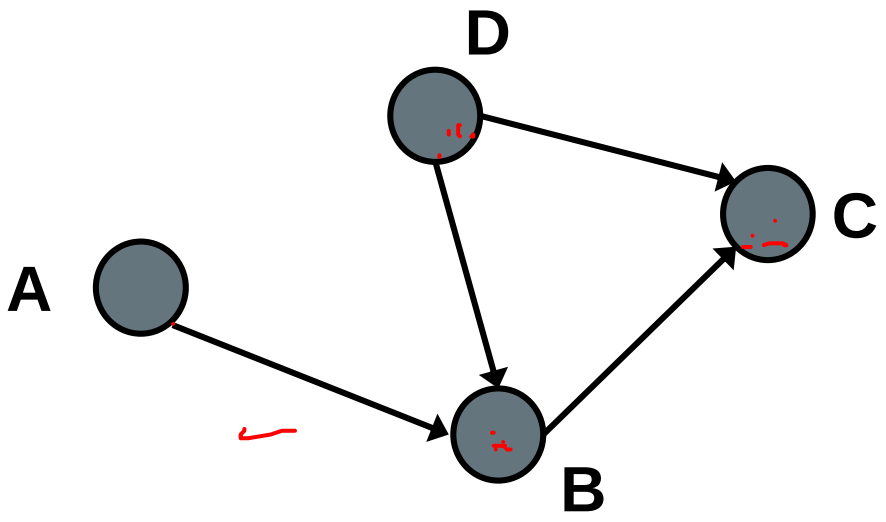
A ***cycle*** is a path consisting of at least three vertices that starts and ends with the same vertex.

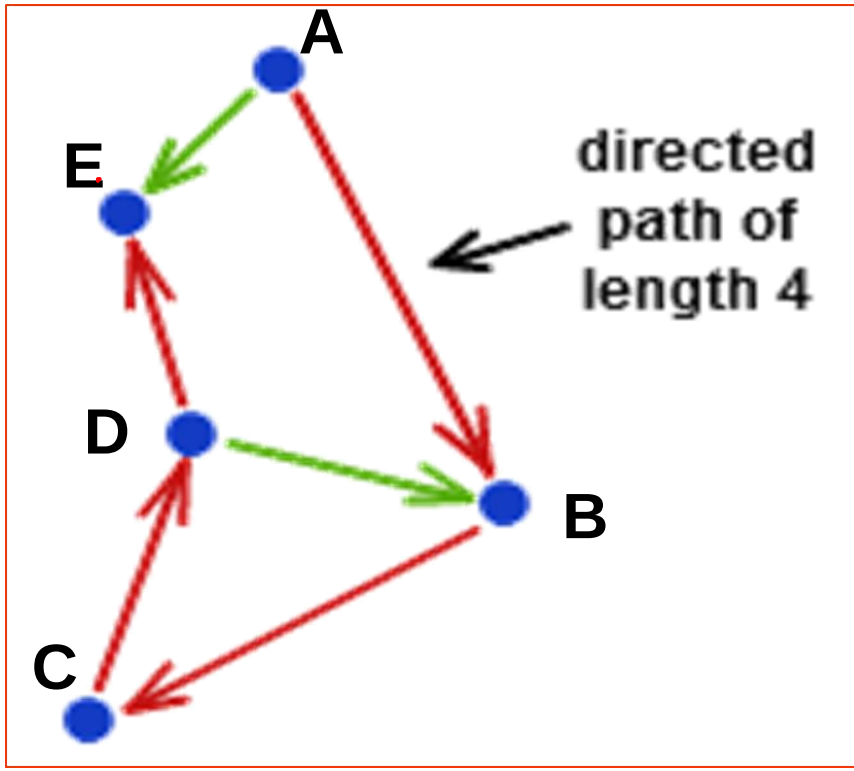
In Fig.11-1(b) : B, C, D, E, B is a cycle

A ***loop*** is a special case of a cycle in which a single arc begins and ends with the same vertex. In a loop the end points of the line are the same.

Length: Length of a path is the **number of edges** in the path

PATHS AND CYCLES IN DIRECTED GRAPHS

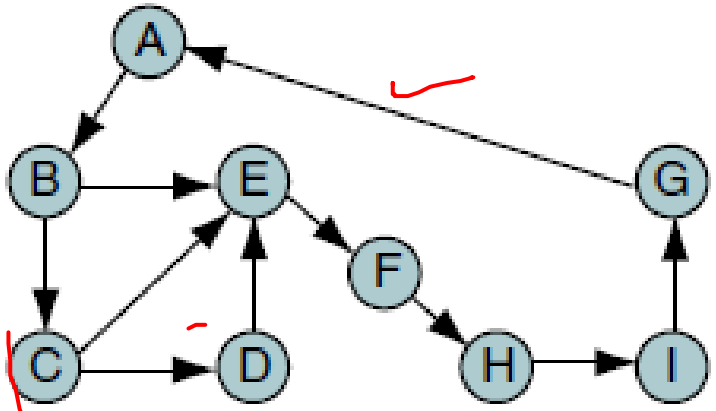




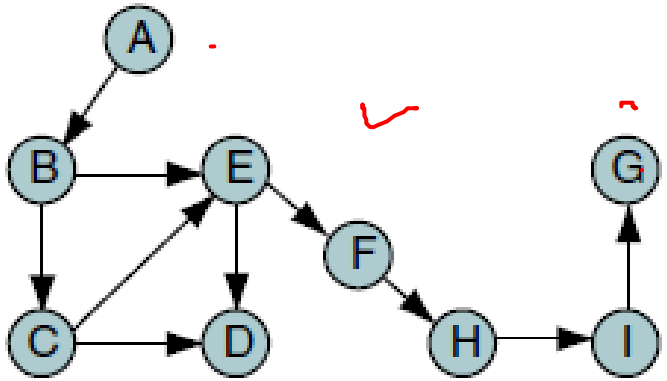
Is there a path from A to D? **No**

Does the graph contain any cycles? **No**

CONNECTED GRAPHS.



(b) Strongly connected



(a) Weakly connected

FIGURE 11-3

Two vertices are said to be ***connected*** if there is a **path between them**.

A graph is said to be **connected** if, ignoring direction, there is a **path from any vertex to any other vertex.**

A **directed graph** is **strongly connected** if there is a path from each vertex to every other vertex in the digraph.

A **directed graph** is **weakly connected** if at least two vertices are not connected.

A connected undirected graph would always be strongly connected, so the concept is not normally used with undirected graphs.

DISJOINT GRAPHS.

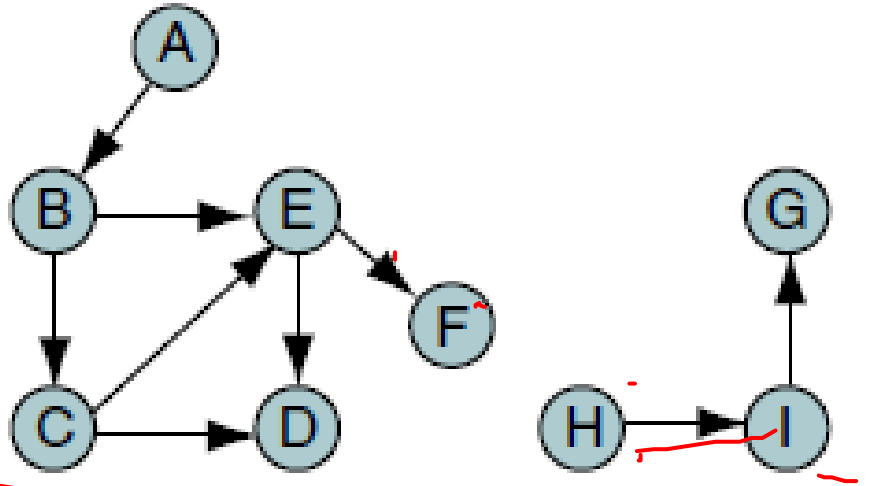


FIGURE 11-3 (c) Disjoint graph

A graph is a disjoint graph if it is not connected.

DEGREE OF A VERTEX.

Number of lines incident to it.

The **indegree** is the number of arcs entering the vertex.

+

The **outdegree** of a vertex in a digraph is the number of arcs leaving the vertex.

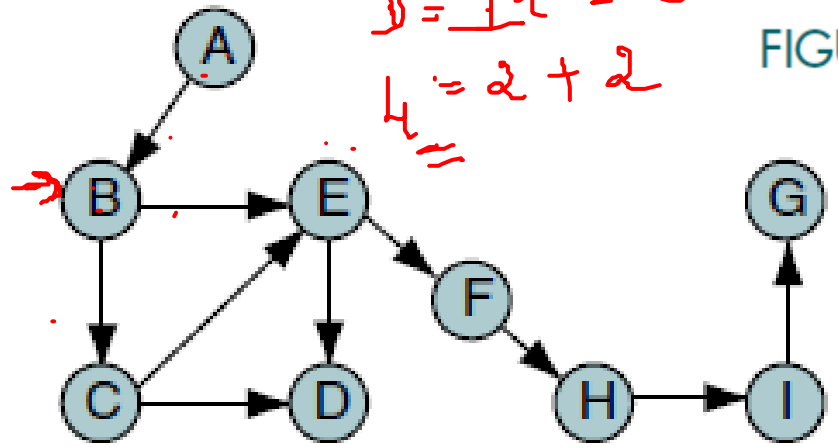


FIGURE 11-3

(a) Weakly connected

The degree of vertex B is 3

The degree of vertex E is 4.

The indegree of vertex B is 1 and its outdegree is 2

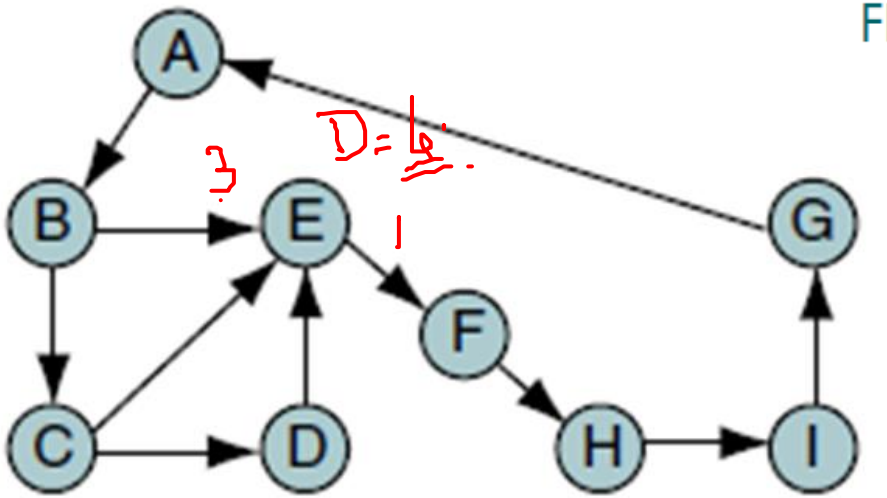
DEGREE OF A VERTEX.

Number of lines incident to it.

The indegree is the number of arcs entering the vertex.

The outdegree of a vertex in a digraph is the number of arcs leaving the vertex.

FIGURE 11-3

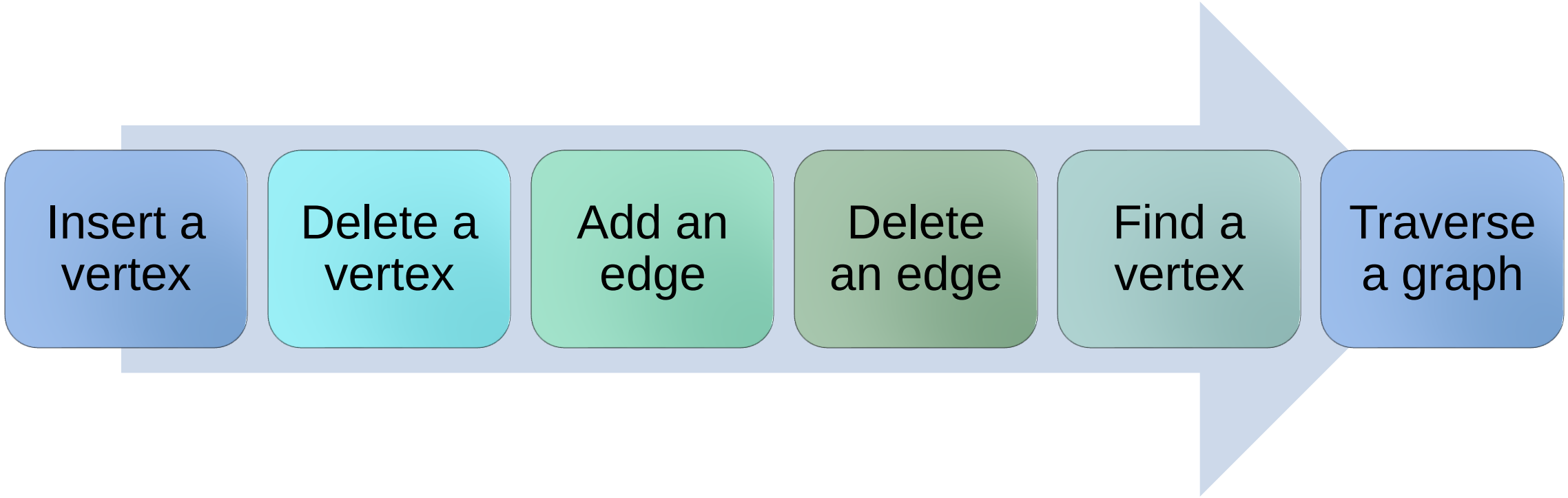


(b) Strongly connected

The indegree of vertex E is 3 and its outdegree is 1.

OPERATIONS.

There are six primitive graph operations that provide the basic modules needed to maintain a graph:



INSERT VERTEX.

Adds a new vertex to a graph.

When a vertex is inserted, it is disjoint. That is, it is not connected to any other vertices in the list.

Inserting a vertex is just the first step in the insertion process.

After a vertex is inserted, it must be connected.

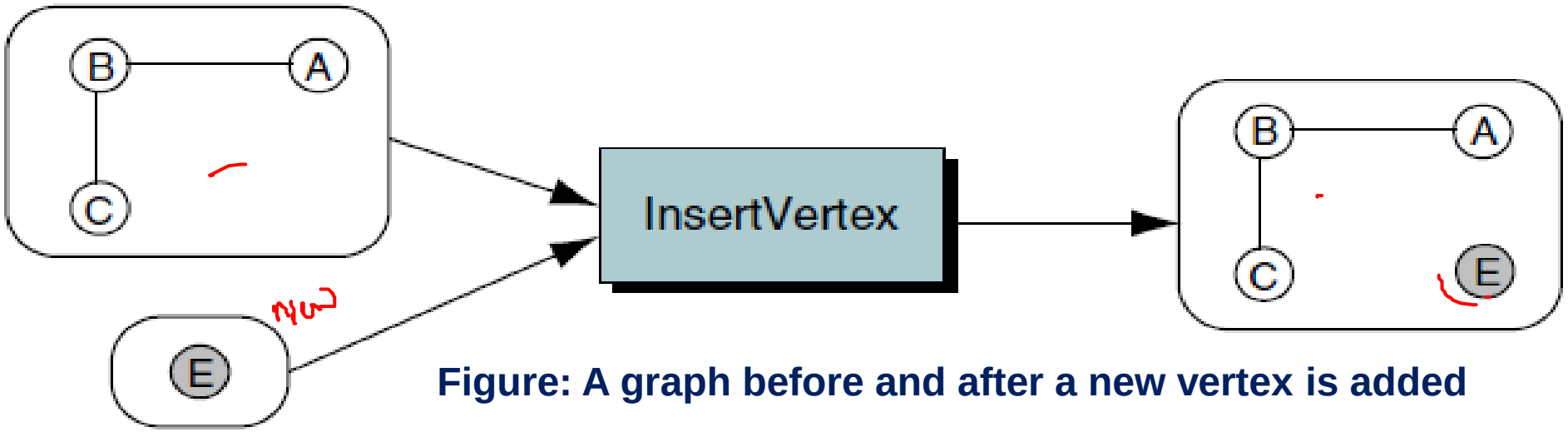
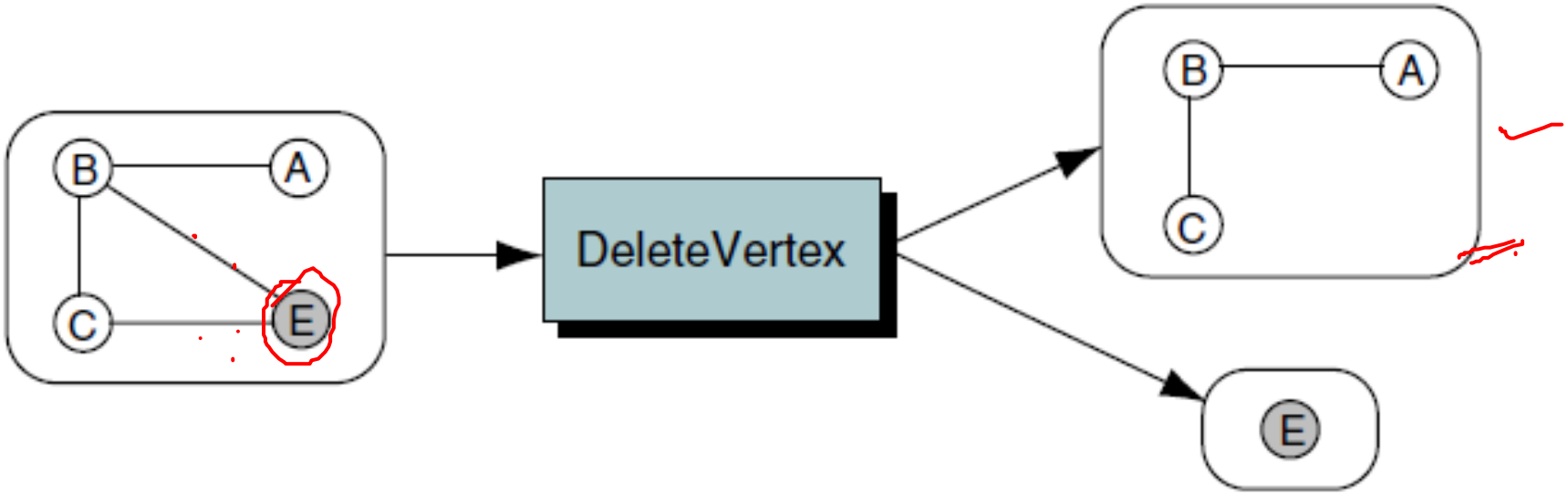


Figure: A graph before and after a new vertex is added

DELETE VERTEX.



Delete vertex removes a vertex from the graph.

When a vertex is deleted, all connecting edges are also removed.

ADD EDGE.

Add edge connects a vertex to a destination vertex.

If a vertex requires multiple edges, add an edge must be called once for each adjacent vertex.

To add an edge, two vertices must be specified.

If the graph is a digraph, one of the vertices must be specified as the source and one as the destination.

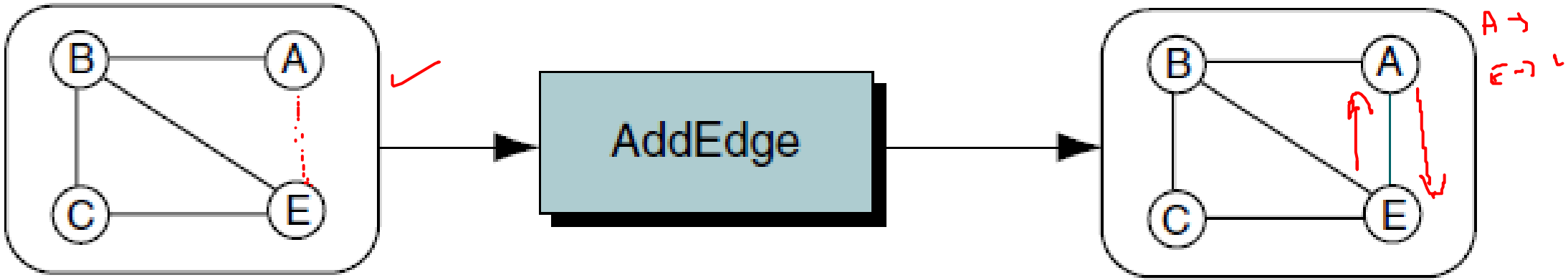


Figure: Adding an edge, {A, E}, to the graph.

DELETE EDGE.

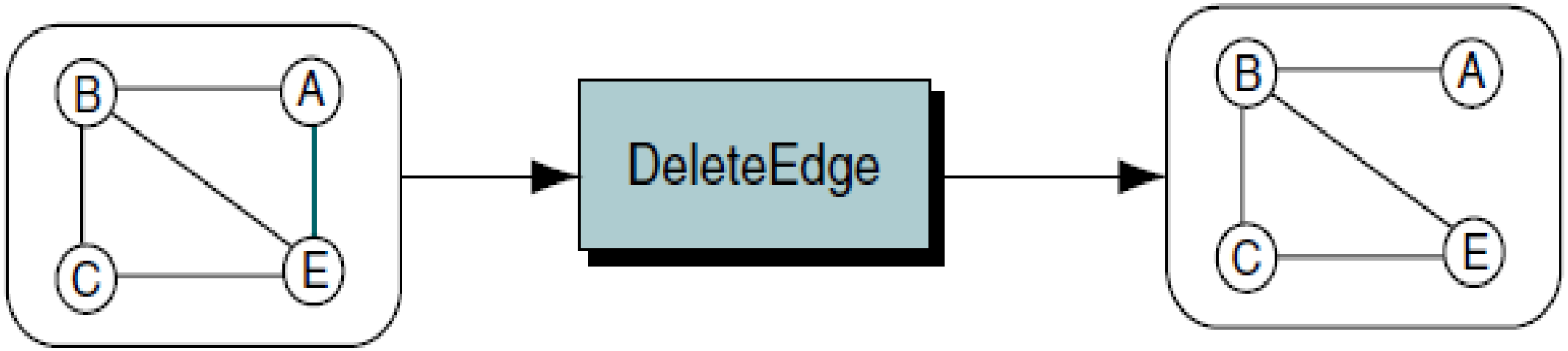


Figure: Deleting the edge, {A, E}, to the graph.

Delete edge removes one edge from a graph.

FIND VERTEX.

Find vertex traverses a graph, looking for a specified vertex.

If the vertex is found, its data are returned. If it is not found, an error is indicated.

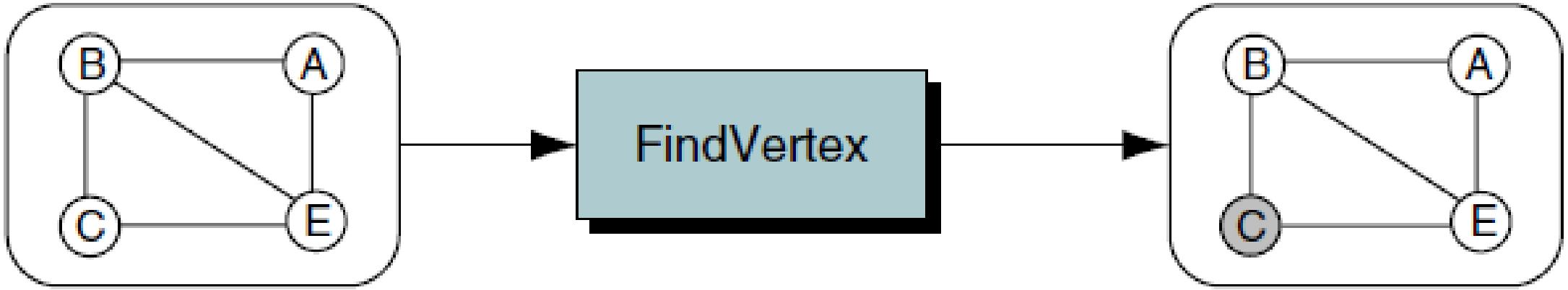
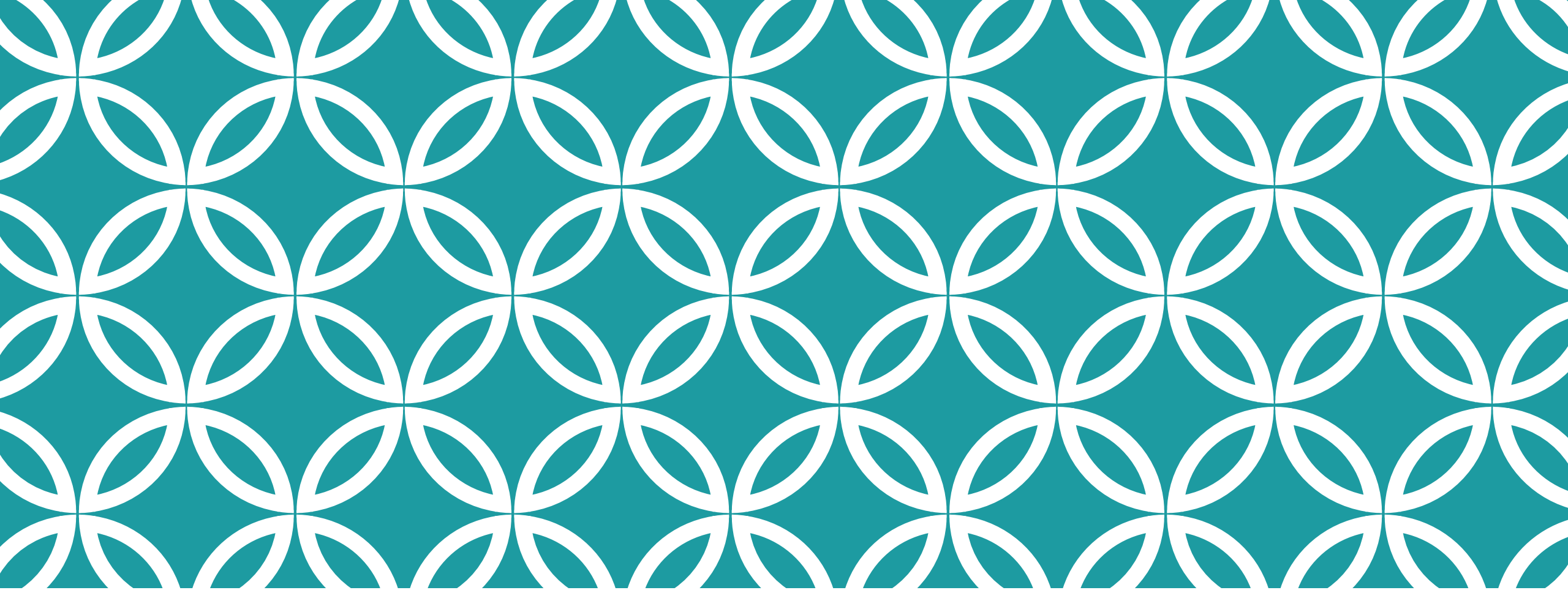


Figure: Find vertex traverses the graph, looking for vertex C..

END.



GRAPH STORAGE STRUCTURES.

Ref. Books: Corman & Forouzan

REPRESENTATIONS OF GRAPHS.

Choice between two standard ways to represent a graph **G = (V, E):**

- **As an adjacency matrix, or**
- **As a collection of adjacency lists.**

Either way applies to both directed and undirected graphs.

Because the adjacency-list representation provides a compact way to represent **sparse graphs**, those for which **|E| is much less than $|V|^2$** , it is usually the method of choice.

An adjacency-matrix representation is chosen, however, when **the graph is dense**, **|E| is close to $|V|^2$** or when we need to be able to tell quickly if there is an **edge connecting two given vertices**.

GRAPH STORAGE STRUCTURES.

To represent a graph, need to store ***two sets.***

- First set represents the **vertices of the graph**, and
- Second set represents the **edges or arcs**

The two most common structures used to store these sets are arrays and linked lists.

This is a major limitation.

- Although the arrays offer some simplicity and processing efficiencies, the number of vertices must be known in advance.
- Only one edge can be stored between any two vertices.

1. ADJACENCY MATRIX.

The adjacency matrix uses:

- ***A vector*** (**one-dimensional array**) for the ***vertices,*** and
- ***A matrix*** (**two-dimensional array**) to store the ***edges***.

If two vertices are adjacent, that is, if there is **an edge** between them, the intersect has a **value of 1**

If there is **no edge** between them, the intersect is **set to 0**.

If the graph is directed, the ***intersection*** in the adjacency matrix indicates the ***direction***.

1. ADJACENCY MATRIX.

For an adjacency matrix representation of a graph **G = (V,E),** we assume that the vertices are numbered 1,2, ….. |V| in some arbitrary manner.

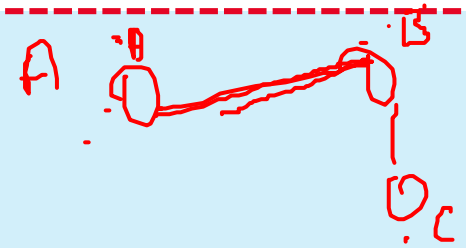
Then the adjacency matrix representation of a graph G,

- Consists of |V| * |V| matrix A = $(a_{i\ j})$ such that,

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \text{ ;} \\ 0 & \text{otherwise} \end{cases}$$

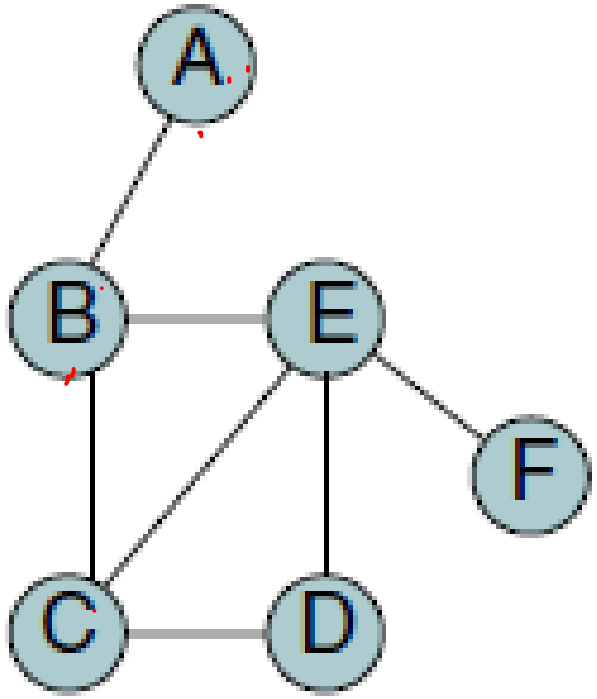
1. ADJACENCY MATRIX.

Number of 1's in the adjacency matrix:



- **Undirected graph:** Sum of degrees of all vertices
- **Directed graph:** Sum of outdegrees of all vertices

ADJACENCY MATRIX FOR UNDIRECTED GRAPH.



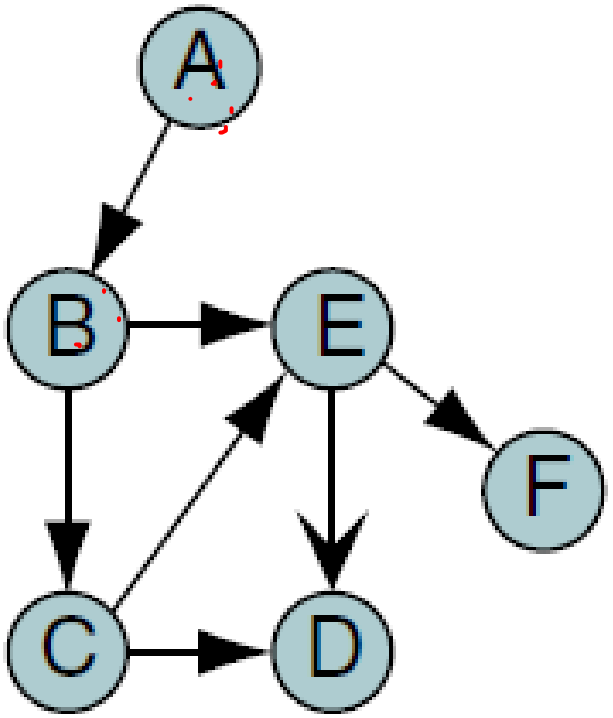
A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix for nondirected graph

ADJACENCY MATRIX FOR DIRECTED GRAPH.



A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Adjacency matrix for nondirected graph

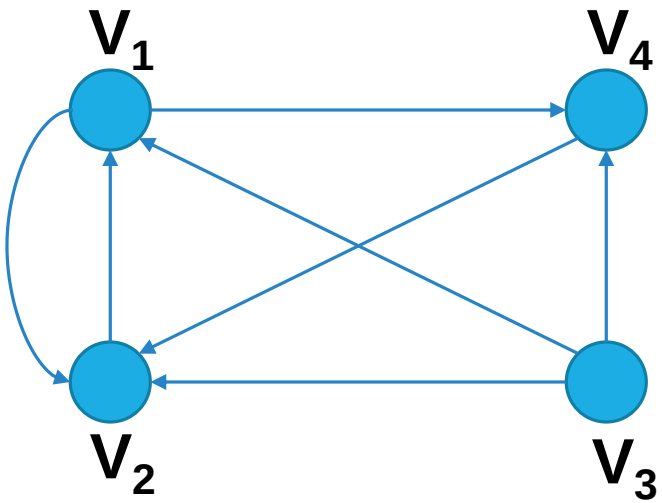
ADJACENCY MATRIX

An **element** of the adjacency matrix is either **0** or **1**

Any **matrix** whose **elements are either 0 or 1** is called **bit matrix** or **Boolean matrix**

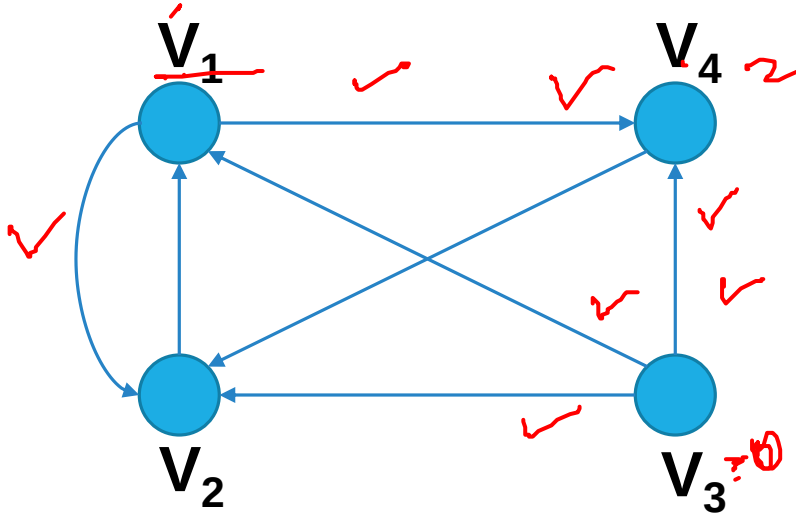
For a given graph G =m (V, E), an **adjacency matrix** depends upon the ordering of the elements of V

For different ordering of the elements of V we get different adjacency matrices.



$$A = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

ADJACENCY MATRIX



A =

	V_1	V_2	V_3	V_4
V_1	0	1	0	1
V_2	1	0	0	0
V_3	1	1	0	1
V_4	0	1	0	0

The **number of elements** in the **i^{th} row** whose **value is 1** is equal to the **out-degree** of node **V_i**

The **number of elements** in the **j^{th} column** whose **value is 1** is equal to the **in-degree** of node **V_j**

For a **NULL graph** which consist of only n nodes but no edges, the **adjacency matrix** has **all its elements 0**. i.e. the adjacency matrix is the NULL matrix

2. ADJACENCY LIST.

The adjacency ~~matrix~~ List uses:

- ***A linked list*** to store the ***vertices,*** and
- ***A two-dimensional linked list*** to store the ***arcs.***

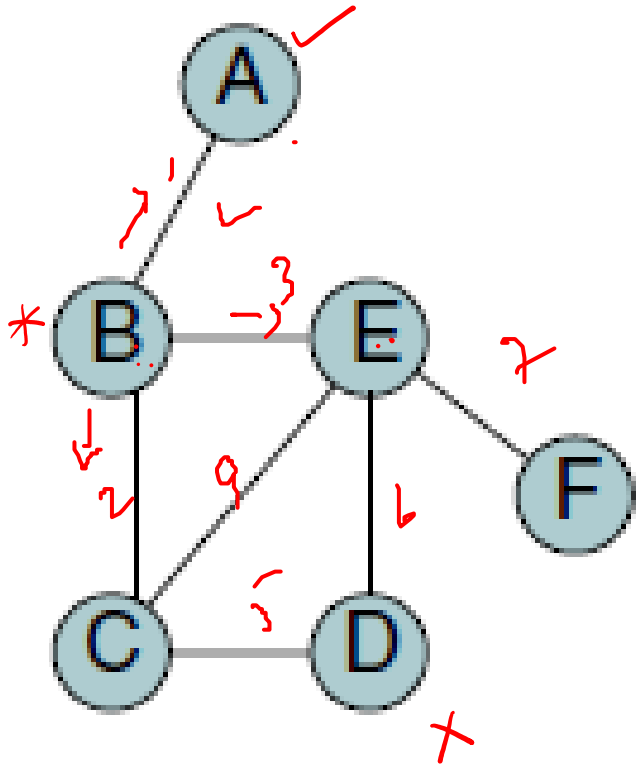
The **vertex list** is a **singly linked list** of the vertices in the list.

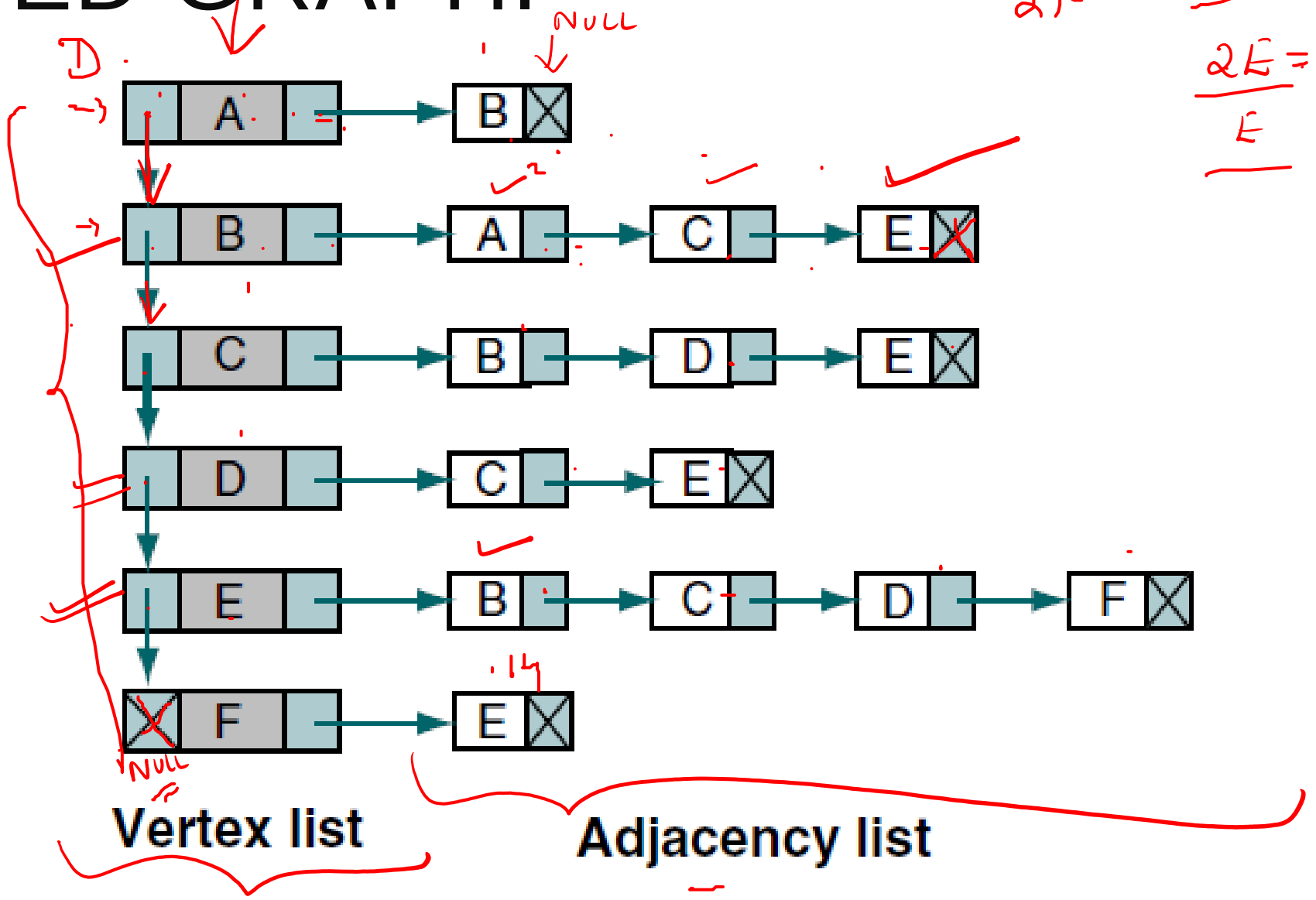
Depending on the application, it could also be implemented using doubly linked lists or circularly linked lists.

The **pointer at the left** of the list **links the vertex** entries.

The **pointer at the right** in the vertex is a **head pointer to a linked list of edges from the vertex**.

ADJACENCY MATRIX FOR UNDIRECTED GRAPH.



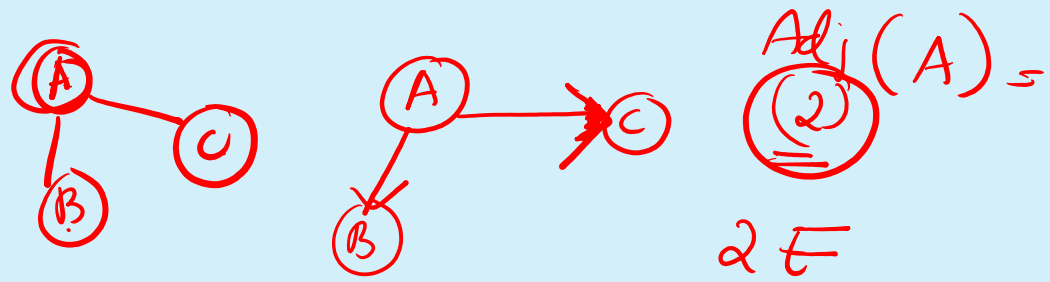


2. ADJACENCY LIST.

The adjacency-list representation of a graph **G = (V,E)** consists of an array **Adj of |V| lists**, one for each vertex in V .

For each **u** ∈ **V** , the adjacency list,

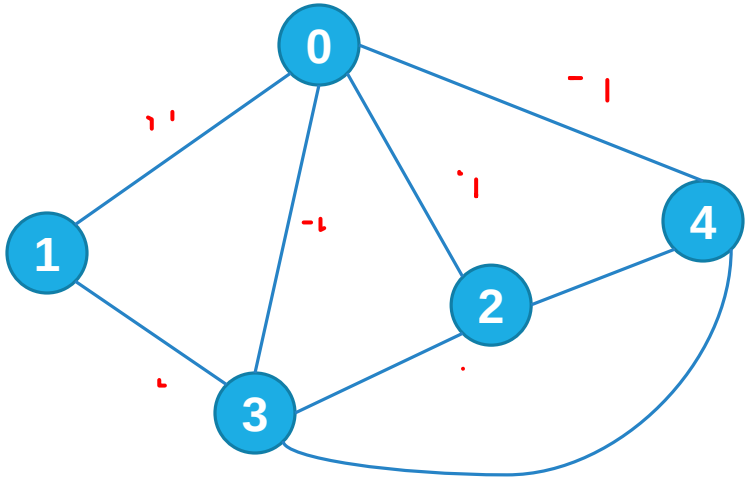
- Adj[u] contains all the vertices **v** s.t. there is an edge (u,v) ∈ E. That is, Adj[u] consists of all the vertices adjacent to u in G. (Alternatively, it may contain pointers to these vertices).

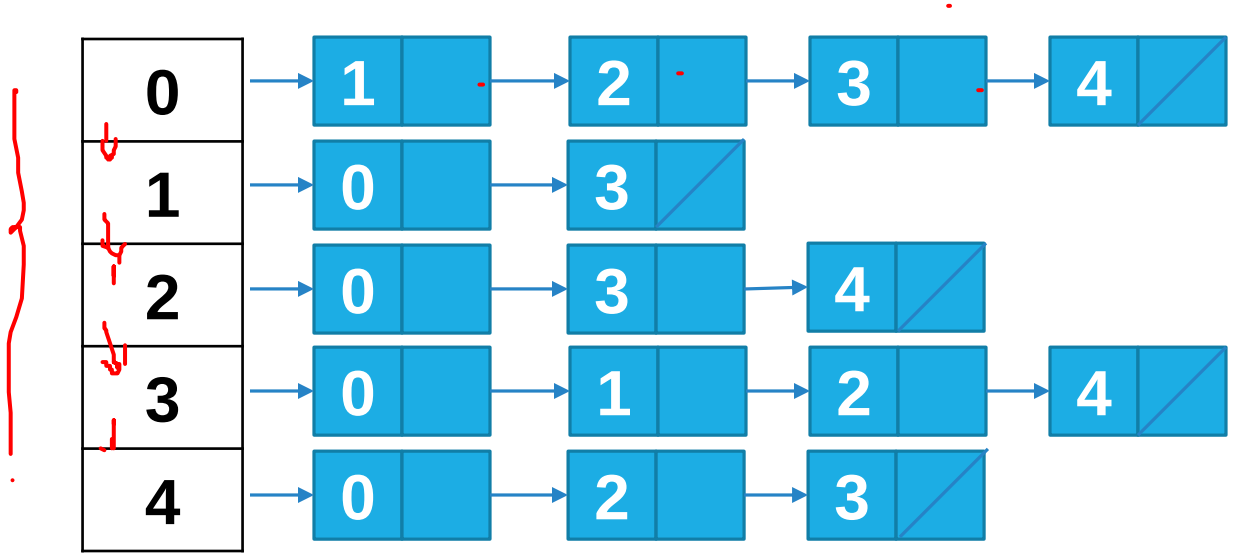


Sum of lengths of all adjacency lists

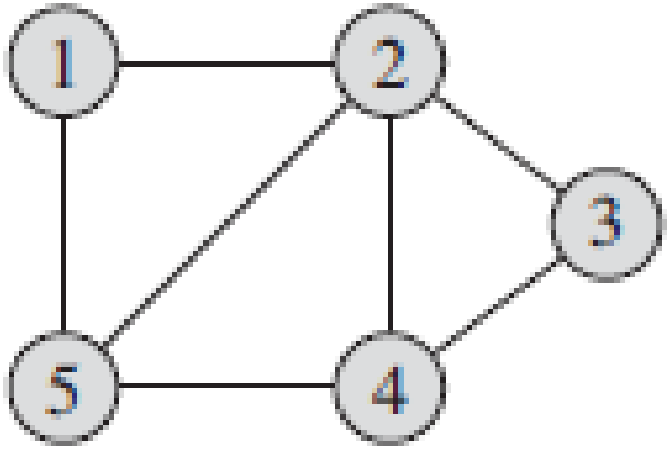
- **|E|** for **directed graph** (an edge (u, v) appears in Adj[u] and not in Adj[v])
- **2|E|** for G is **undirected** (an edge (u, v) appears in both Adj[u] and Adj[v])

ADJACENCY LIST: ONE MORE EXAMPLE.





REPRESENTATIONS FOR UNDIRECTED GRAPH: EXAMPLE 1.

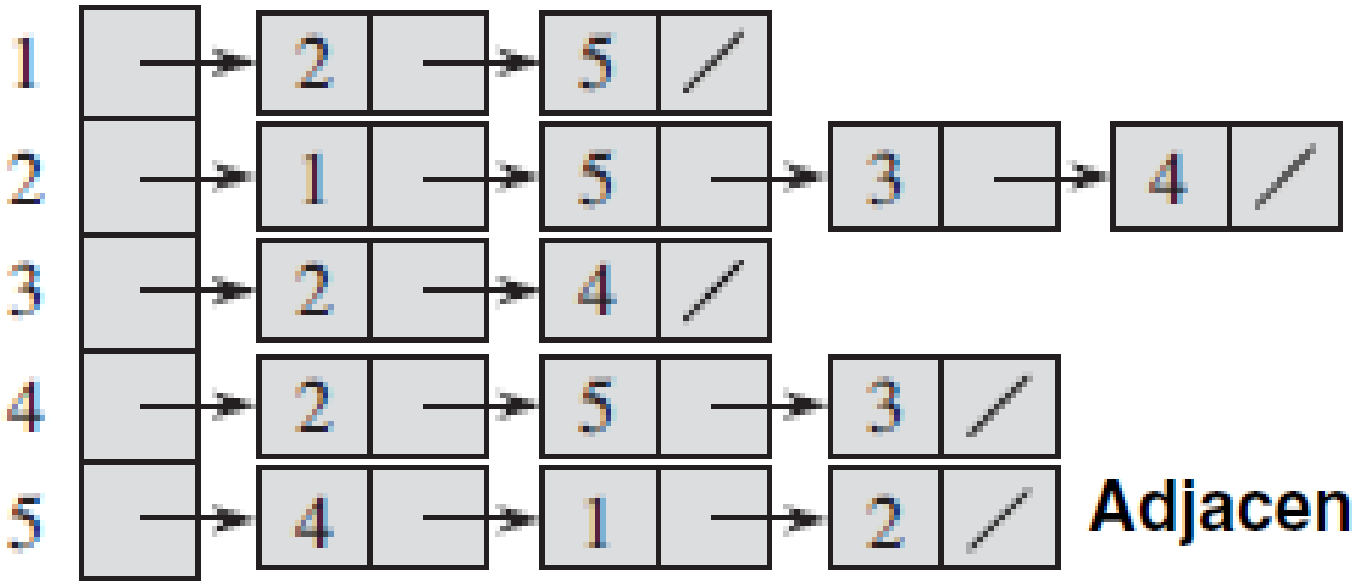


1
2
3
4
5

Vertex vector

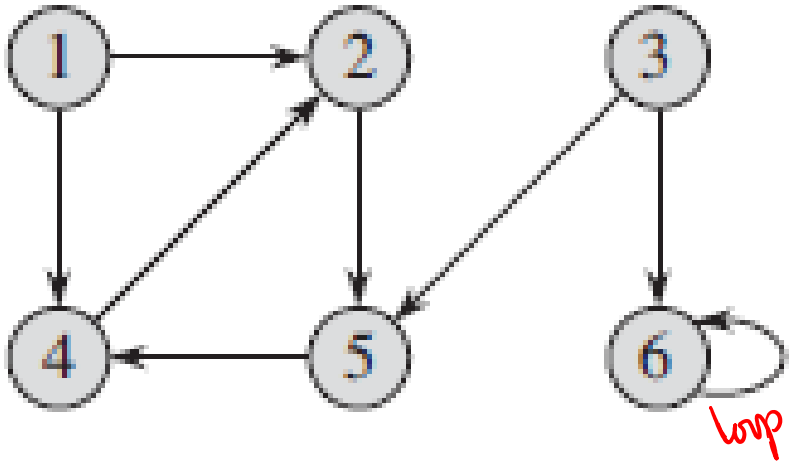
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency matrix

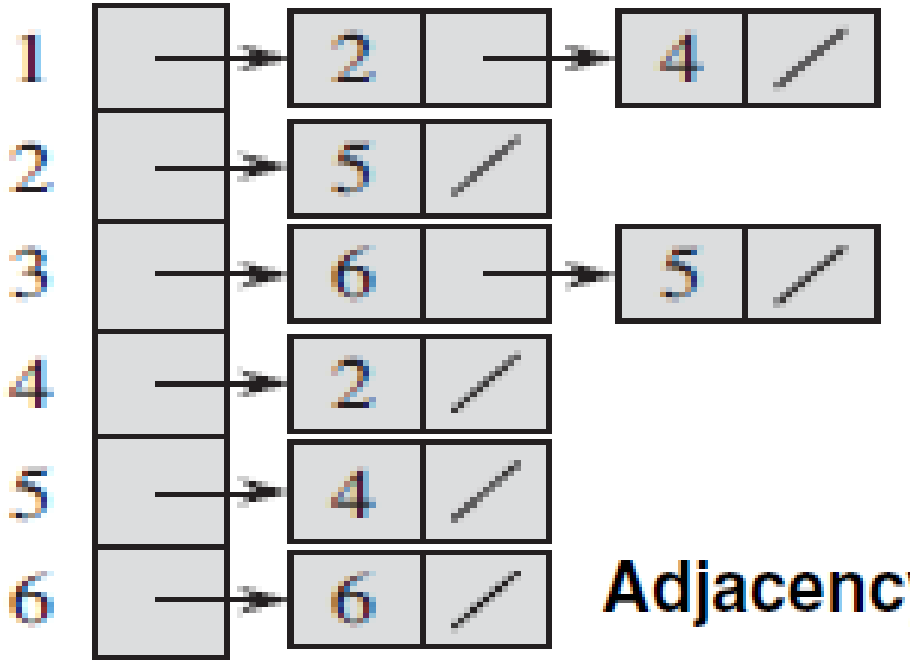


Adjacency list

REPRESENTATIONS FOR DIRECTED GRAPH: EXAMPLE 2.



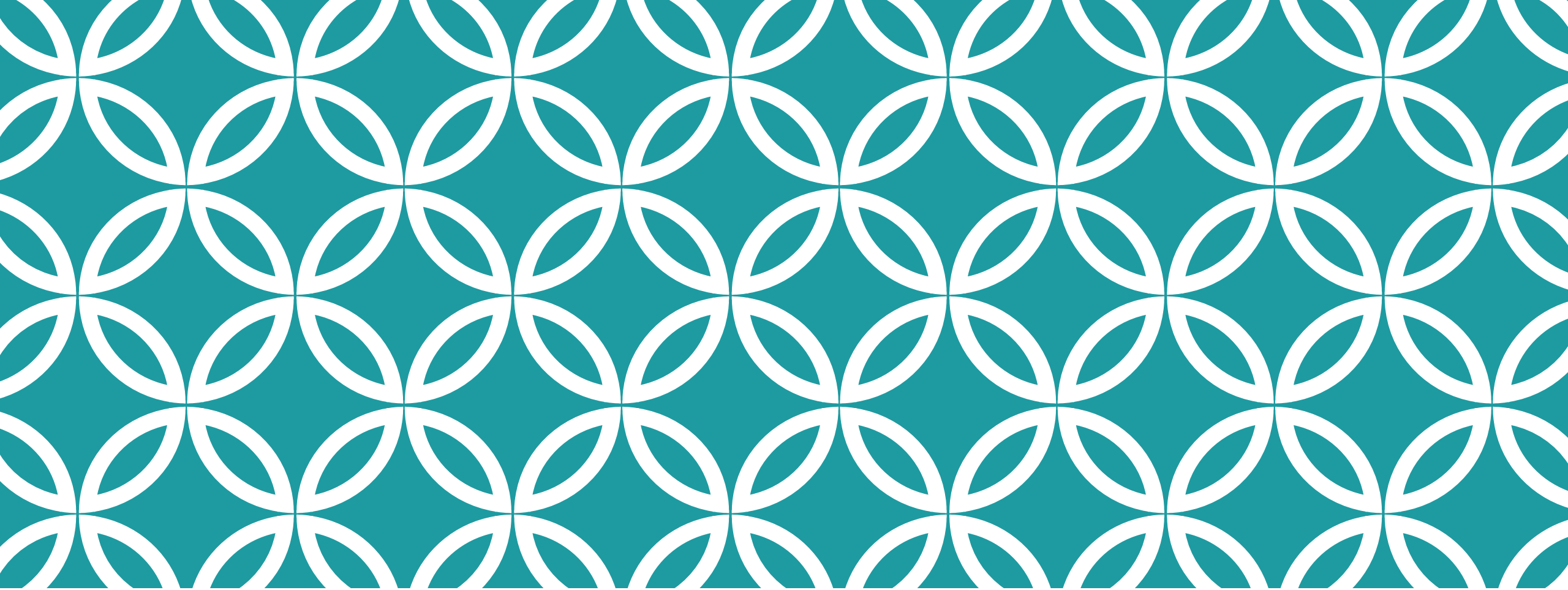
Vertex vector
1
2
3
4
5
6



Adjacency list

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency matrix



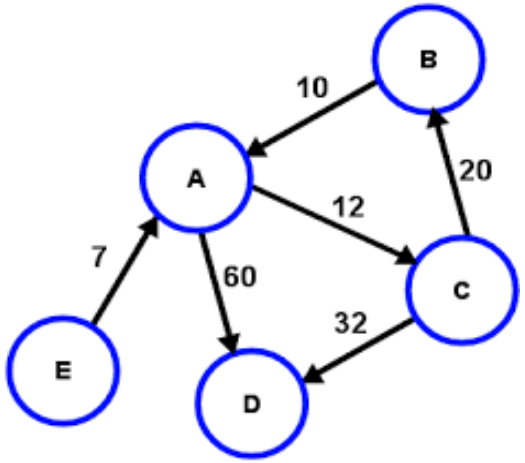
NETWORKS.

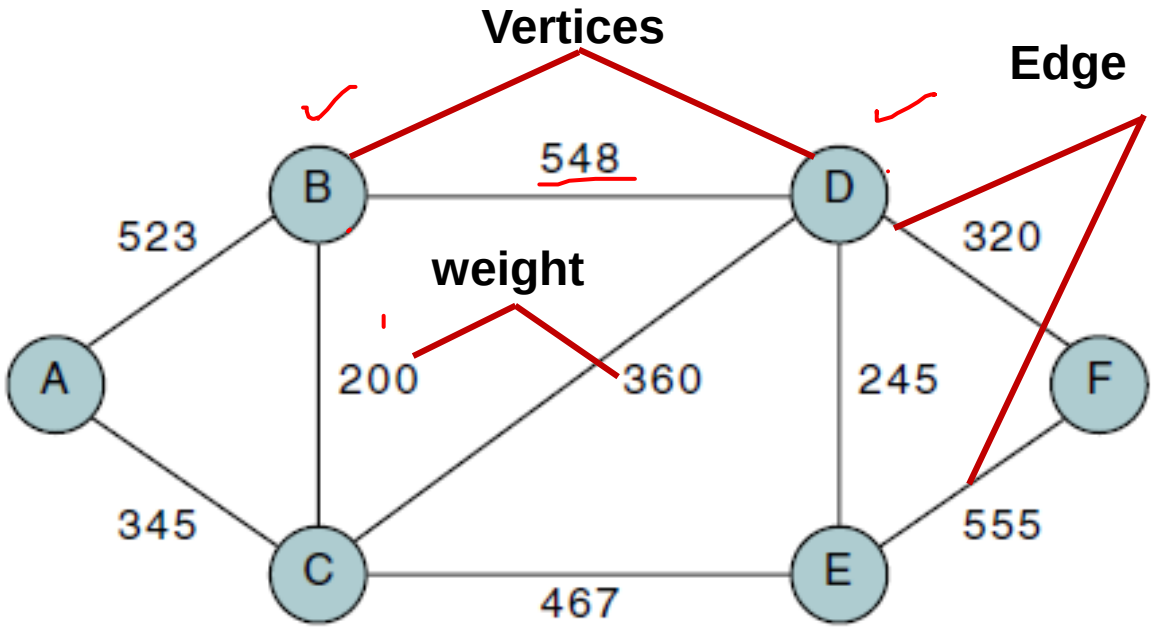
NETWORKS.

A network is a graph whose lines are weighted or called **weighted graph**.

Meaning of the weights depends on the application. For example, an airline might use a graph to represent the routes between cities that it serves. Here,

- **Vertices** represent **cities**, and
- **Edge** is **a route** between 2 cities.
- Edge's **weight** could be:
  - Miles between the 2 cities
  - Price of the flight

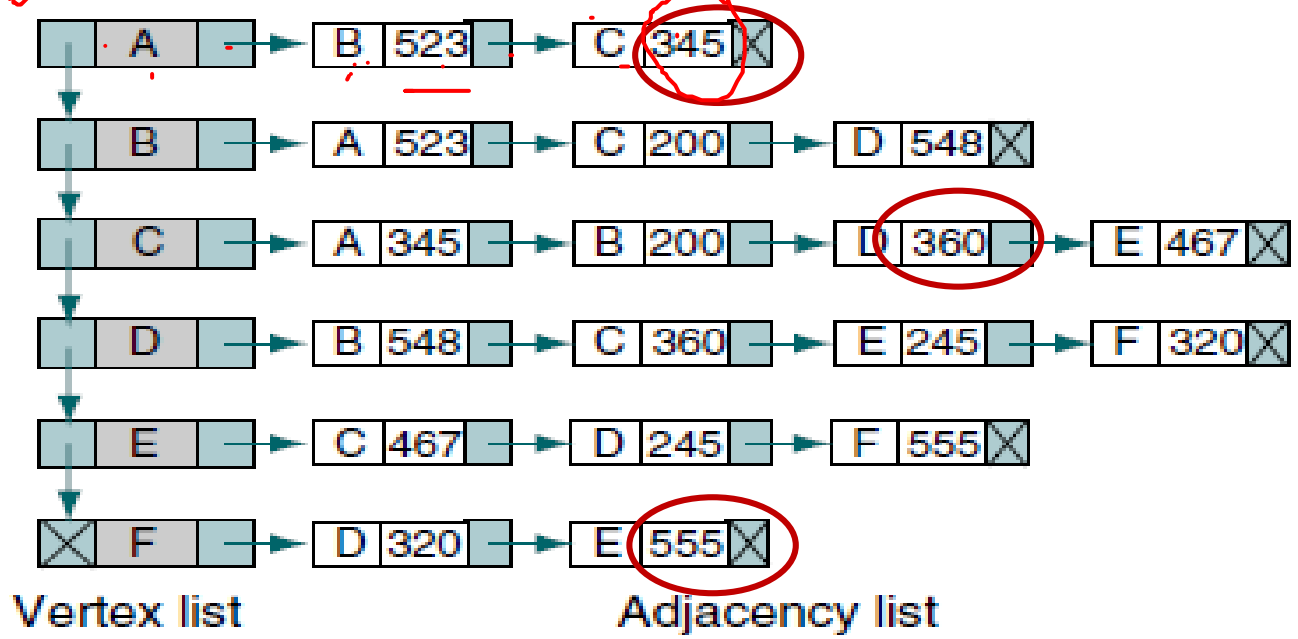


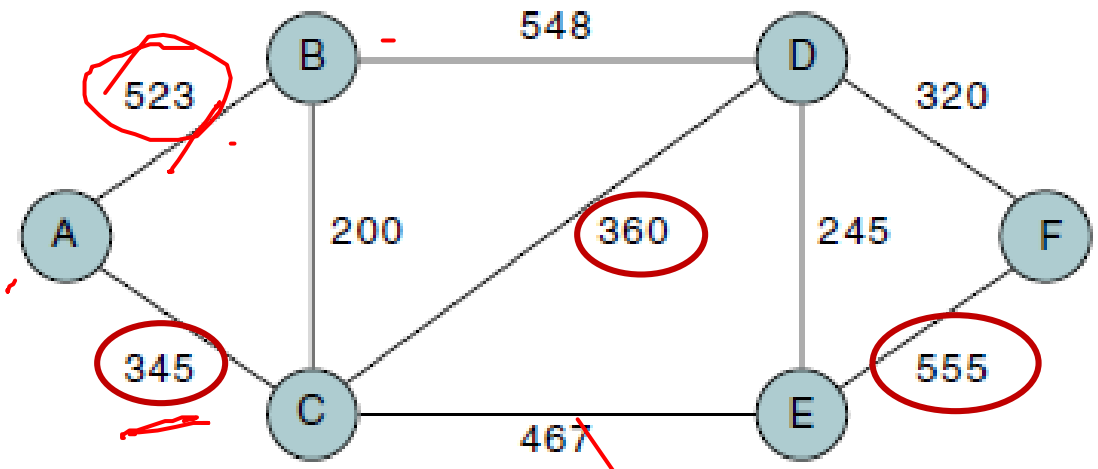


City Network

STORING WEIGHTS IN GRAPH STRUCTURES.

- Since ***weight is an attribute*** of **an edge**, it is stored in the structure that contains the edge.
- In adjacency matrix, weight is stored as the intersection value.
- In adjacency list, stored as the value in the adjacency linked list.





	A	B	C	D	E	F
A	0	523	345	0	0	0
B	523	0	200	548	0	0
C	345	200	0	360	467	0
D	0	548	360	0	245	320
E	0	0	467	245	0	555
F	0	0	0	320	555	0

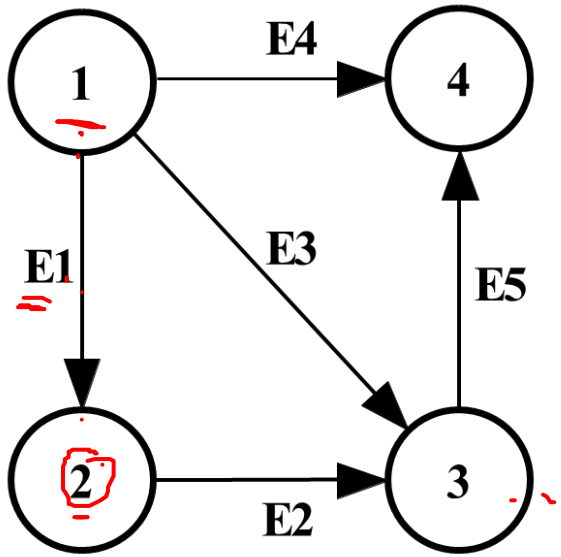
Vertex vector

Adjacency matrix

INCIDENCE MATRIX OF A DIGRAPH.

Incidence matrix is a |E| x |V| matrix B = $(b_{i\,j})$ such that,

$$b_{i\,j} = \begin{cases} -1 \text{ if edge i leaves vertex j} \\ 1 \text{ if edge i enters vertex j} \\ 0 \text{ otherwise} \end{cases}$$



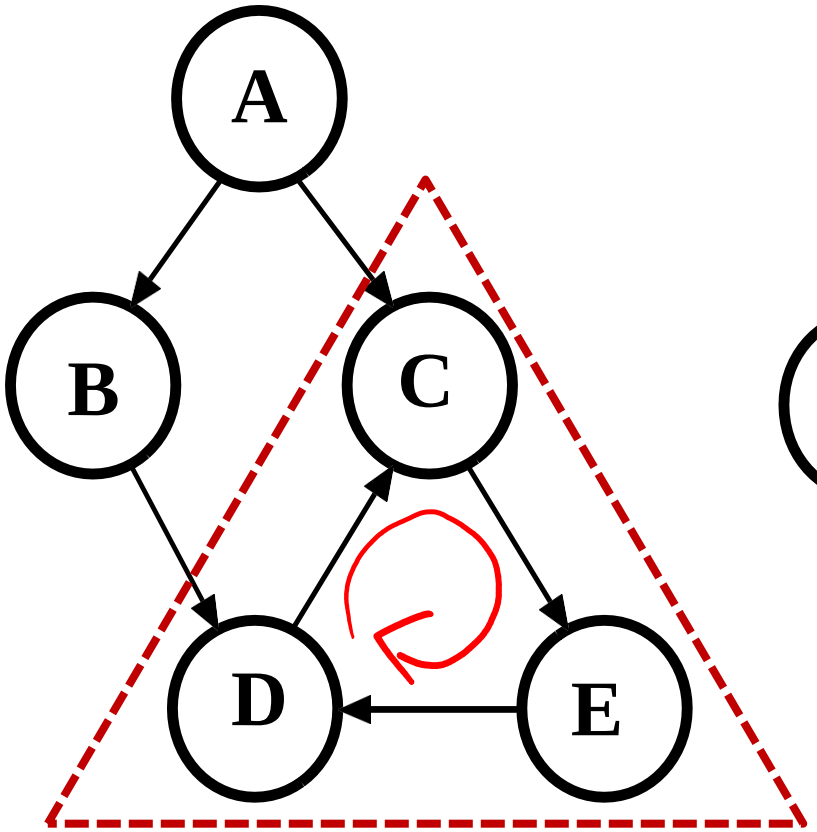
With no self loops

	1	2	3	4
E1	-1	1	0	0
E2	0	-1	1	0
E3	-1	0	1	0
E4	-1	0	0	1
E5	0	0	-1	1

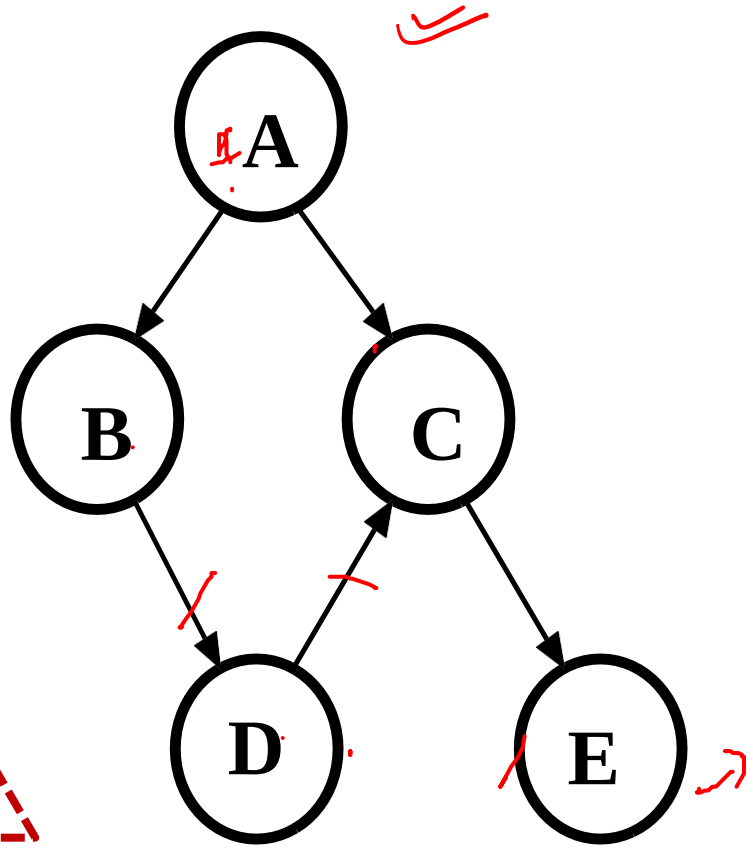
It has one column for each vertex and one row for each edge

DIRECTED ACYCLIC GRAPHS(DAG).

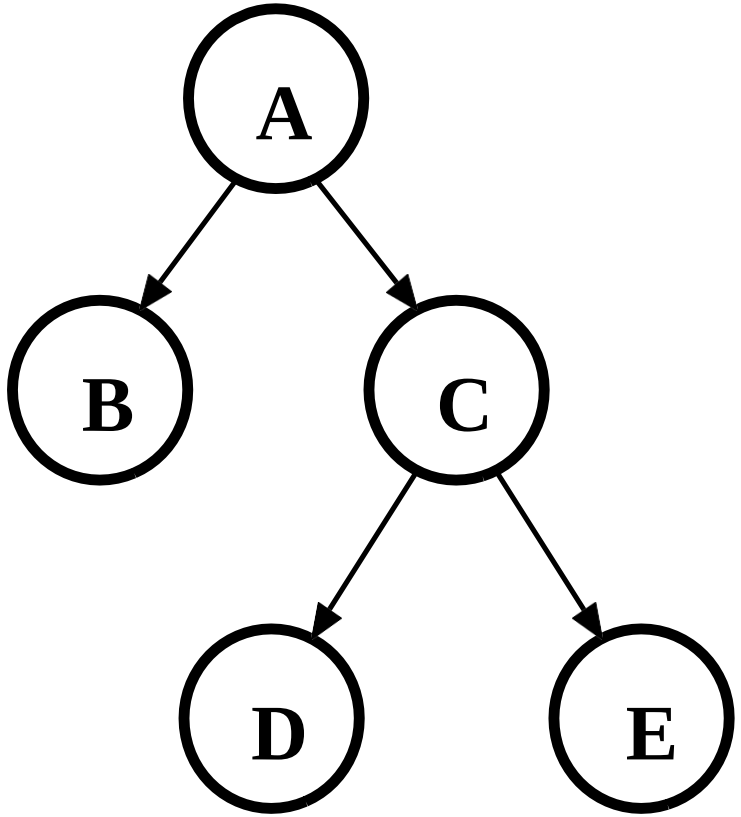
A directed graph with **no directed cycles.**



Directed Graph



Directed Acyclic Graph

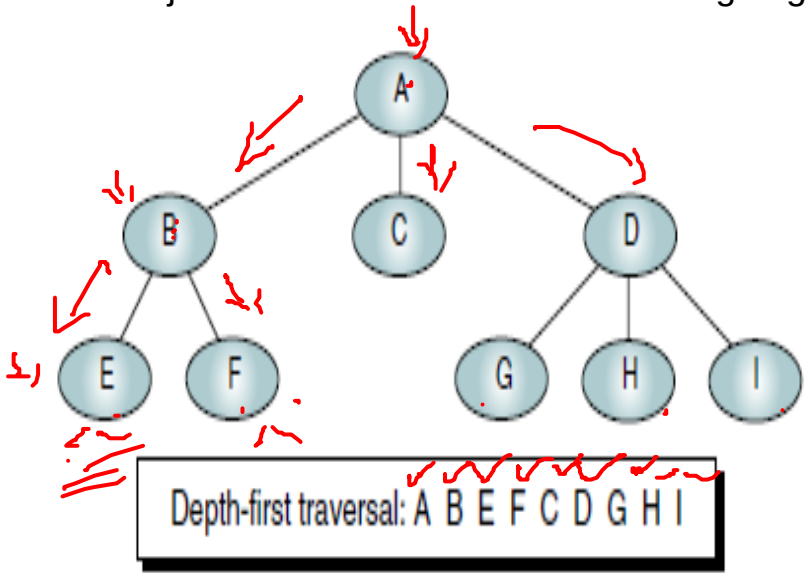


Tree

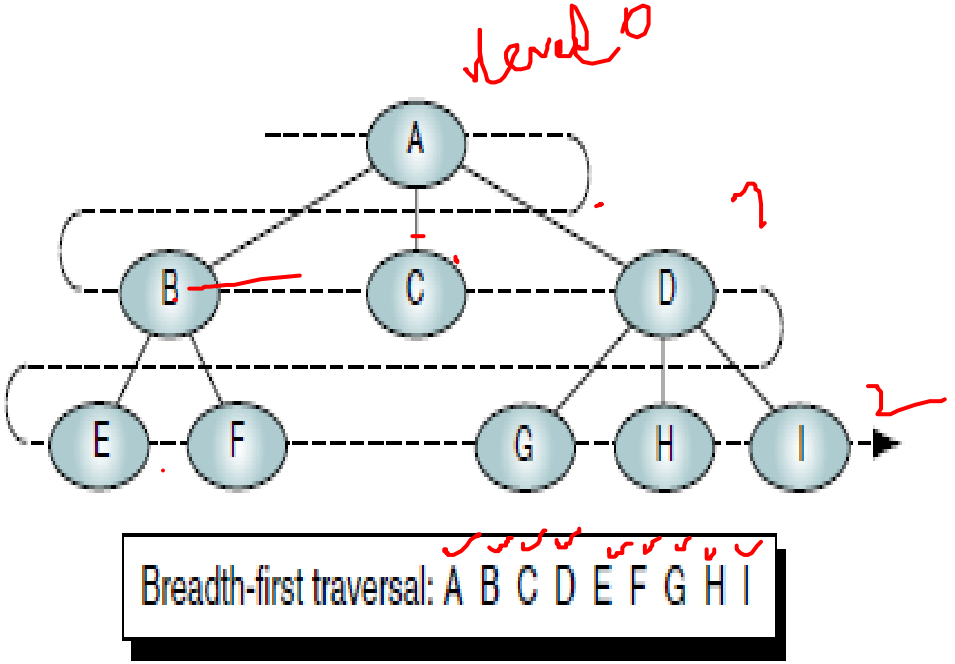
GRAPH TRAVERSAL

Two Commonly used Traversal Techniques are

- **Depth First Search (DFS)**
  - Process all of a vertex's descendants before moving to an adjacent vertex.
- **Breadth First Search (BFS)**
  - Process all adjacent vertices of a vertex before going to the next level.



Depth-first Traversal of a Tree

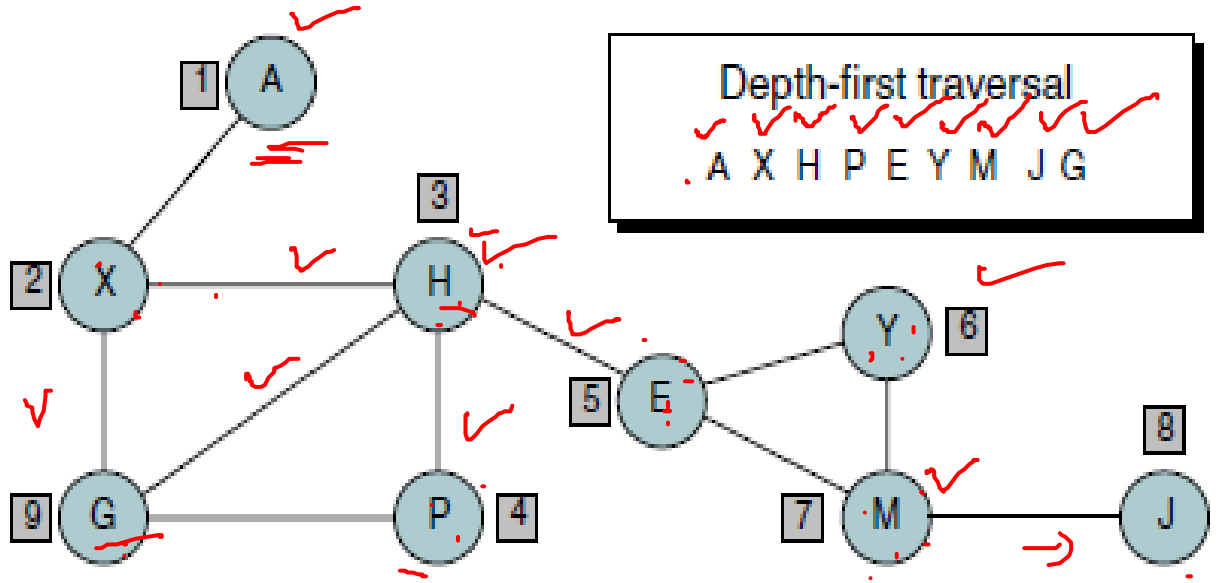


Breadth-first Traversal of a Tree

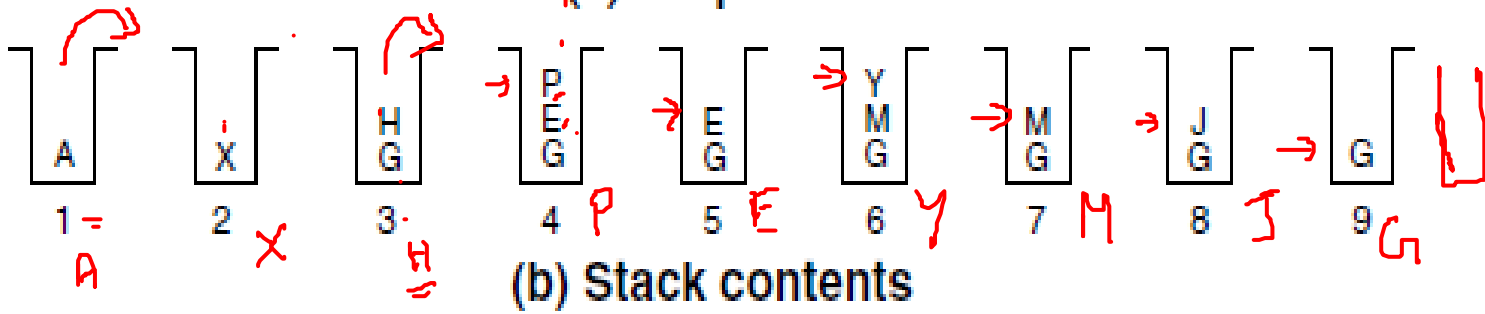
DEPTH FIRST SEARCH (DFS)

Steps:

1. Begin by pushing the first vertex **A** into the stack.
2. Then loop. Pop stack. After processing the vertex, **push all of the adjacent vertices into the stack**. To process **X** at step 2, therefore, pop **X** from the stack, process it, and then push **G & H** into the stack, giving the stack contents for step 3.
3. When **stack is empty**, traversal is complete.



(a) Graph



(b) Stack contents

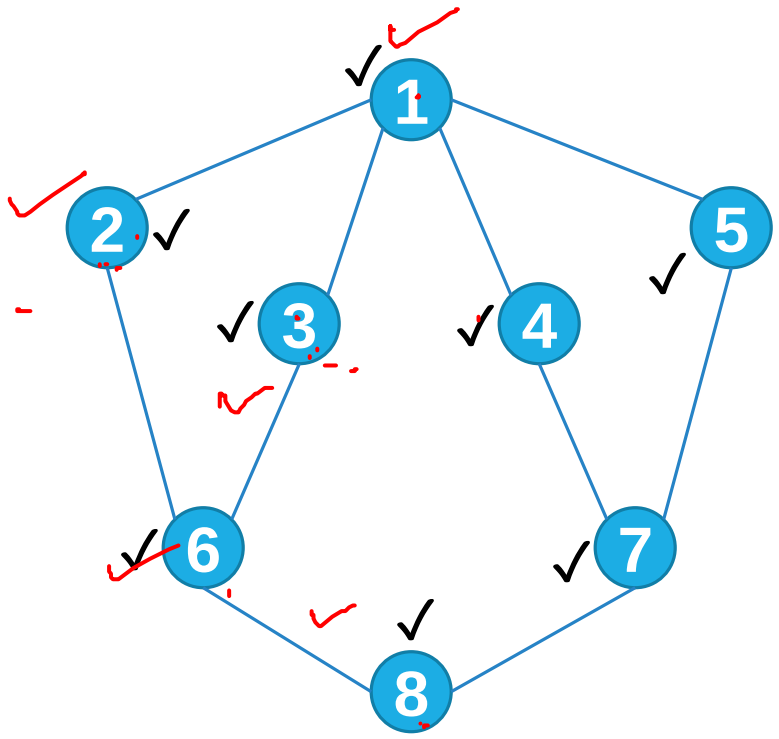
Depth-first Traversal of a Graph

DEPTH FIRST SEARCH (DFS)

It is like preorder traversal of tree

Traversal can start from any vertex V_i

V_i is visited and then all vertices adjacent to V_i are traversed recursively using DFS



DFS (G, 1) is given by

Step 1: Visit (1)

Step 2: DFS (G, 2)
DFS (G, 3)
DFS (G, 4)
DFS (G, 5)

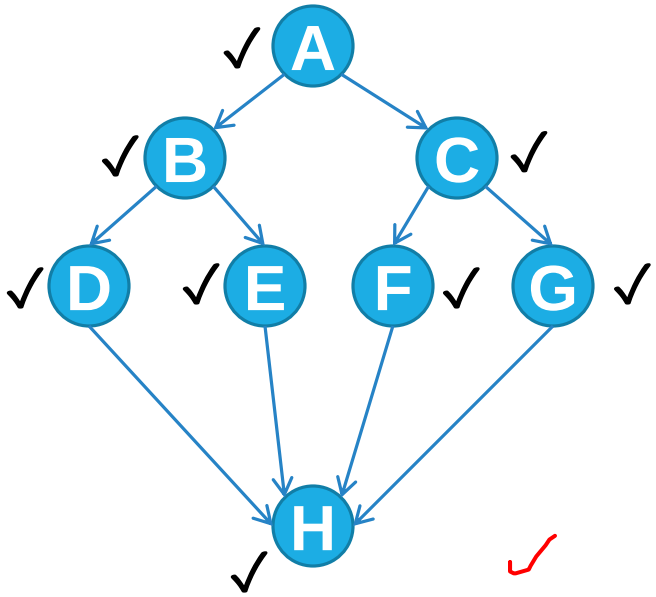
DFS (G, 2):
Step1: Visit(2)
Step 2: DFS (G, 6)

DFS (G, 6):
Step1: Visit(6)
Step 2: DFS (G, 3)
DFS (G, 8)

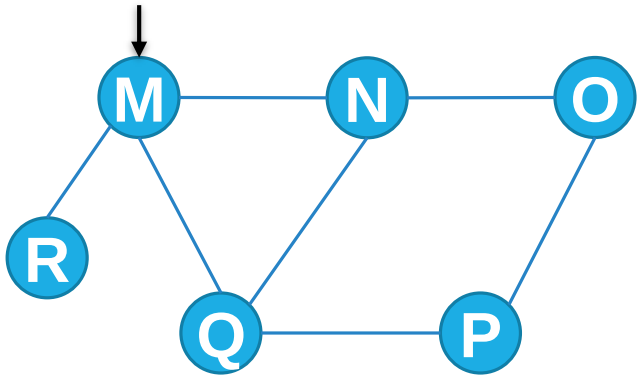
DFS of given graph starting **from** node **1** is given by

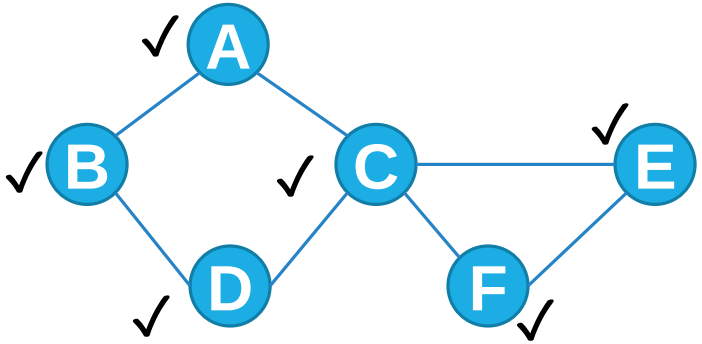
1 2 6 3 8 7 4 5

DEPTH FIRST SEARCH (DFS)



A B D H E C F G



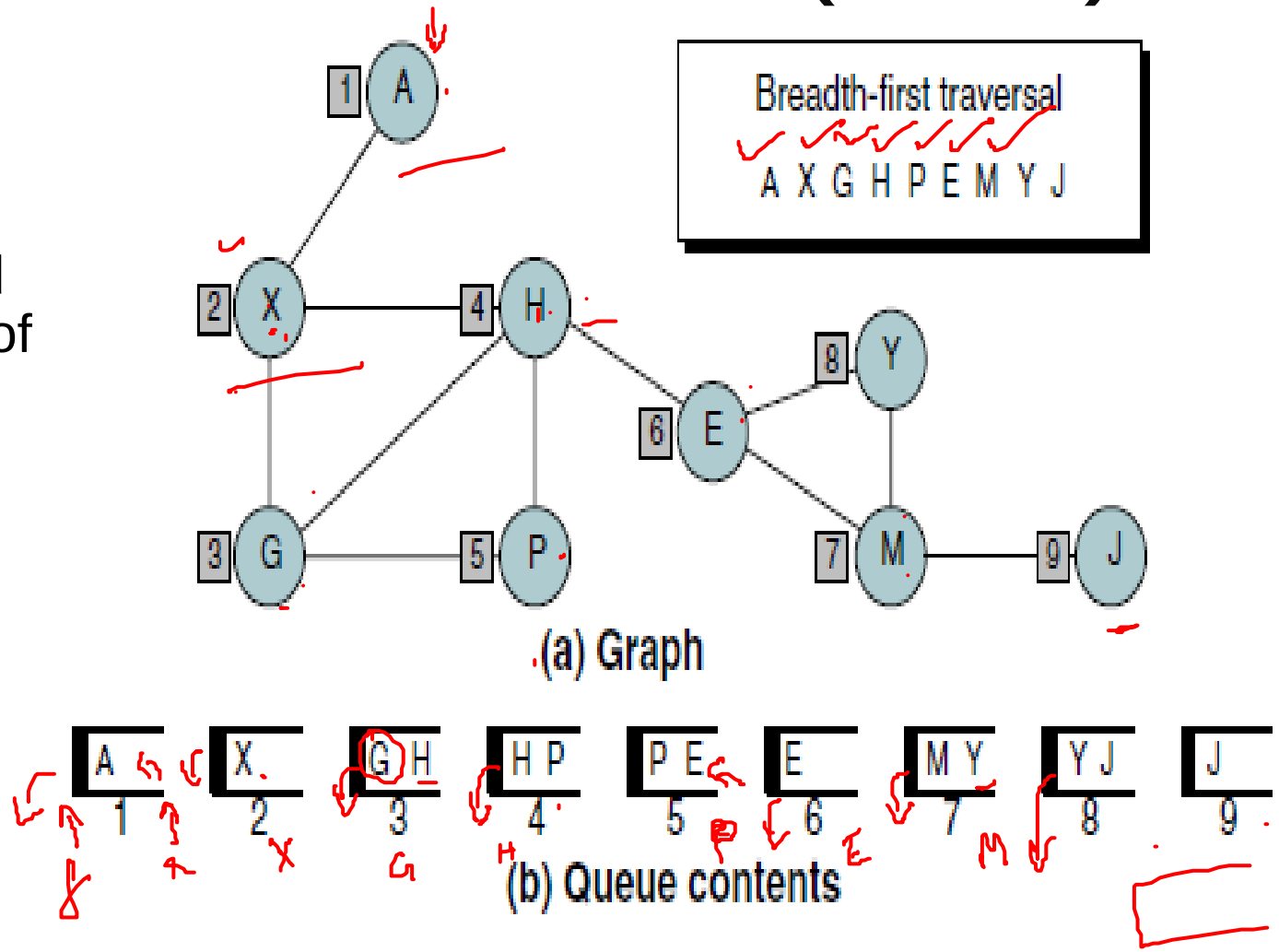


A B D C F E

BREADTH FIRST SEARCH (BFS)

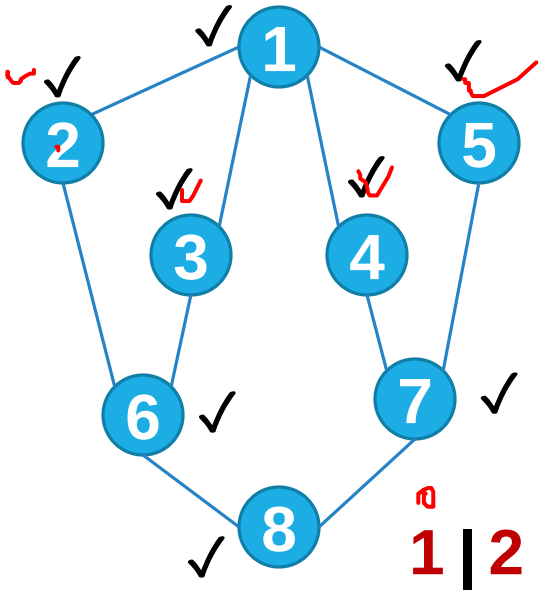
Steps:

1. Begin by enqueuing vertex **A** in the queue.
2. Then loop, dequeuing the queue and processing the vertex from the front of the queue. After processing the vertex, **place all of its adjacent vertices into the queue**. Thus, at step 2, dequeue vertex **X**, process it, and then place vertices **G & H** in the queue. In step 3, process vertex **G**.
3. When **queue is empty**, ***traversal is complete.***

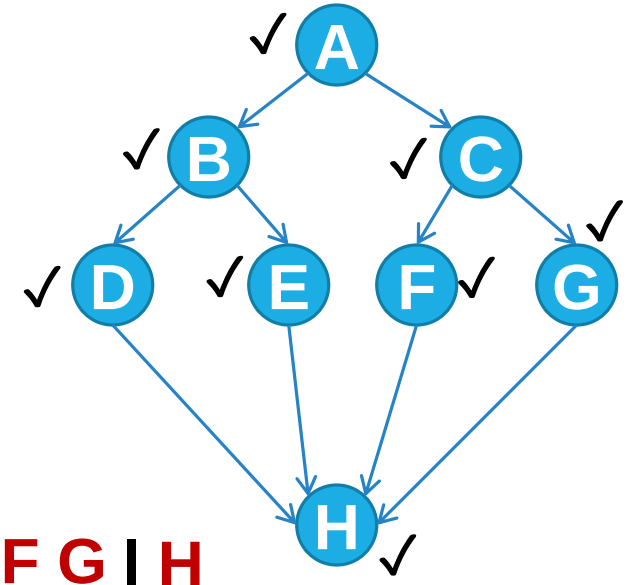


Breadth-first Traversal of a Graph

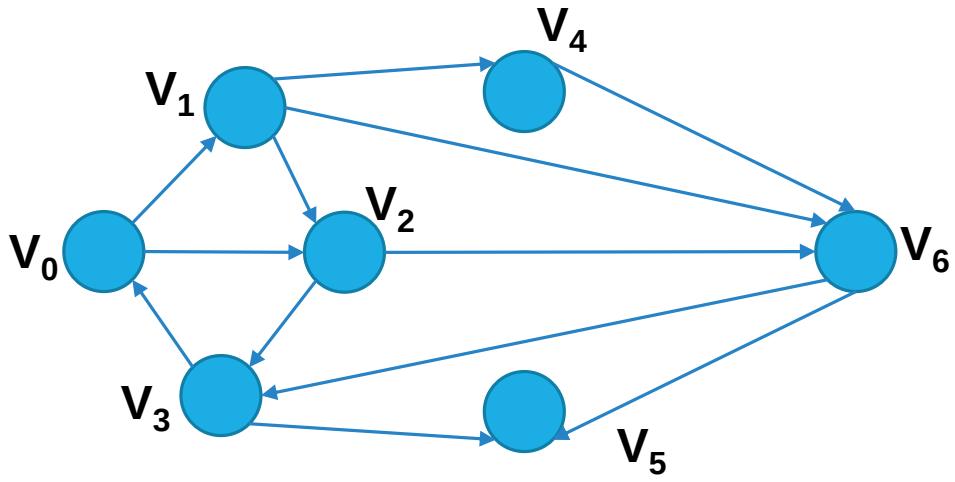
Breadth First Search (BFS)



1 | 2 3 4 5 | 6 7 | 8

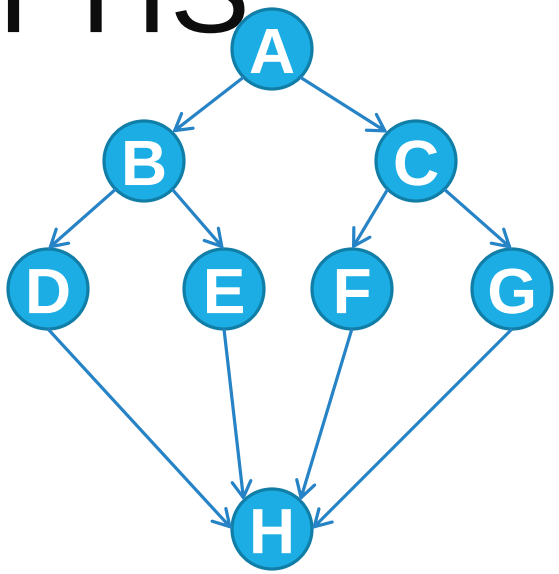


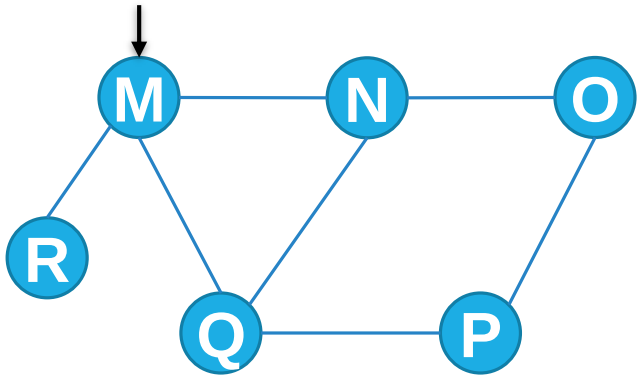
A | B C | D E F G | H

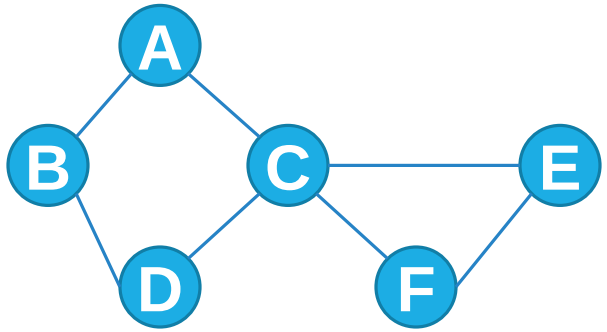


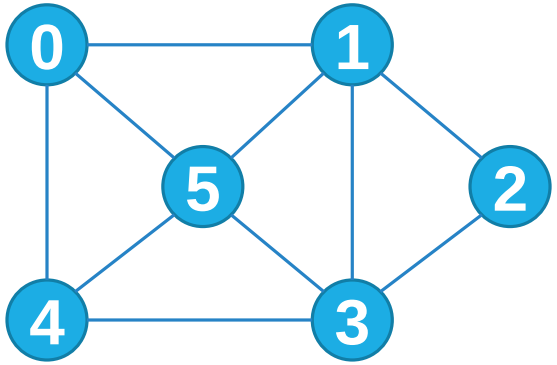
V_0 | V_1 V_2 | V_4 V_6 V_3 | V_5

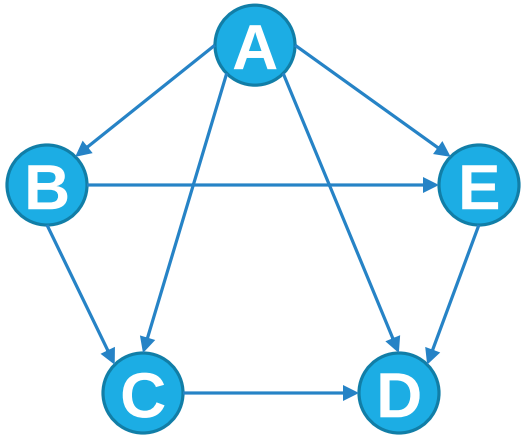
WRITE DFS & BFS OF FOLLOWING GRAPHS

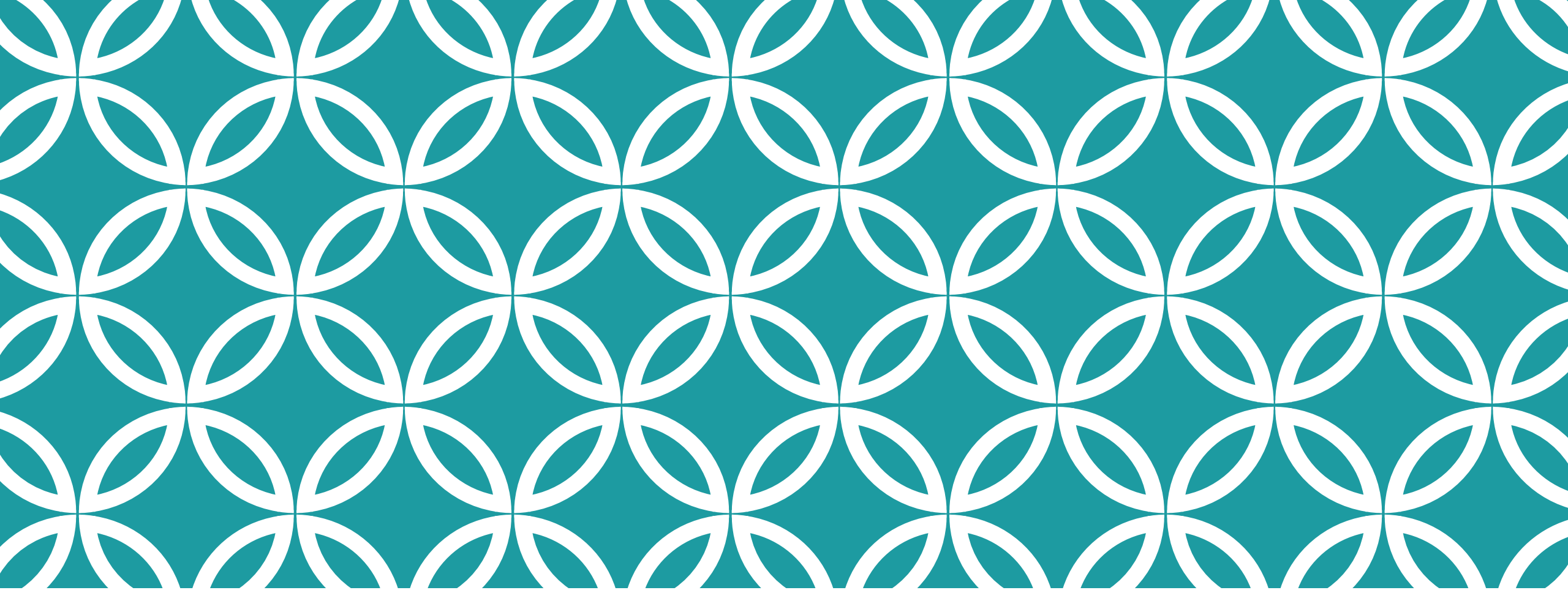












END.

THREADED TREES.

THREADED TREES.

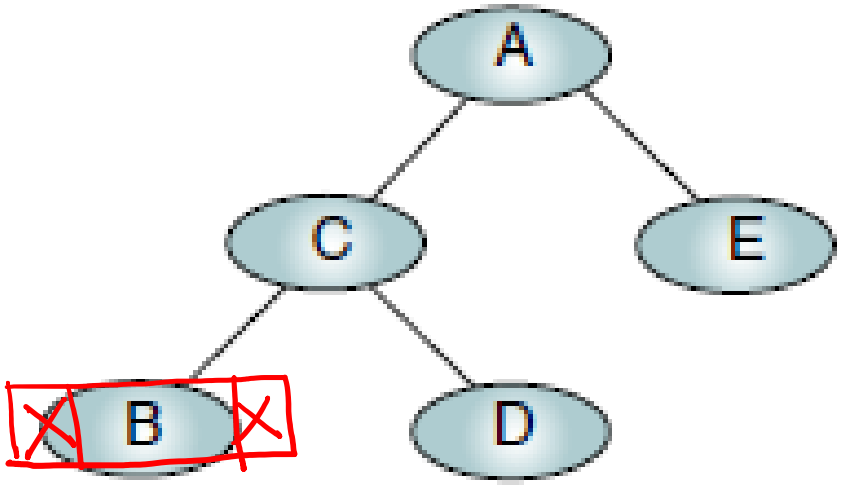
- Binary tree traversal algorithms are written using either recursion or programmer-written stacks. If the tree must be traversed frequently**, using stacks** rather than recursion may be more **efficient**.
- A third alternative is a ***threaded tree***. In a threaded tree, **null pointers are replaced with pointers to their successor nodes.**
- To **build a threaded tree** ----> first build a standard binary search tree.
- Then **traverse the tree**, changing the null right pointers to point to their successors.
- The traversal for a threaded tree is straightforward. Once you locate the far-left node, loop happens, following the thread (the right pointer) to the next node. No recursion or stack is needed. When you find a null thread (right pointer), the traversal is complete.

THREADED TREES.

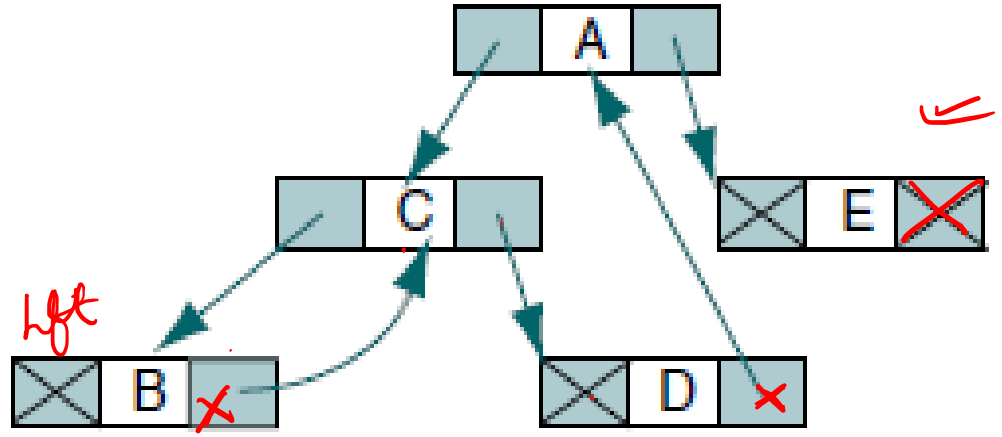
Inorder traversal (LNR)

INORDER – LNR

B C D A E



(a) BInary tree



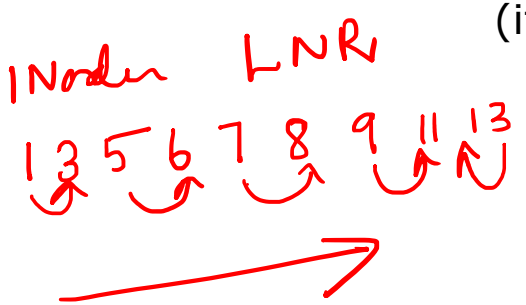
(b) Threaded binary tree

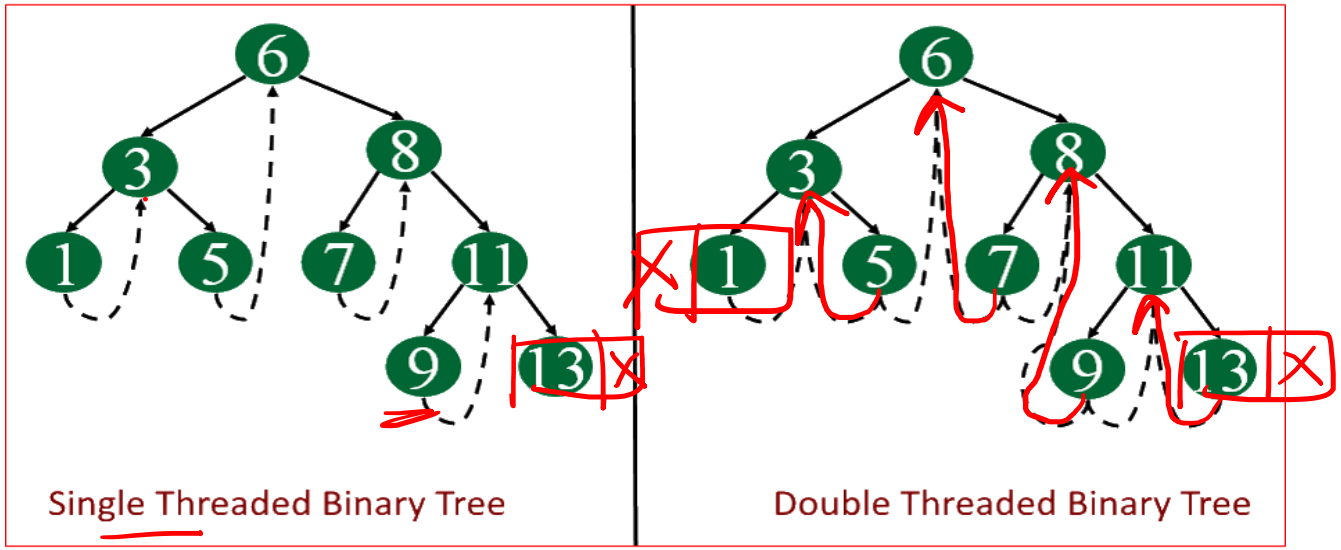
To find the far-left leaf, **backtracking** is performed to process the right subtrees while navigating downwards. This is especially inefficient when the parent node has no right subtree.

Binary Trees

Threaded Binary Tree

- A binary tree with n nodes has n + 1 null pointers
  - Waste of space due to null pointers
  - Replace by threads pointing to inorder successor and/or inorder predecessor (if any)



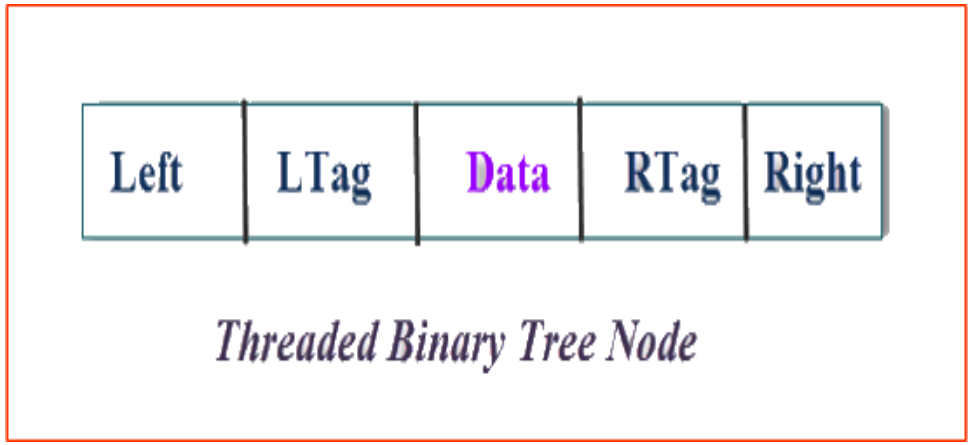


- Single threaded: Makes inorder traversal easier
- Double threaded: Makes inorder and postorder easier

Binary Trees

Threaded Binary Tree (Continued…)

- Implementation requirements
  - Use a boolean value – thread or child pointer (0 : child, 1 : thread)

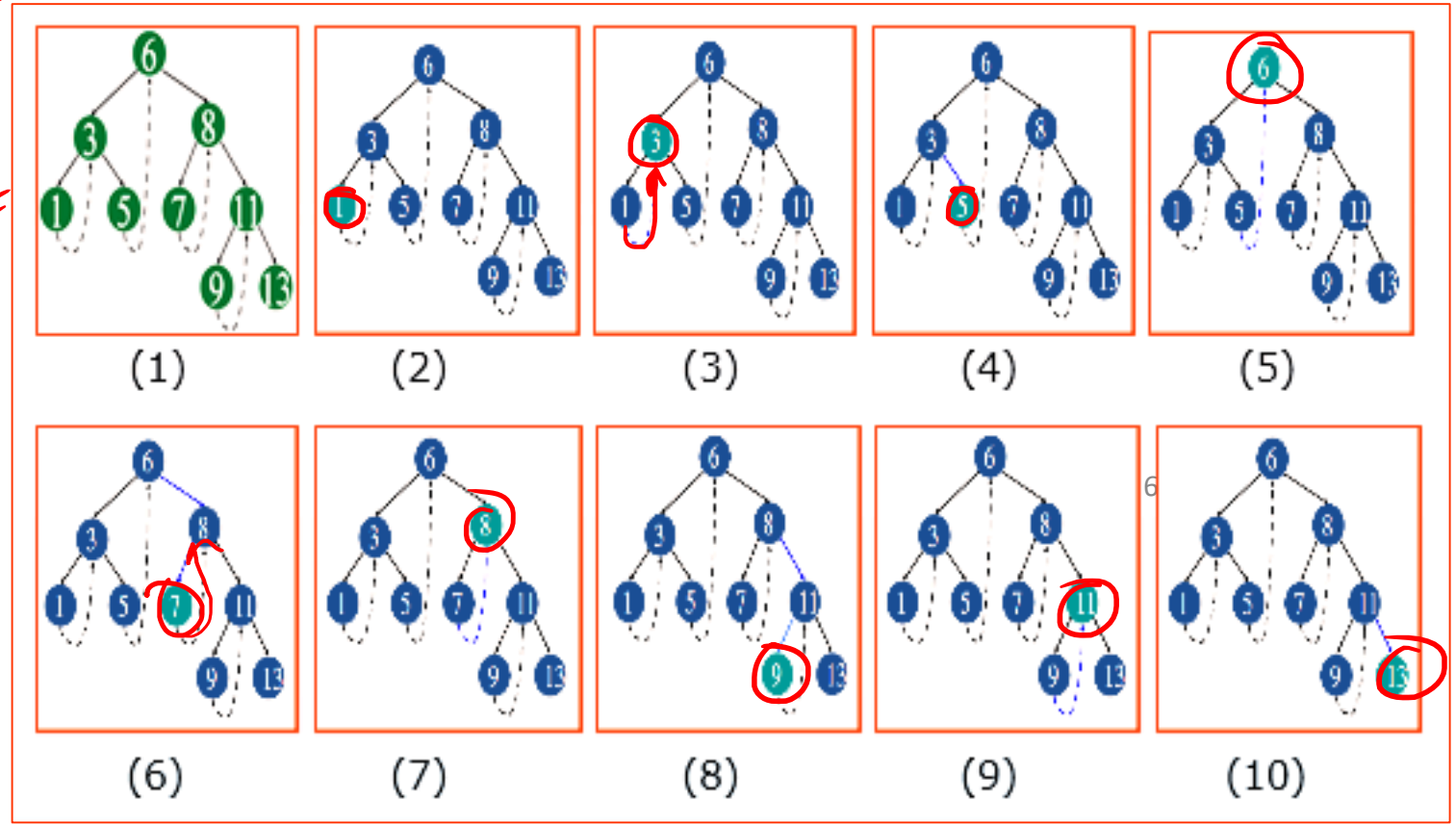


- Advantage: Stack not required for inorder traversal
- Disadvantage: Adjustment of pointers during insertion and deletion of nodes

Binary Trees

Threaded Binary Tree (Continued...)

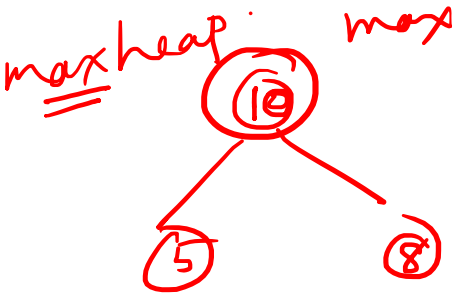
- Example (for inorder traversal)



Priority Queue

What is a priority queue?

- Queue in which each element has a priority associated with it
- Priority – a unique number that determines the relative importance of that element compared to other

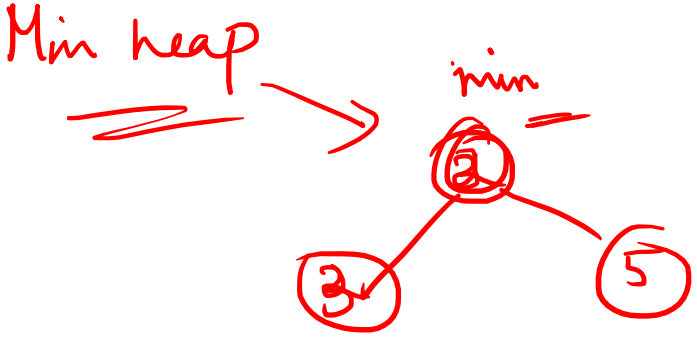


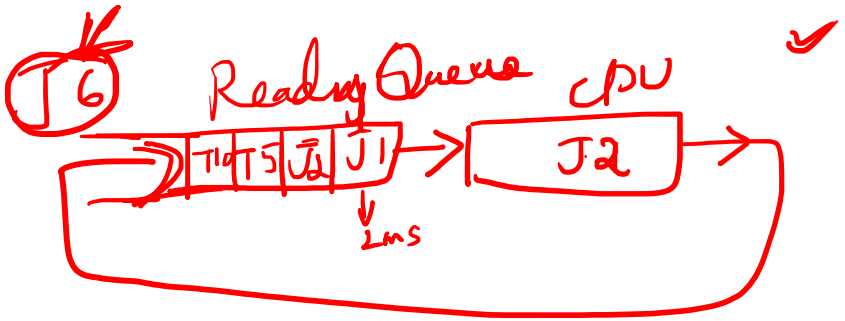
Operations on Priority Queue

- Insertion as usual
- Deletion

Max heap

1. Max (ascending) priority queue – delete element with highest priority
2. Min (descending) priority queue – delete element with least priority





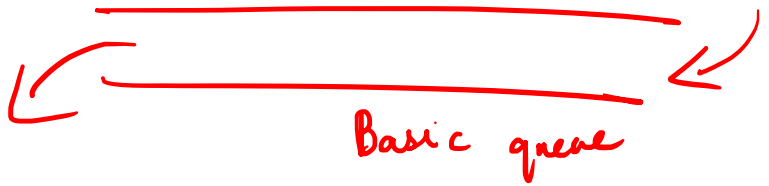
Double Ended Queue

Enqueue
Dequeue

What is a double ended queue (deque) ?

- Queue in which insertion (enqueue) and deletion (dequeue) can be made at both ends

Front
Rear



Operations on a deque

- insert_rear
- insert_front
- delete_front
- delete_rear

Types of deque

- Input-restricted: Insert at rear end only, delete from any end
- Output-restricted: Delete from front end only, insert at any end

Priority Queue and Heap

Priority Queue

- Queue: FIFO data structure
- Priority Queue: Order of deletion is determined by the element's priority
  - Deleted either in increasing or decreasing priority
  - Efficiently implemented with heap data structure
    - Heap is a complete binary tree; can be implemented using the array representation
  - Other representations are leftist trees, fibonacci heaps, binomial heaps, skew heaps and pairing heaps
    - Implemented with linked data structures

AVL TREES.

BALANCED TREES.

AVL TREES.

While the binary search tree is simple and easy to understand, it has **one major problem.**

Not balanced.

AVL TREES.

Also called **AVL Search Trees.**

In **1962**, two Russian mathematicians,

G. M. Adelson-Velskii and **E. M. Landis**, created the **balanced binary tree structure** named after them

“the AVL tree”

AVL SEARCH TREES.

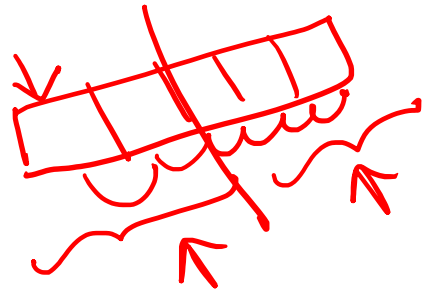
Ht of left subtree − Ht of rt subtree = {-1, 0, 1}

An AVL tree is a **search tree** in which the **heights of the subtrees differ by no more than 1. {-1, 0, 1}**

It is thus a **balanced binary tree**.

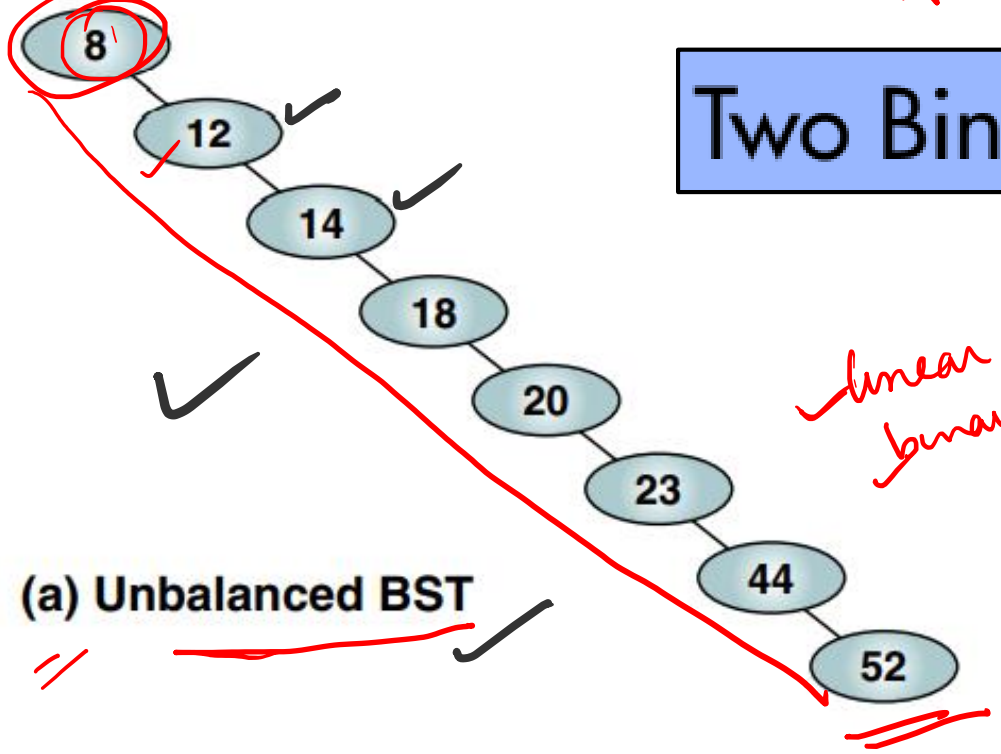
-1: Right High (RH)	**0: Even High (EH)**	**+1: Left High (LH)**
LST **is shorter than** RST	LST **is equal to** RST	LST **is longer than** RST

It takes 2 tests to locate 12.
It takes 3 tests to locate 14.
It takes 8 tests to locate 52.



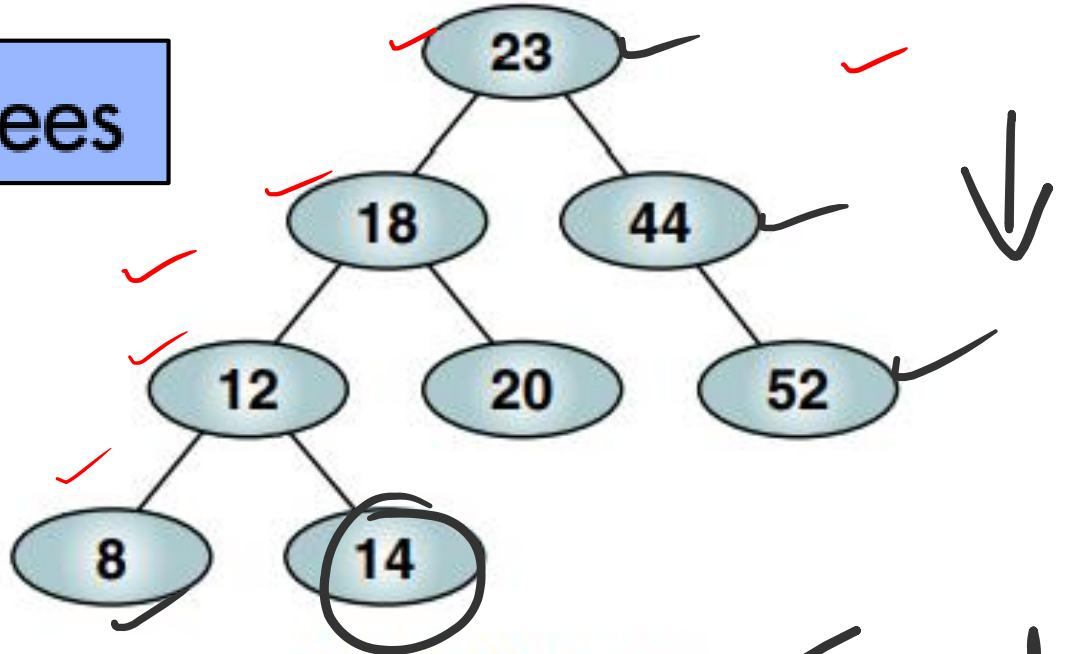
It takes 4 tests to locate 8 and 14.
It takes 3 tests to locate 20 and 52.
The maximum search effort is either 3 or 4.

Two Binary Trees



(a) Unbalanced BST

linear search
binary search



(b) AVL tree

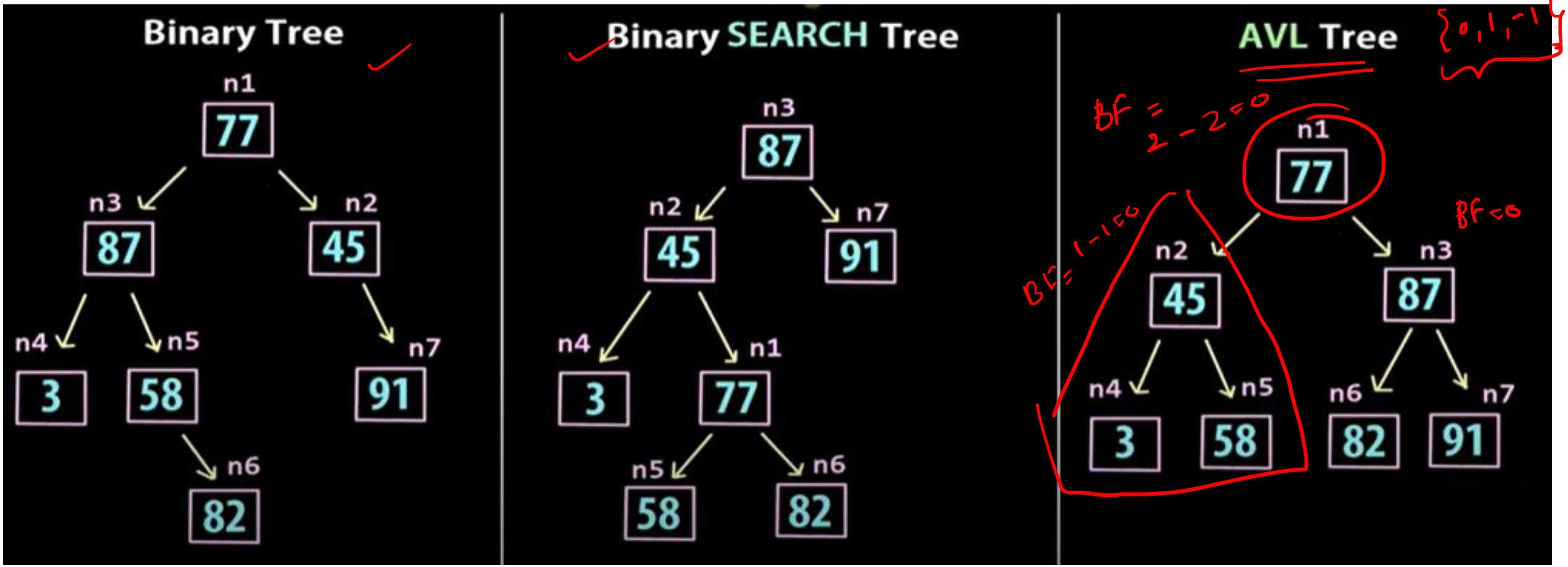
Balanced

Search effort for this binary search tree is O(n)

Search effort for this AVL tree is O(log n)

Examples for AVL Trees.

An AVL tree is a height-balanced binary search tree



AVL SEARCH TREES – Definition.

An AVL tree is a binary tree that either is empty or consists of two AVL subtrees, TL, and TR,

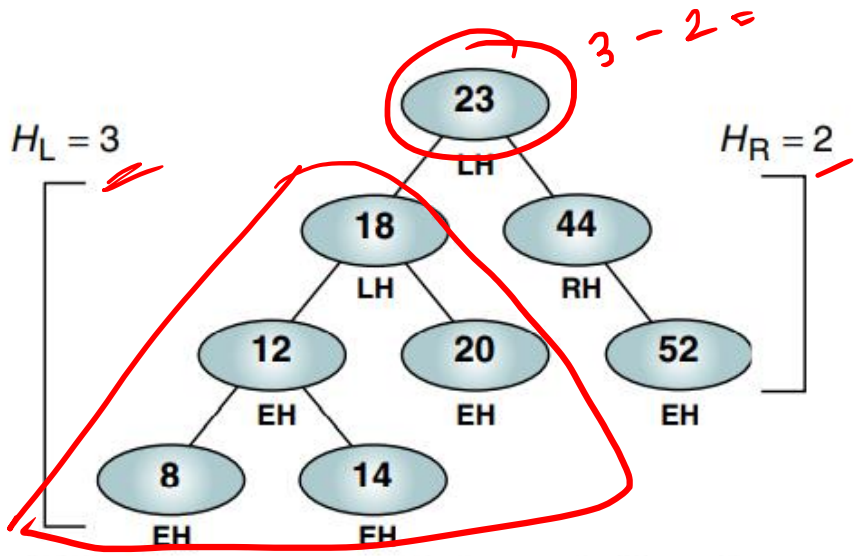
whose heights differ by no more than 1.

$$|H_L - H_R| <= 1$$

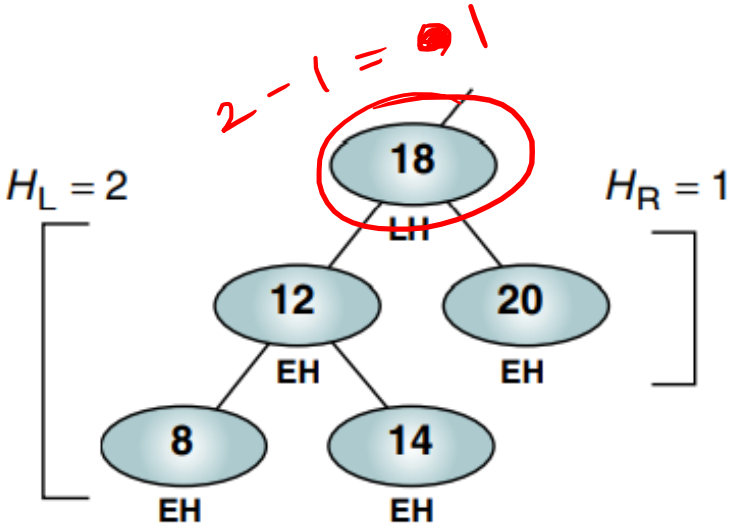
H_L is the ***height of the left subtree*** and H_R is the ***height of the right subtree***.

Because AVL trees are balanced by working with their height, they are also known as **height- balanced trees**.

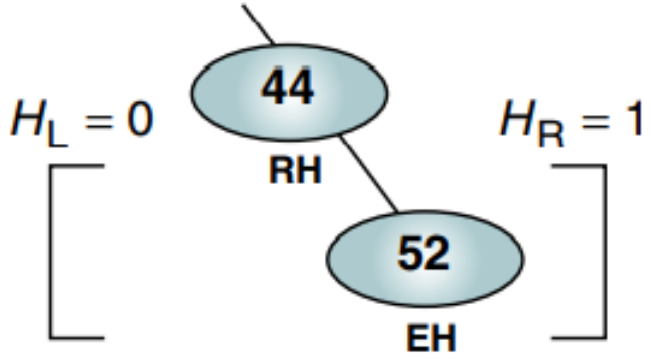
BALANCE FACTORS of AVL Trees.



(a) Tree 23 appears balanced: $H_L - H_R = 1$



(b) Subtree 18 appears balanced: $H_L - H_R = 1$



(c) Subtree 44 is balanced: $| H_L - H_R | = 1$

Why BALANCING TREES?

- Whenever we **insert a node** into a tree or delete a node from a tree, the *resulting tree may be unbalanced*.
- When we detect that a ***tree has become unbalanced***, we must ***rebalance*** it.
- **AVL trees are balanced by rotating nodes either to the left or to the right.**

Consider the basic balancing algorithms for four cases of unbalancing:

UNBALANCED

Right of right

A subtree of a tree that is right high has also become right high.

Left Rotation

Single Rotation Left

Left of left

A subtree of a tree that is left high has also become left high.

Right Rotation

Single Rotation Right

Right of left:

A subtree of a tree that is left high has become right high.

Left Right Rotation

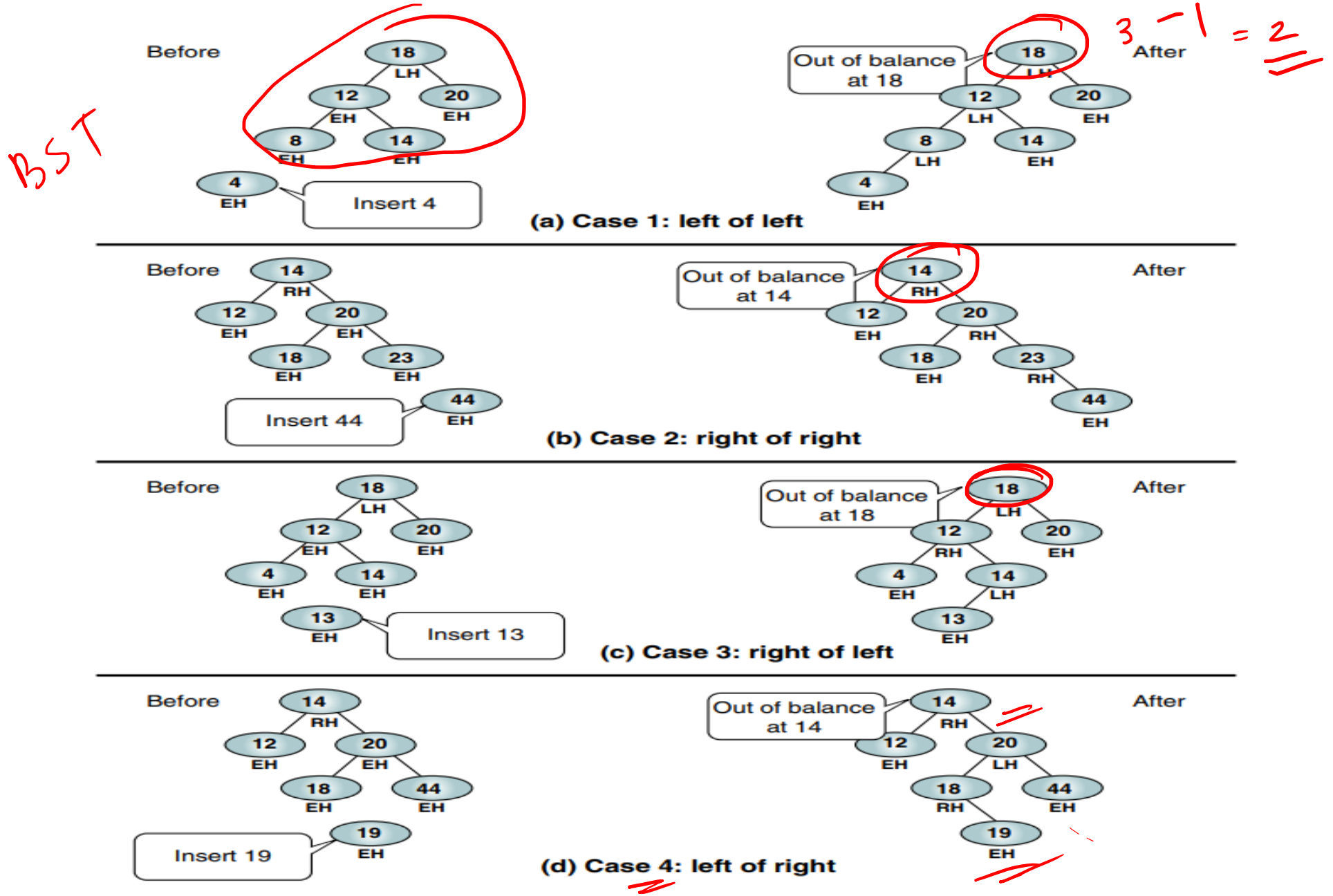
Double Rotation

Left of right:

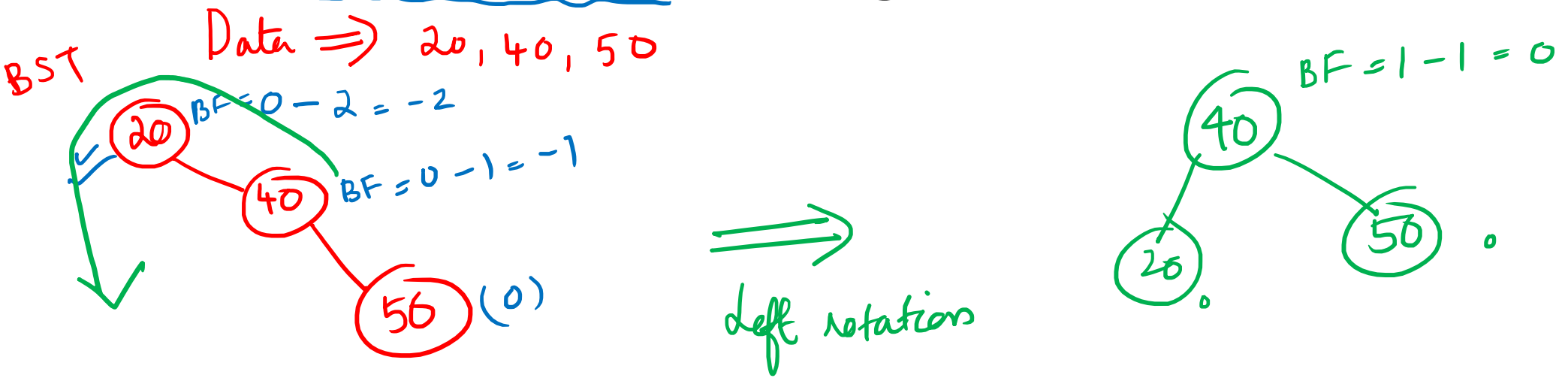
A subtree of a tree that is right high has become left high.

Right Left Rotation

Double Rotation

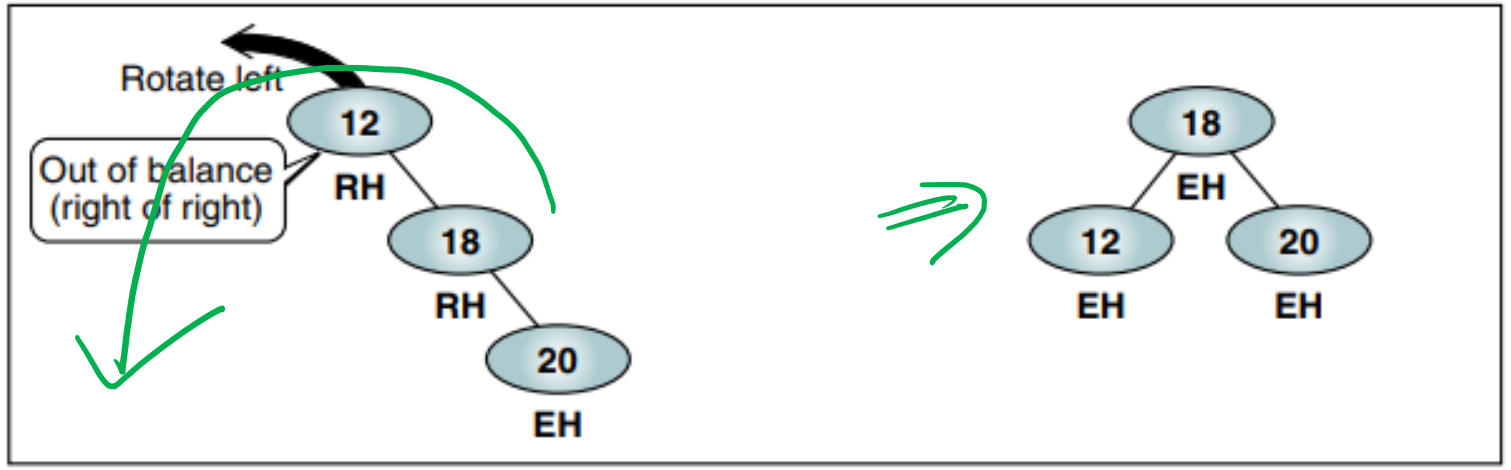

Before
18
LH
12
EH
20
EH
8
EH
14
EH
4
EH
Insert 4
Out of balance at 18
After
(a) Case 1: left of left
Before
14
RH
12
EH
20
EH
18
EH
23
EH
Insert 44
44
EH
Out of balance at 14
After
(b) Case 2: right of right
Before
Insert 13
Out of balance at 18
After
(c) Case 3: right of left
Before
Insert 19
19
EH
Out of balance at 14
After
(d) Case 4: left of right

Case 1: Left Rotation or Single Rotation Left

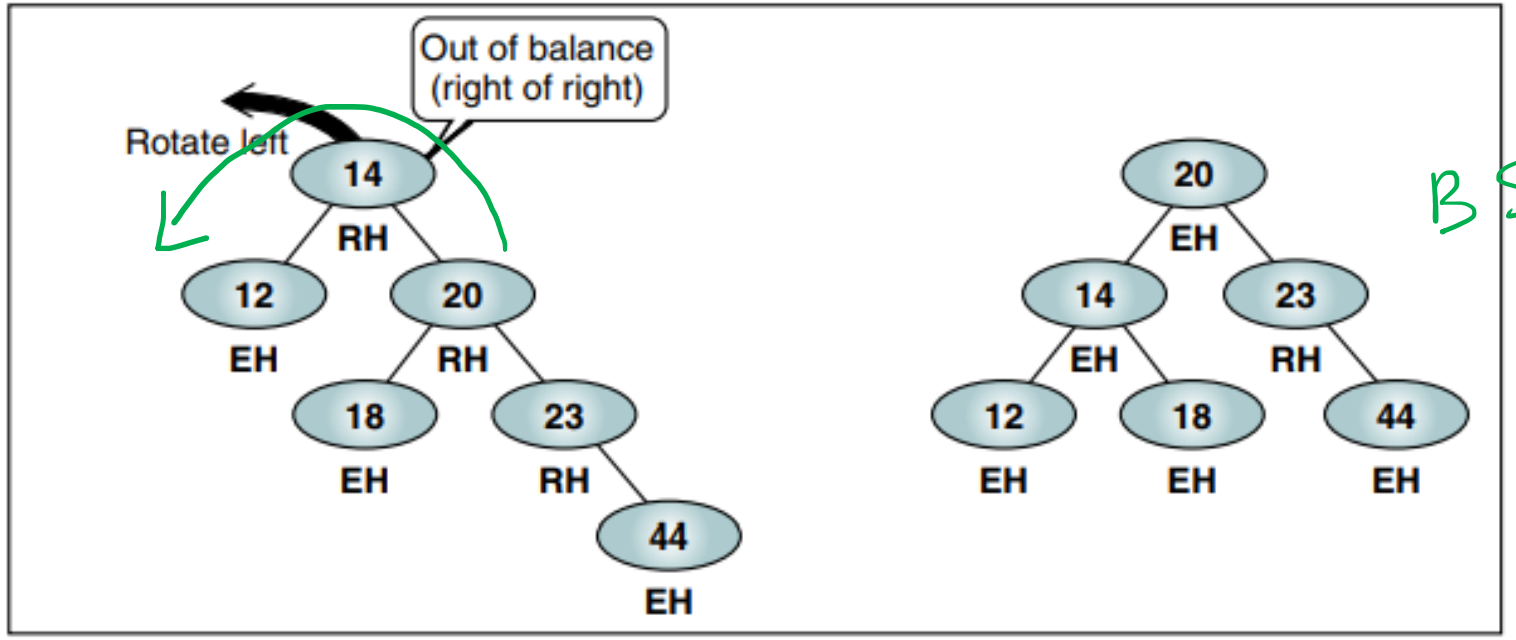


Insertion is Right of Right

Case 1: Left Rotation or Single Rotation Left



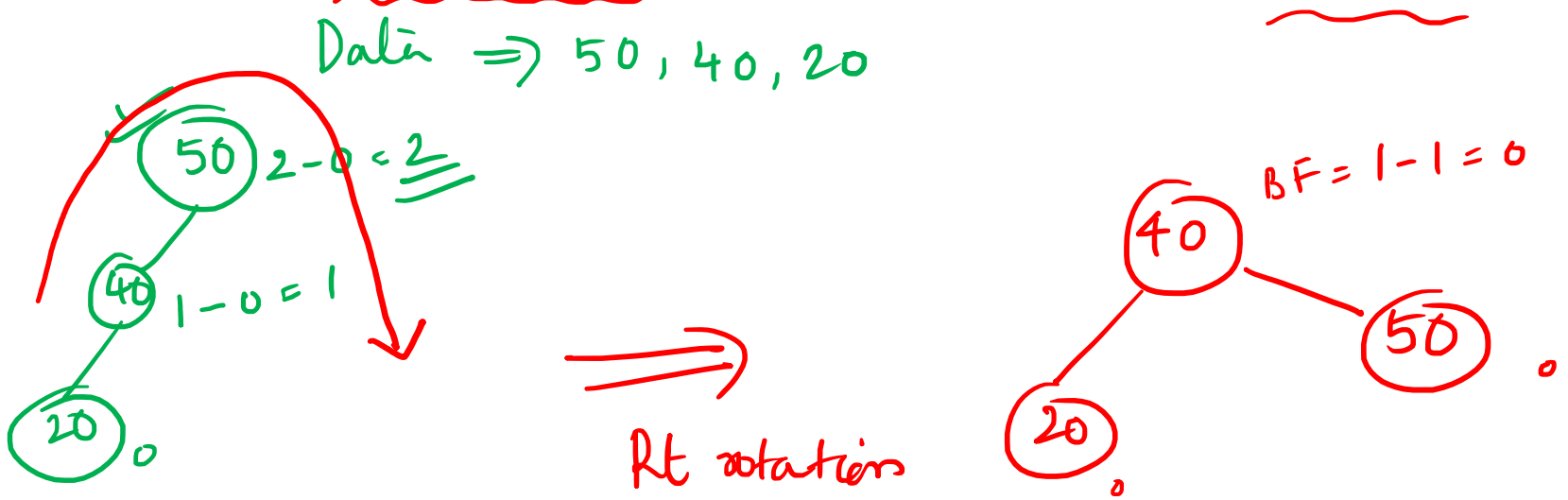
(a) Simple left rotation



BST

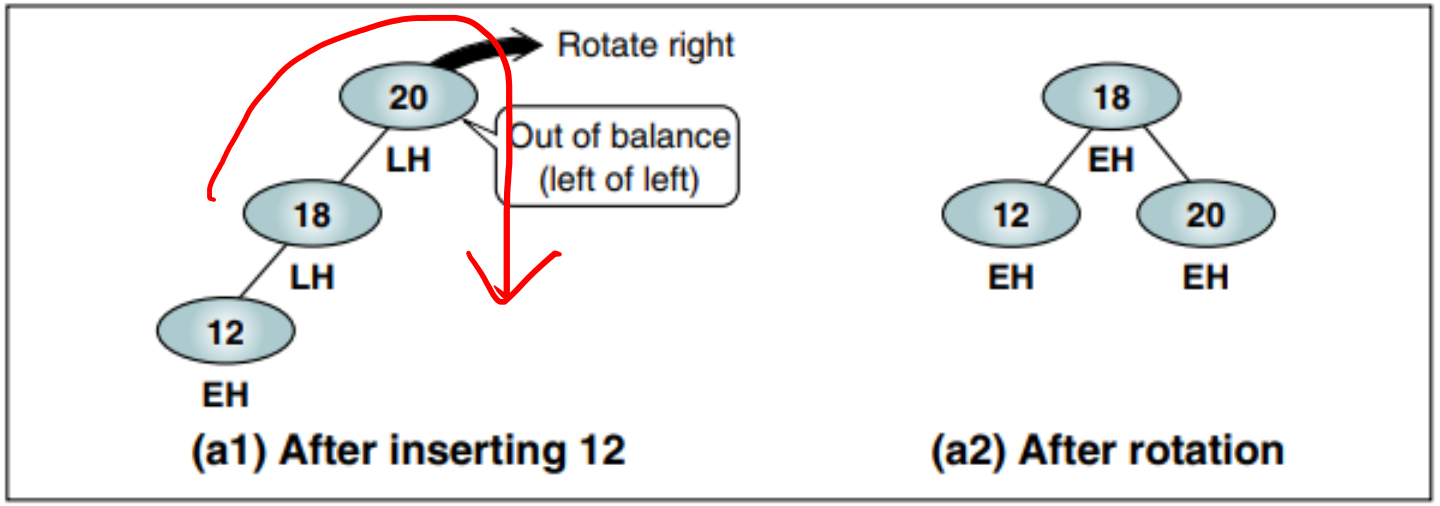
(b) Complex left rotation

Case 2: Right Rotation or Single Rotation Right

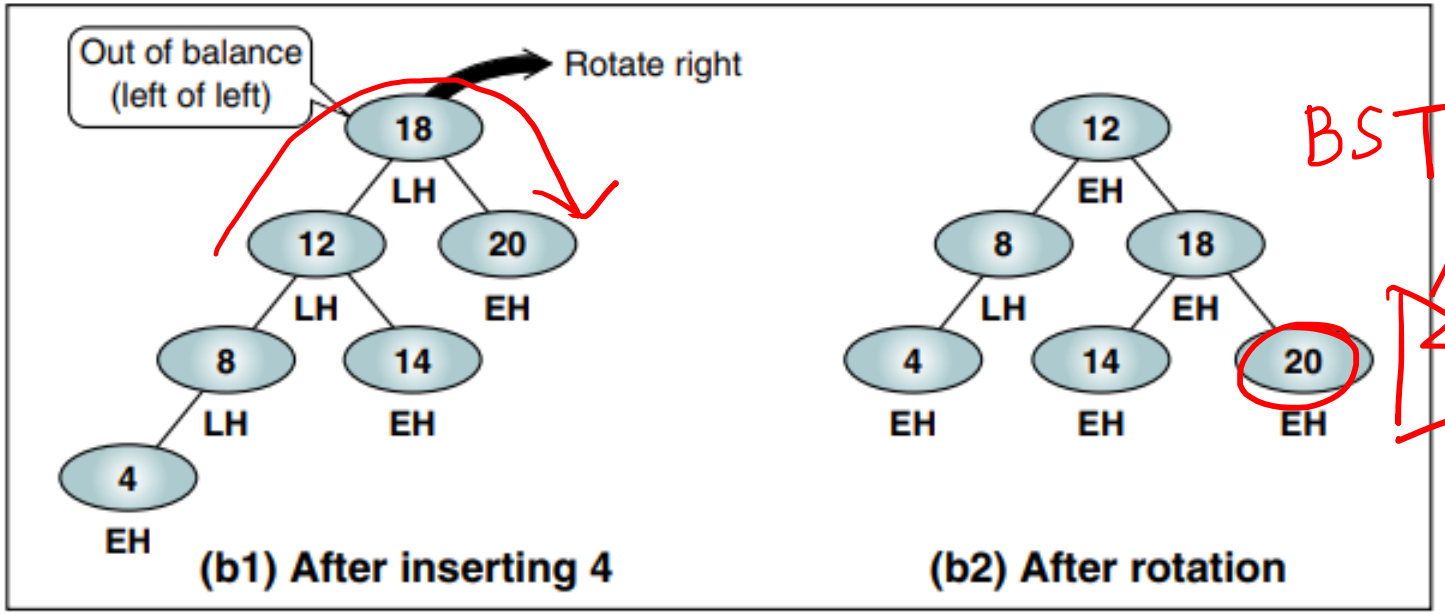


Insertion is Left of Left

Case 2: Right Rotation or Single Rotation Right



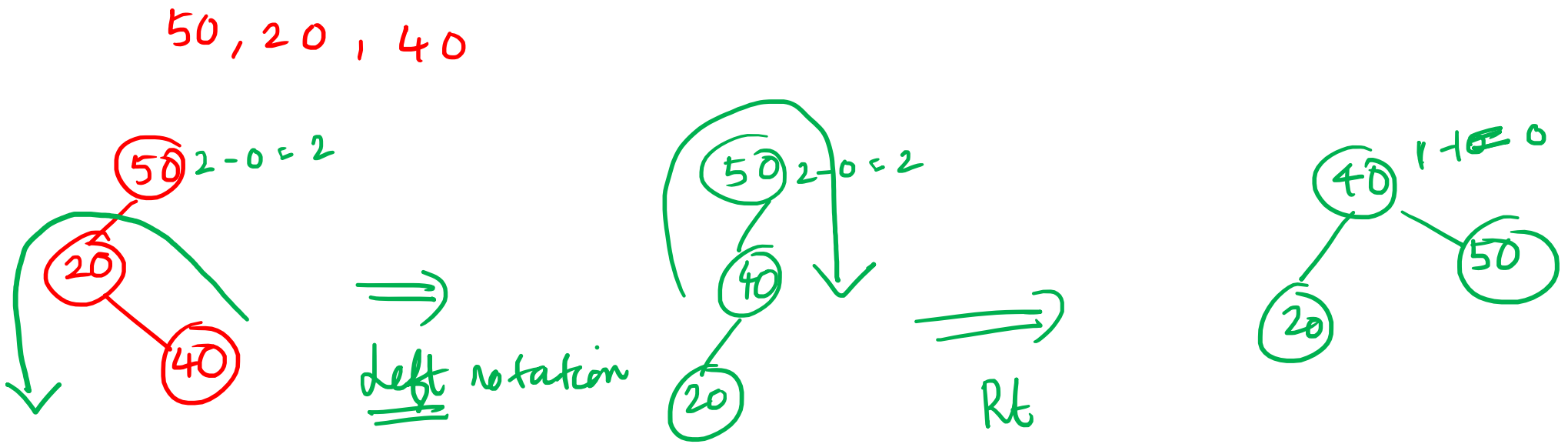
(a) Simple right rotation



(b) Complex right rotation

BST

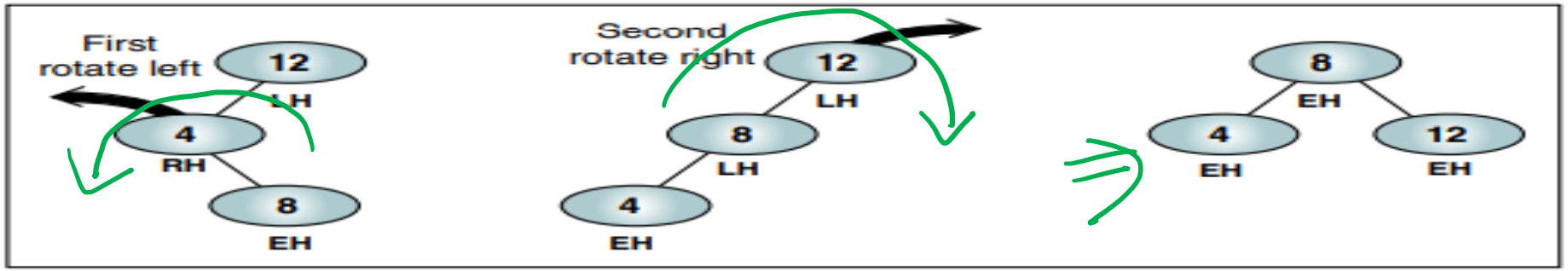
Case 3: Left - Right Rotation or Double Rotation



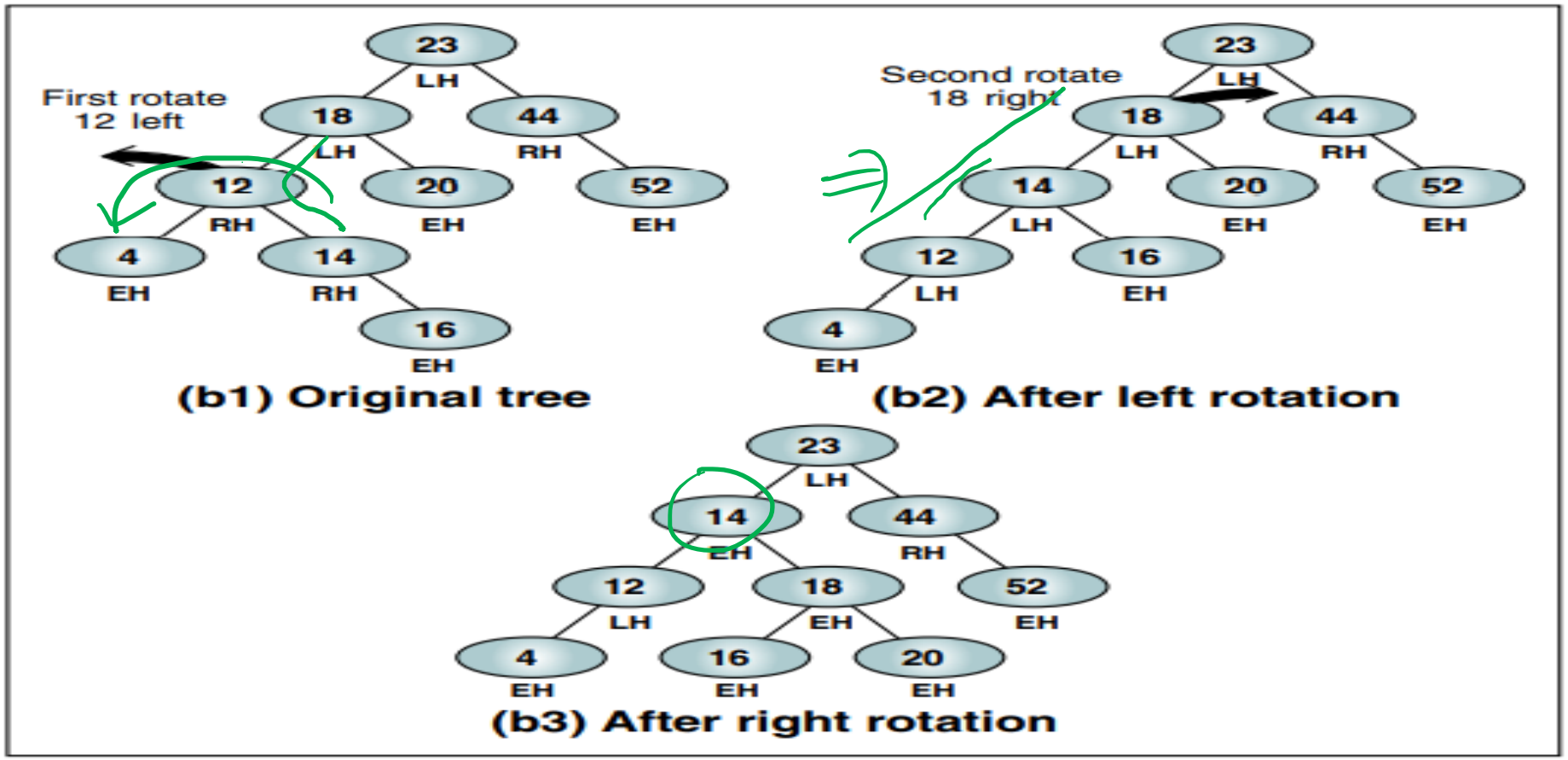
Insertion is Left and then Right (Right of left Insertion).

.

Case 3: Left - Right Rotation or Double Rotation Left



(a) Simple double rotation right



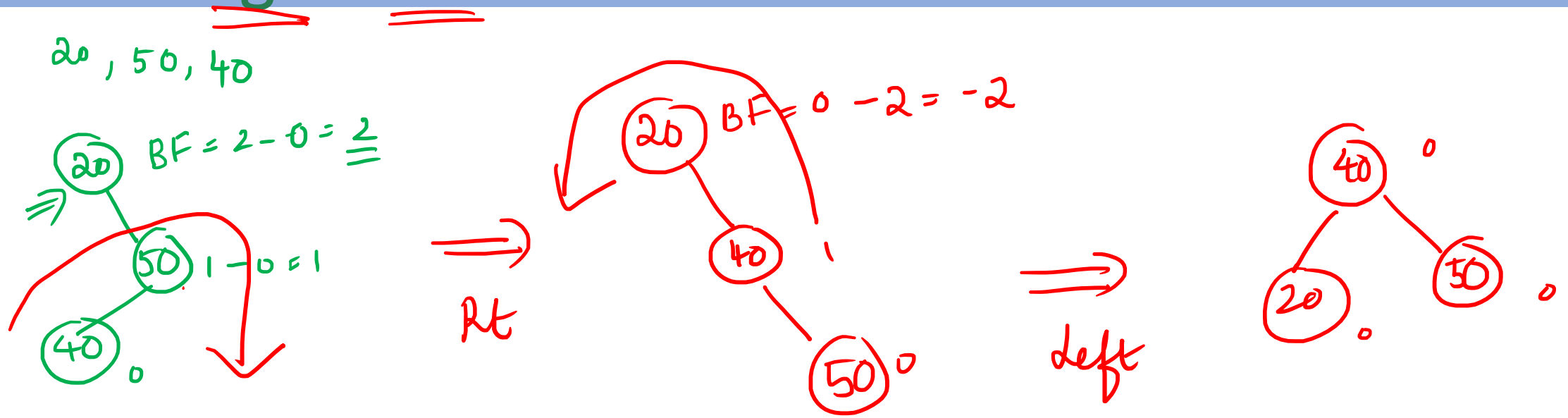
(b1) Original tree

(b2) After left rotation

(b3) After right rotation

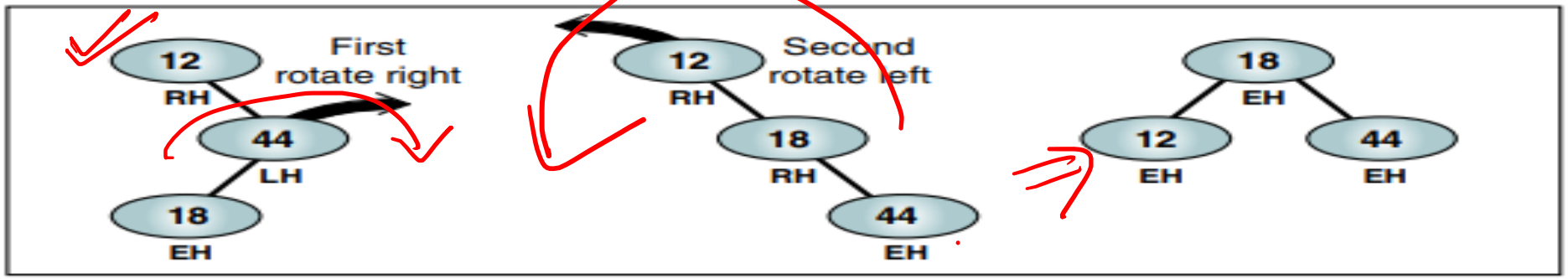
(b) Complex double rotation right

Case 4: Right-Left Rotation or Double Rotation

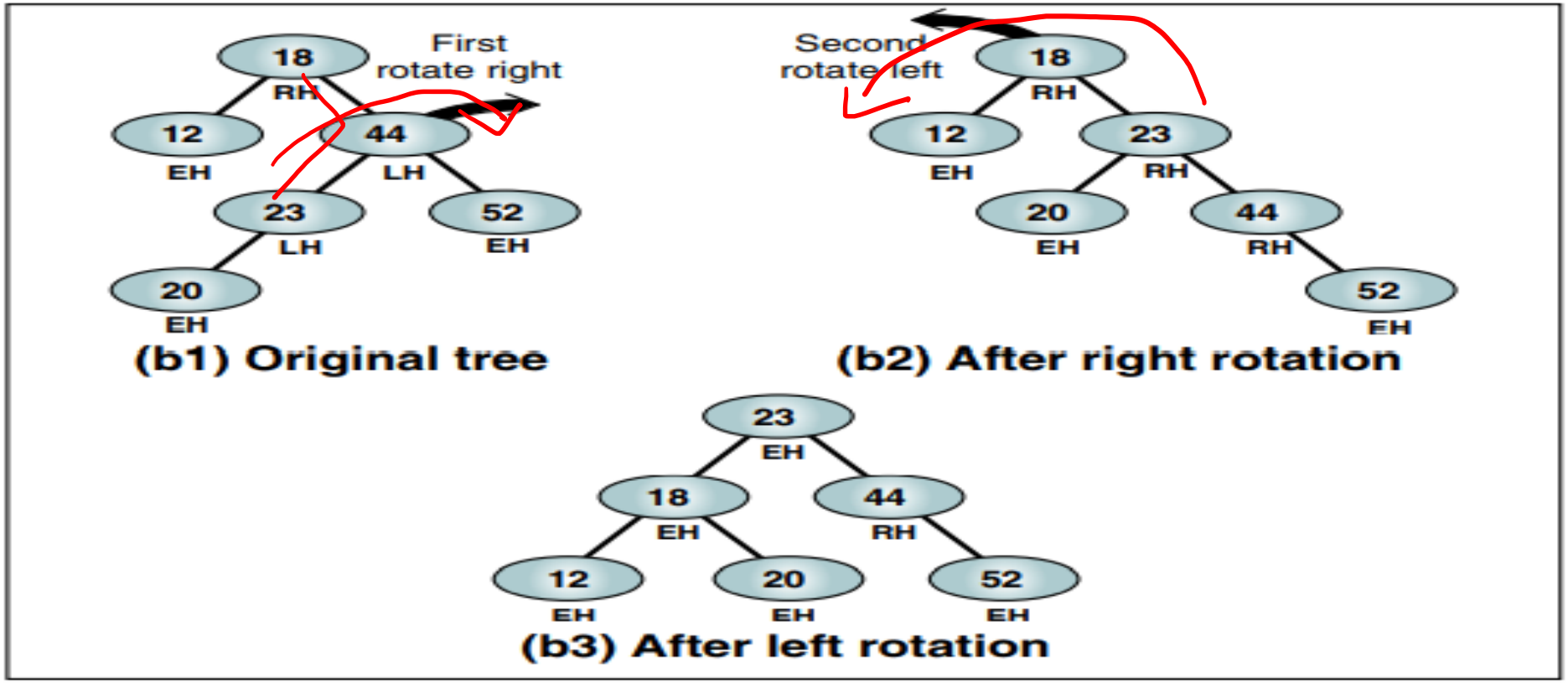


Insertion is Right and then left (Left of Right Insertion).

Case 4: Right-Left Rotation or Double Rotation



(a) Simple double rotation right



(b1) Original tree

(b2) After right rotation

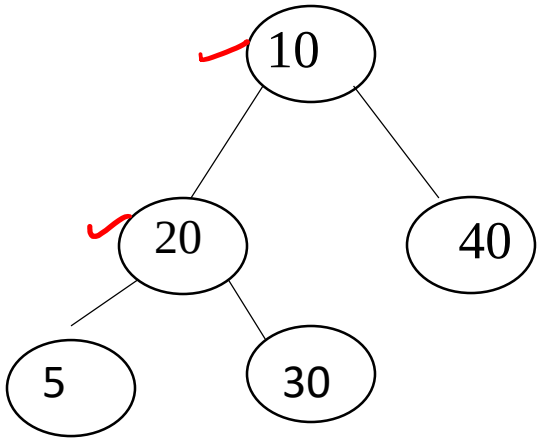
(b3) After left rotation

(b) Complex double rotation right

Iterative traversals of binary tree

Iterative preorder traversal:

- Every time a node is visited, its info is printed and node is pushed to stack, since the address of node is needed to traverse to its right later.



Pre - PLR

10, 20, 50, 30, 40

- Here 10 is printed and this node is pushed onto stack and then move left to 20. This is done because we need to go right of 10 later.

  Similarly 20, 5, 30 and 40 are moved to stack

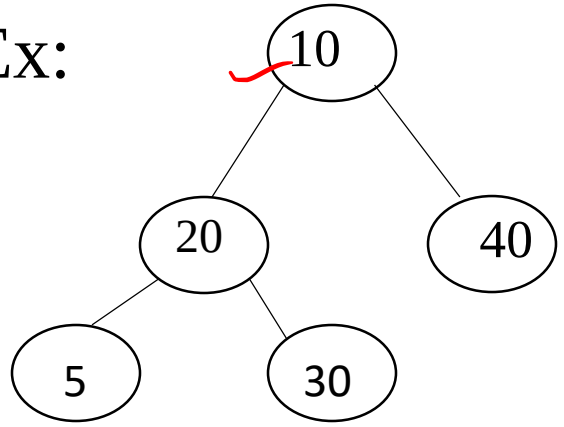
- Once traversing the left side is over, pop the most pushed node and go to its right.

  In the previous tree, after 5 is printed, recently pushed node 20 is popped and move right to 30.

```
/*function for iterative preorder traversal*/
Void preorder(NODEPTR root)
{
  NODEPTR cur;
  NODEPTR stack[20];
  int top=-1;
  if(root==NULL)
  {
        printf("tree is empty");
        return;
  }
```

```
cur=root;
for(; ;)
{ while(cur!=NULL)
  {
        printf("%d", cur→info);
        push(cur,&top,s)          /*push the node to stack*/
        cur=cur→llink;
  }
  if(!stack_empty(top))           /*more nodes existing*/
  {
        cur=pop(&top,s);          /* pop most recent node*/
        cur=cur→rlink;            /*traverse right*/
  }
  else return;
}}
```

Ex:

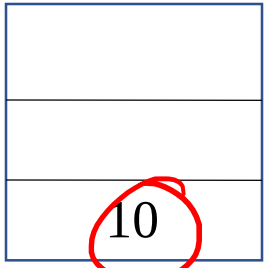


1. After 1st iteration of while loop

10 is printed

Node 10 is pushed to stack

Cur=cur→llink; i.e Cur=20

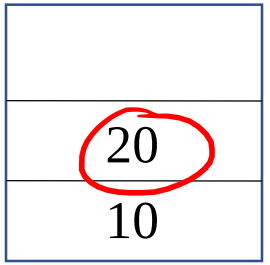


2. After 2nd iteration of while loop

20 is printed

Node 20 is pushed to stack

Cur=cur→llink; i.e Cur=5

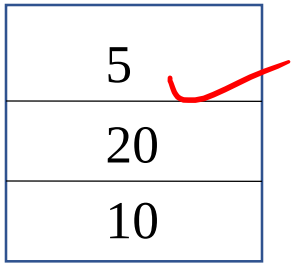


3. After 3rd iteration of while loop

5 is printed

Node 5 is pushed to stack

Cur=cur→llink; i.e Cur=NULL

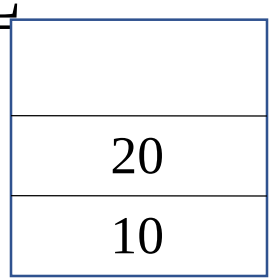


4. While loop terminates since cur==NULL

Stack is not empty

cur=pop( );i.e cur=node 5;

cur=cur→rlink; i.e cur=NULL

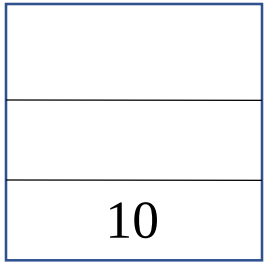


5. cur==NULL, while loop not entered

Stack not empty

cur=pop( ); i.e cur=node 20

cur=cur→rlink; i.e cur=30

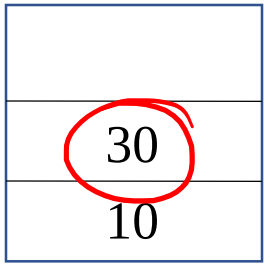


6. While loop is entered

30 is printed

Node 30 is pushed to stack

Cur=cur→llink; i.e cur=NULL

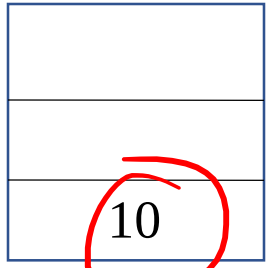


7. While loop terminates since cur==NULL

Stack not empty

cur=pop( );i.e cur=node 30;

cur=cur→rlink; i.e cur=NULL

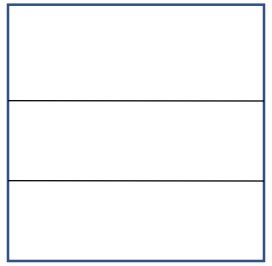


8. cur==NULL, while loop not entered

Stack not empty

cur=pop( ); i.e cur=node 10

cur=cur→rlink; i.e cur=40

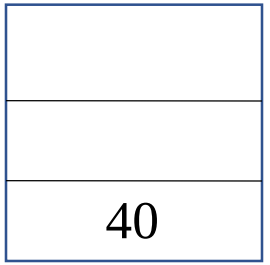


9. While loop is entered

40 is printed

Node 40 is pushed to stack

cur=cur→llink; i.e cur=NULL

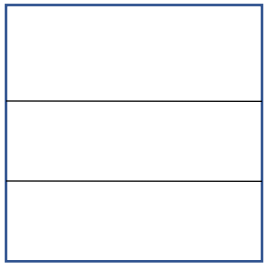


10. While loop terminates since cur==NULL

Stack not empty

cur=pop( );i.e cur=node 40;

cur=cur→rlink; i.e cur=NULL



11. cur==NULL and stack empty

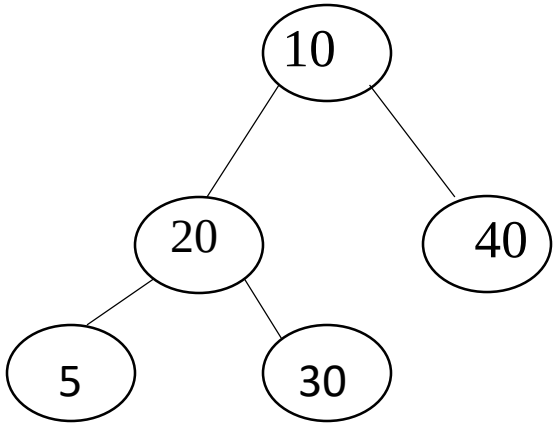
return

Hence elements printed are 10, 20, 5, 30, 40

Preorder.

Iterative inorder traversal:

- Every time a node is visited, it is pushed to stack without printing its info and move left.
- After finishing left, pop element from stack, print it and move right.



Here 10, 20 , 5 is pushed to stack. Then pop 5, print it and move right.

Now pop 20, print it and move right and push 30 and move left.

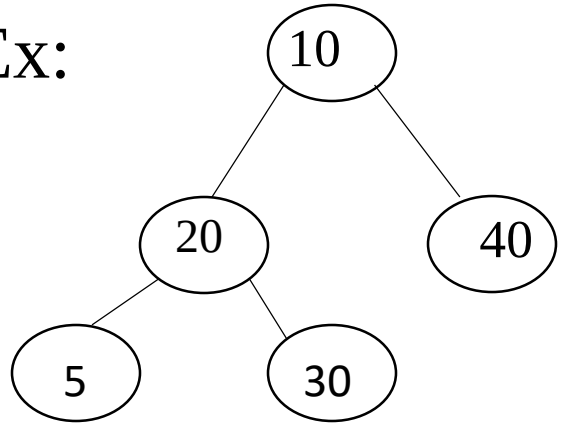
Pop 30, print it and move right.

Pop 10, print it and move right and push 40 and so on

```
/*function for iterative inorder traversal*/
Void inorder(NODEPTR root)
{
  NODEPTR cur;
  NODEPTR stack[20];
  int top=-1;
  if(root==NULL)
  {
        printf(“tree is empty”);
        return;
  }
```

```
cur=root;
for(; ;)
{ while(cur!=NULL)
  {
        push(cur,&top,s)          /*push the node to stack*/
        cur=cur→llink;
  }
  if(!stack_empty(top))           /*more nodes existing*/
  {
        cur=pop(&top,s);          /* pop most recent node*/
        printf("%d", cur→info);
        cur=cur→rlink;            /*traverse right*/
  }
  else return;
}}
```

Ex:



1. <u>After 1st iteration of while loop</u>

Node 10 is pushed to stack

Cur=cur→llink; i.e Cur=20

10

2. <u>After 2nd iteration of while loop</u>

Node 20 is pushed to stack

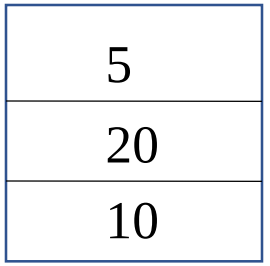
Cur=cur→llink; i.e Cur=5

20
10

3. After 3rd iteration of while loop

Node 5 is pushed to stack

Cur=cur→llink; i.e Cur=NULL



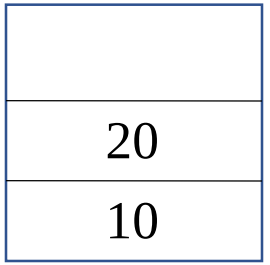
4. While loop terminates since cur==NULL

Stack is not empty

cur=pop( );i.e cur=node 5;

Print 5

cur=cur→rlink; i.e cur=NULL



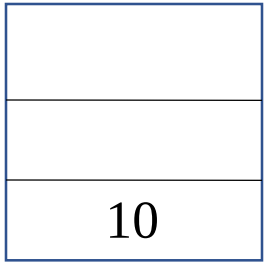
5. cur==NULL, while loop not entered

Stack not empty

cur=pop( ); i.e cur=node 20

Print 20

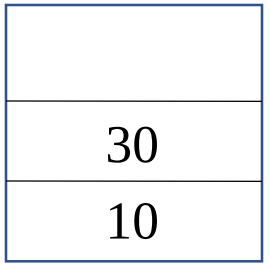
cur=cur→rlink; i.e cur=30



6. While loop is entered

Node 30 is pushed to stack

Cur=cur→llink; i.e cur=NULL



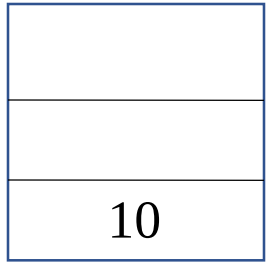
7. While loop terminates since cur==NULL

Stack not empty

cur=pop( );i.e cur=node 30;

Print 30

cur=cur→rlink; i.e cur=NULL



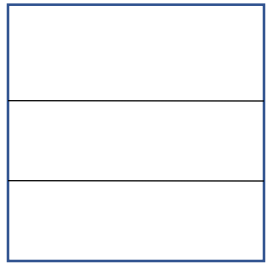
8. cur==NULL, while loop not entered

Stack not empty

cur=pop( ); i.e cur=node 10

Print 10

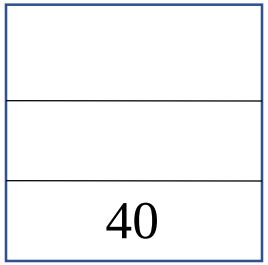
cur=cur→rlink; i.e cur=40



9. While loop is entered

Node 40 is pushed to stack

cur=cur→llink; i.e cur=NULL



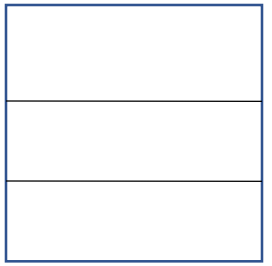
10. While loop terminates since cur==NULL

Stack not empty

cur=pop( );i.e cur=node 40;

Print 40

cur=cur→rlink; i.e cur=NULL



11. cur==NULL and stack empty

return

Hence elements printed are: 5 20 30 10 40

Iterative postorder traversal

- Here a flag variable to keep track of traversing. Flag is associated with each node. Flag==-1 indicates that traversing right subtree of that node is over.

Algorithm:

- Traverse left and push the nodes to stack with their flags set to 1, until NULL is reached.
- Then flag of current node is set to -1 and its right subtree is traversed. Flag is set to -1 to indicate that traversing right subtree of that node is over.
- Hence is flag is -1, it means traversing right subtree of that node is over and you can print the item. if flag is not –ve, traversing right is not done, hence traverse right.

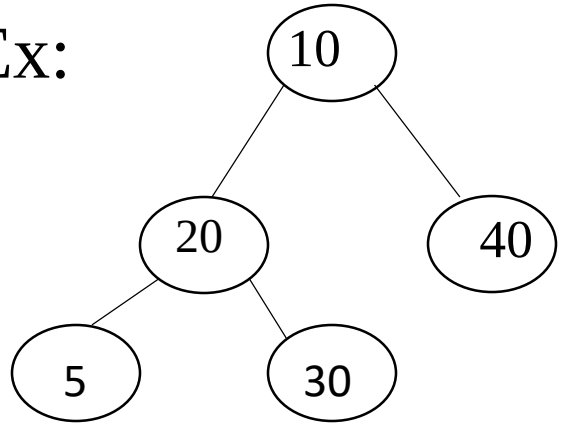
```
/*function for iterative postorder traversal*/
Void postorder(NODEPTR root)
{
  struct stack
  {
        NODEPTR node;
        int flag;
  };
  NODEPTR cur;
  struct stack  s[20];
  int top=-1;
  if(root==NULL)
  {
        printf(“tree is empty”);
        return;
  }
```

```
cur=root;
for(; ;)
{  while(cur!=NULL)
   {        s[++top].node=cur;          /*traverse left of tree and push  the nodes to the stack and
            s[top].flag=1;                          set flag to 1*/
            cur=cur→llink;
   }
   while(s[top].flag<0)
   {
            cur=s[top--].node;          /*if flag is –ve, right subtree  is visited and hence node
            printf("%d", cur→info);              is poped and printed*/
            if(stack_empty(top))                 /*if stack is empty, traversal  is complete*/
            return;
}
```

```
    cur= s[top].node;          /*after left subtree is
   cur=cur→rlink;               traversed, move to right and
   s[top].flag=-1;              set its flag to -1 to indicate
                                right subtree is traversed*/

}
}
```

Ex:



Initially cur =10;

1. <u>After 1st iteration of 1st while loop</u>

Node 10 is pushed to stack & its flag set to 1.

Cur=cur→llink; i.e Cur=20

10	1

2. <u>After 2nd iteration of 1st while loop</u>

Node 20 is pushed to stack & its flag set to 1

Cur=cur→llink; i.e Cur=5

20	1
10	1

3. After 3rd iteration of 1st while loop

Node 5 is pushed to stack & its flag set to 1.

Cur=cur→llink; i.e Cur=NULL

5	1
20	1
10	1

4. While loop terminates since cur==NULL

Since s[top].flag!=-1, 2nd while not entered.

cur=s[top].node; i.e cur=node 5;

Cur=cur→rlink; i.e cur=NULL;

s[top].flag =-1

5	-1
20	1
10	1

5. cur==NULL, 1st while loop not entered

Since s.[top].flag<0, 2nd while is entered.

cur=s[top].node; i.e cur=node 5;

Print 5

Stack is not empty, continue;

20	1
10	1

6. s[top].flag!=-1, 2nd While loop is exited

Cur=s[top].node; i.e cur=20;

Cur=cur→rlink; i.e cur=30;

s[top].flag =-1

20	-1
10	1

7. 1st while is entered

Node 30 is pushed to stack & its flag set to 1.

Cur=cur→llink; i.e Cur=NULL

30	1
20	-1
10	1

8. cur==NULL, 1st while loop exits

Since s[top]!=-1, 2nd while not entered.

cur=s[top].node; i.e cur=30;

cur=cur→rlink; i.e cur=NULL;

s[top].flag =-1

30	-1
20	-1
10	1

9. cur==NULL, 1st while loop not entered

Since s.[top].flag<0, 2nd while is entered.

cur=s[top].node; i.e cur=node 30;

Print 30

Stack is not empty, continue;

20	-1
10	1

10. s[top].flag<0, 2nd While loop continues

cur=s[top].node; i.e cur=node 20;

Print 20

Stack is not empty, continue;

10	1

11. s[top].flag!=-1, 2nd While loop is exited

Cur=s[top].node; i.e cur=10;

Cur=cur→rlink; i.e cur=40;

s[top].flag =-1

10	-1

12. 1st while is entered

Node 40 is pushed to stack & its flag set to 1.

Cur=cur→llink; i.e Cur=NULL

40	1
10	-1

13. cur==NULL, 1st while loop exits

Since s[top]!=-1, 2nd while not entered.

cur=s[top].node; i.e cur=40;
cur=cur→rlink; i.e cur=NULL;
s[top].flag =-1

40	-1
10	-1

14. cur==NULL, 1st while loop not entered

Since s.[top].flag<0, 2nd while is entered.

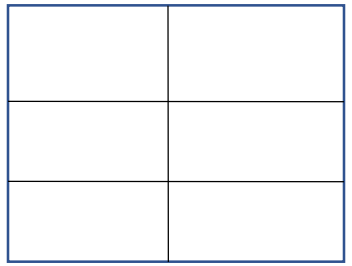
cur=s[top].node; i.e cur=node 40;

Print 40

Stack is not empty, continue;

10	-1

15. s[top].flag<0, 2nd While loop continues



cur=s[top].node; i.e cur=node 10;

Print 10

Stack is empty, stop;

Hence elements printed in postorder are: 5, 30, 20, 40, 10

Thank You.