

# Searching Algorithms

Dr. Amit Praseed

# The Concept of Searching

- Searching data involves determining whether a value (search key) is present in the data and, if so, finding the value's location.
- For a searching algorithm S:
  - Input: A list of items  $[a_1, a_2, a_3 \dots a_n]$ , and a search key  $k$
  - Output:
    - A Boolean (True/False) value indicating whether the element was found or not
    - The position at which the element was found

# A Simple Approach – Linear Search

- The simplest searching algorithm involves inspecting every element in the list and comparing it with the search key
- Assumptions:
  - The list indexing is from L to U, where  $0 \leq L \leq U$
  - The *LINEAR\_SEARCH* function takes as arguments a list  $A[L..U]$  and the search key
  - The *LINEAR\_SEARCH* function returns
    - the position of the element, if the element is found
    - -1, if the element is not found (**will this work if assumption 1 is not enforced?**)

# Linear Search Algorithm

***LINEAR\_SEARCH (A[L..U], key)***

*i* = *L*

***while i < U, do***

{

***if A[i] == key***

***return i;***

*i*++;

}

***return -1***

# Complexity of Linear Search

- **Best Case:**
  - $key == A[L]$
  - Complexity:  $O(1)$
- **Worst Case:**
  - $key == A[U-1]$  or element is absent
  - Complexity:  $O(n)$  [Assuming that  $L - U + 1 = n$ ]
- **Average Case:**
  - Complexity:  $O(n)$  [How?]

# Searching with Input Constraints

- The input list to the linear search algorithm is **unconstrained**
  - It can be any unsorted array
- If the input array is sorted, then we can reduce the complexity of searching
  - **Binary Search**
  - Instead of searching for *key* throughout the list, simply search for it in the part where it is most likely to be found

# Binary Search

- Assume that the key to be searched in the array is 23
- Seeing the distribution of the array, it seems much more efficient to search near the end of the array than near the beginning
- But the issue is, we don't know the contents of the array in most cases
  - Perhaps 23 could be the first element also

2	3	5	6	8	10	13	16	17	21	23	25	26	28	31
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

# Binary Search

- The idea behind binary search is :
  - We split the array into two halves, and search only one half depending on where the key is likely to be found
  - We perform this splitting operation repeatedly
  - “Divide and Conquer” approach
- How do we split?
  - Select the middle element of the array
  - If  $\text{key} > \text{middle element}$ , it is likely to be in the **second half**
  - If  $\text{key} < \text{middle element}$ , it is likely to be in the **first half**
  - If  $\text{key} = \text{middle element}$ , our job is done

# Binary Search Algorithm

```
BINSEARCH (arr, beg, end, key)
  while beg<=end, do
    mid = (beg+end)/2
    if arr[mid]==key
      return mid
    else if arr[mid]>key
      end=mid-1
    else
      beg=mid+1
  return -1
```

**NOTE:** The function will be called as BINSEARCH(arr, L, U, key), where L and U are the first and last valid indices of the array

# Binary Search - Example

- Key = 23

2	3	5	6	8	10	13	16	17	21	23	25	26	28	31
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

- Array indexing is from 0 to 14, so
  - $\text{mid}=(0+14)/2=7$

2	3	5	6	8	10	13	16	17	21	23	25	26	28	31
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

- The middle element is 16, which is less than 23
  - We search in the second half of the array, by restricting the search to  $\text{arr}[\text{mid}+1 \dots \text{end}]$

2	3	5	6	8	10	13	16	17	21	23	25	26	28	31
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

# Binary Search - Example

- Now,  $\text{beg}=8$ ,  $\text{end}=14$ 
  - $\text{mid}=(8+14)/2=11$



- The middle element, 25 is greater than the key
  - We restrict our search between indices beg (8) and mid-1 (10)



- Now  $\text{beg}=8$ ,  $\text{end}=10$ 
  - $\text{mid}=9$



# Binary Search - Example

- The middle element, 21, is less than the key, 23
  - Restrict search between indices  $\text{mid}+1$  (10) and  $\text{end}$  (10)



- Now  $\text{beg}=\text{end}=10$ 
  - $\text{mid}=10$



- Here the middle element is the key that is being searched, so the algorithm returns position 10

# Complexity of Binary Search

- In the previous example, we required only 4 comparisons (for mid values 7, 11, 9 and 10)
  - For linear search we would have required 10
- Since binary search successively divides the search space by half in every iteration, the average complexity of binary search is  $O(\log_2 n)$ 
  - What are the best case and worst case complexities?
- However, **binary search does have the handicap that it requires input to be sorted**

# Calculating Time complexity

- Let say the iteration in Binary Search terminates after  $k$  iterations. In the above example, it terminates after 3 iterations, so **here  $k = 4$**
- At each iteration, the array is divided by half. So let's say the length of array at any iteration is  $n$ .
  - At **Iteration 1**,

Length of array =  $n$

- At **Iteration 2**,

Length of array =  $n/2$

- At **Iteration 3**,

Length of array =  $(n/2)/2 = n/2^2$

- Therefore, after **Iteration  $k$** ,

Length of array =  $n/2^k$

# Contd...

- Also, we know that after

After  $k$  divisions, the length of array becomes 1

- Therefore

$$\begin{aligned}\text{Length of array} &= n/2^k = 1 \\ \Rightarrow n &= 2^k\end{aligned}$$

- Applying log function on both sides:

$$\begin{aligned}\Rightarrow \log_2(n) &= \log_2(2^k) \\ \Rightarrow \log_2(n) &= k \log_2(2)\end{aligned}$$

- As  $(\log_a(a) = 1)$

Therefore,

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

Hence, the time complexity of Binary Search is

$$\log_2(n)$$

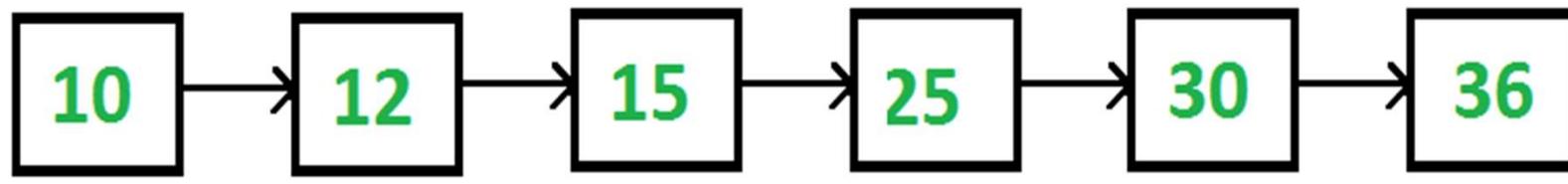
# The Tree Data Structure

Dr. Amit Praseed

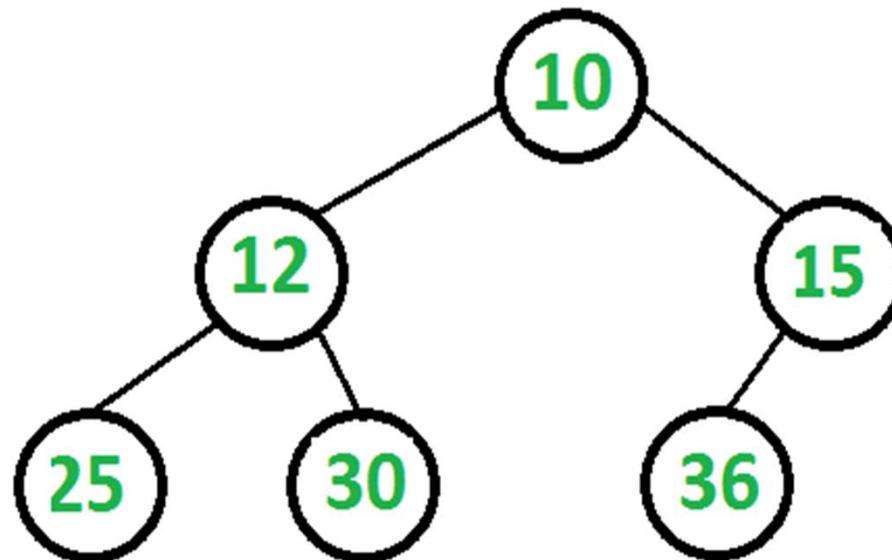
# Trees vs Linked Lists

- In a linked list, every node (except the last) is connected to exactly two nodes – a predecessor node and a successor node
- A linked list fails to efficiently represent many of the relationships that are commonly encountered in real life, for example organizational hierarchies or family trees
- In such situations, a node may need to have multiple successors, which gives rise to the concept of a tree

# Trees vs Linked Lists



Linked List

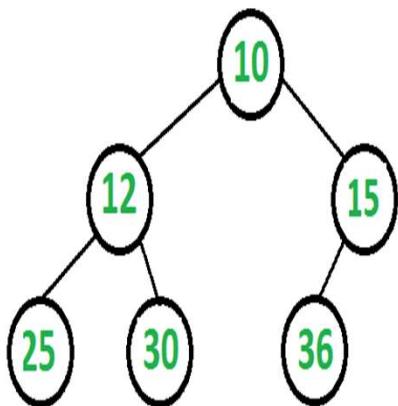


Tree

# A Formal Definition of a Tree

- A tree is a collection of nodes such that either
  - the collection is empty, or
  - the collection contains the following
    - A distinguished node  $r$  called the root
    - Zero or more non-empty subtrees  $T_1, T_2, \dots, T_k$ , each of whose roots are connected by an edge from  $r$
- A tree is a collection of  $N$  nodes, one of which is a designated node called the root, and contains  $N-1$  edges

# Tree Terminology



- Node 10 is called **the root** of the tree
- Nodes 12 and 15 are **the children** of node 10
- Node 10 is the **parent** of nodes 12 and 15
- **Subtree** of a node: A tree whose root is a child of that node
- **Depth of a node**: Number of edges from **the node to the tree's root node**.
- **Height of a node**: Number of edges on the **longest path** from the node to a leaf.
- **Height of a Tree = Height of Root**

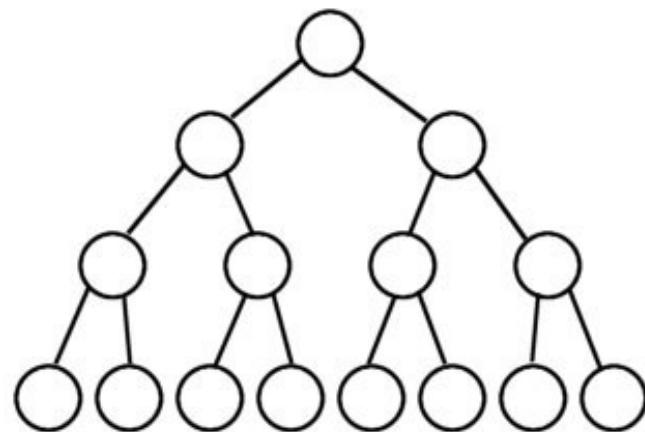
# Binary Trees

- Binary tree: a node has atmost 2 non-empty subtrees
- Set of nodes T is a binary tree if either of these is true:
  - T is empty
  - Root of T has two subtrees, both of which are binary trees

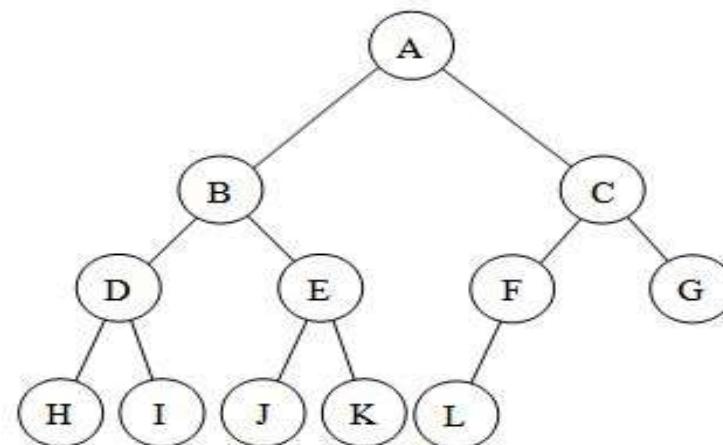
# Complete and Full Binary Trees

- A full binary tree is a tree in which **every node other than the leaves has two children.**
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- **Perfect Binary Tree??**

**Full Binary Tree**

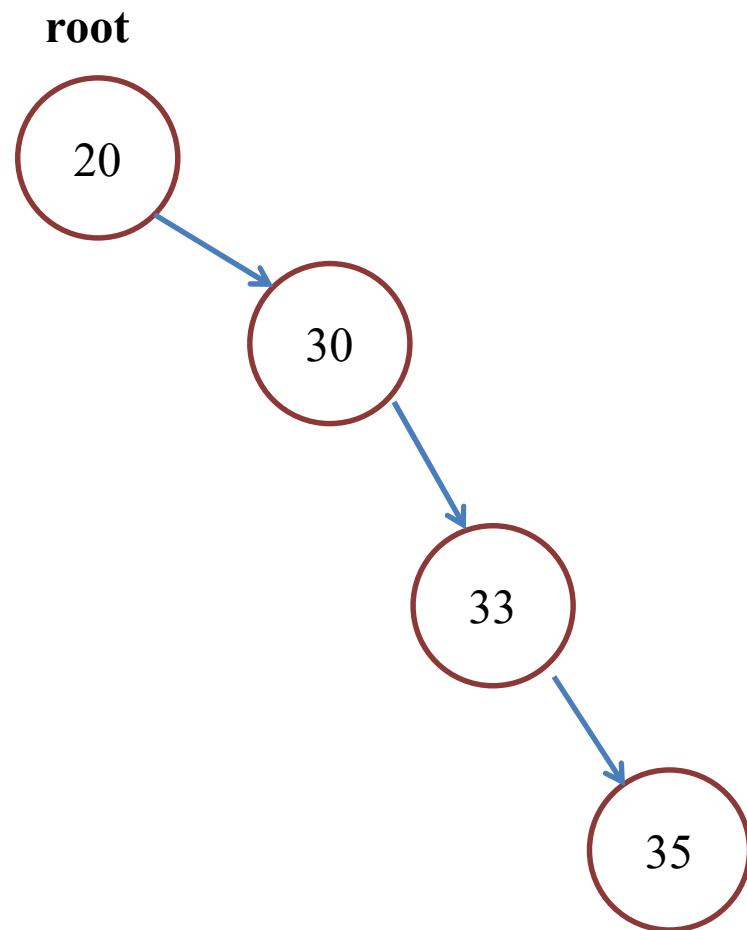


**Complete Binary Tree**



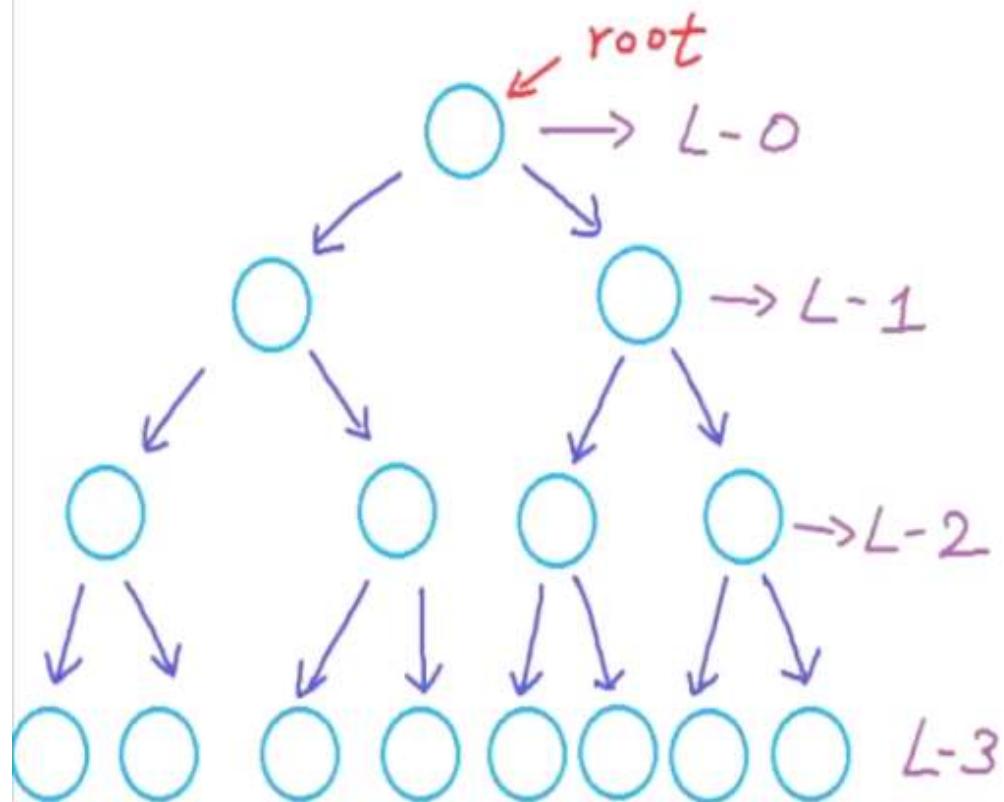
# Height of a Binary Tree

- The height of a full or complete binary tree with  $n$  nodes will be **log n** [Why?]
- However, in the worst case, a binary tree may degenerate into a linked list
  - Such a tree is called a **skewed tree**
  - Height of a completely skewed tree will be  **$n-1$**
- Thus the height of a binary tree is  **$[n-1, \log n]$**



A right skewed (binary) tree

# Height of Perfect Binary Tree



Perfect Binary tree

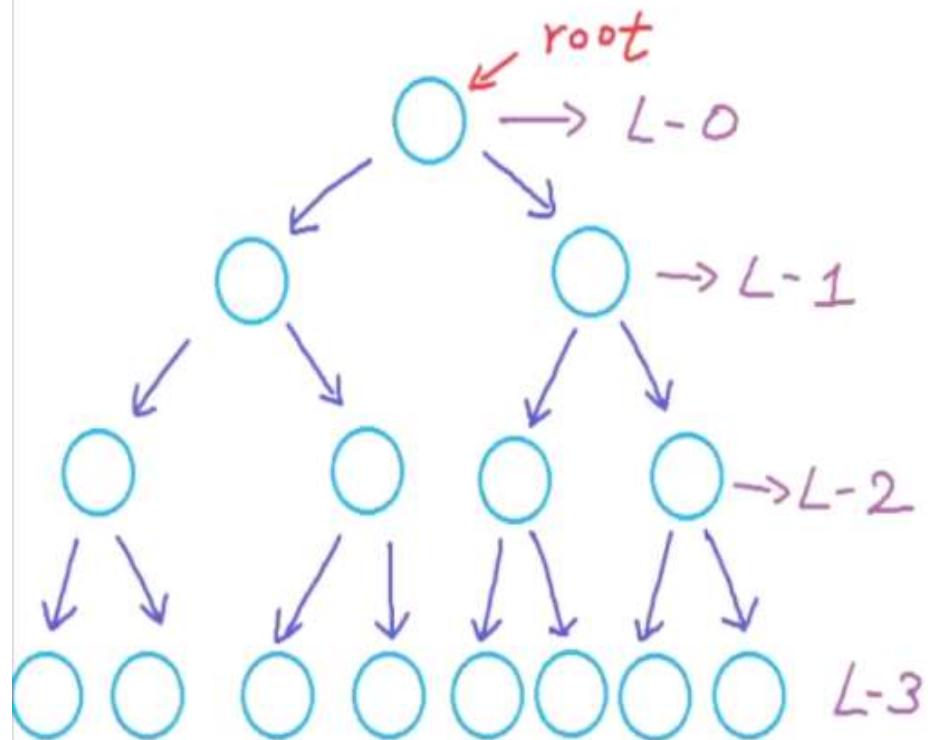
Maximum no. of nodes  
in a <sup>binary</sup> tree with height  $h$

$$= 2^0 + 2^1 + \dots + 2^h$$

$$= 2^{h+1} - 1$$

$$= 2^{(\text{no. of levels})}$$

# Contd...



Perfect Binary tree

Maximum no. of nodes  
in a <sup>binary</sup> tree with height  $h$

$$= 2^0 + 2^1 + \dots + 2^h$$

$$= 2^{h+1} - 1 \quad n = \text{no. of nodes}$$

$$n = 2^{h+1} - 1 \rightarrow$$

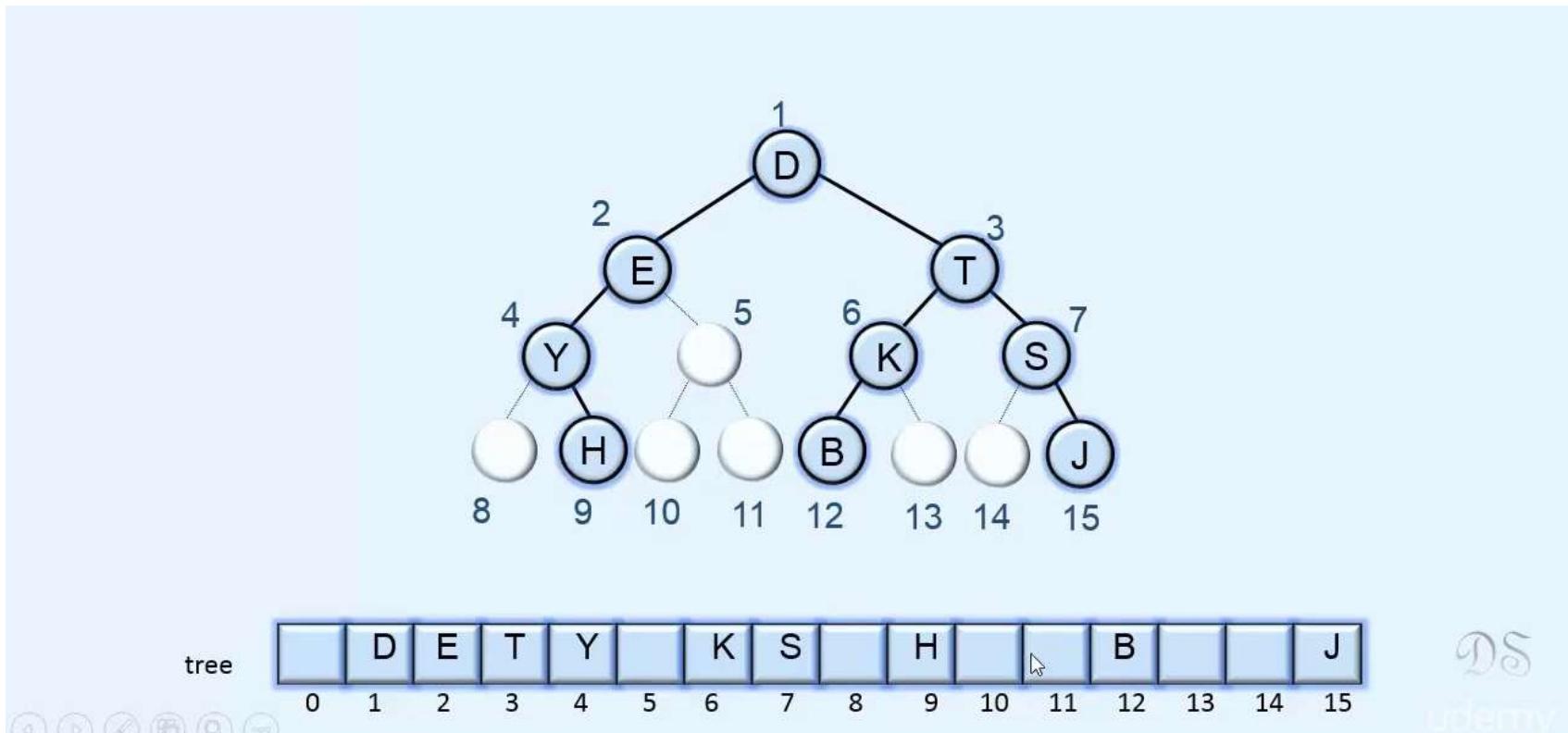
$$\Rightarrow 2^{h+1} = (n+1)$$

$$\Rightarrow h = \log_2(n+1) - 1$$

Activate Windows  
Go to settings to activate Windows.

Height of Perfect Binary Tree =  $\log_2(n)$

# Array Representation of Binary Trees



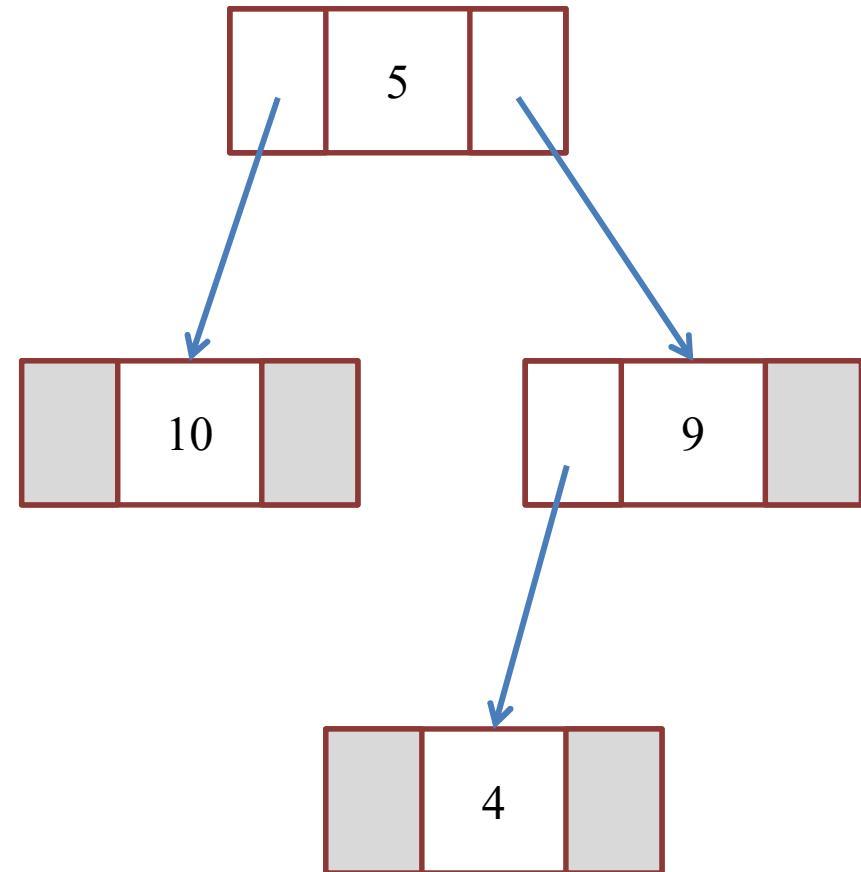
Parent of node at location  $i$  is at location  $i/2$

Children of a node at location  $i$  are at locations  $2*i$  and  $2*i+1$

# Binary Trees using Pointers

- Every node within a binary tree can be represented as a structure

```
struct node  
{  
    int data;  
    struct node *lchild;  
    struct node *rchild;  
}
```



# Traversing a Binary Tree

- A binary tree can be traversed by simply following its lchild and rchild pointers
- This is not as trivial as a linked list, which has only one possible order of traversal
- A binary tree has multiple ways of traversal, depending upon whether the node, left child or right child are explored first
  - Inorder Traversal
  - Preorder Traversal
  - Postorder Traversal

# In-order Traversal

- During the in-order traversal algorithm, the left subtree is explored first, followed by root, and finally nodes on the right subtree.

*INORDER (root)*

*INORDER(root → lchild)*

*PROCESS(root)*

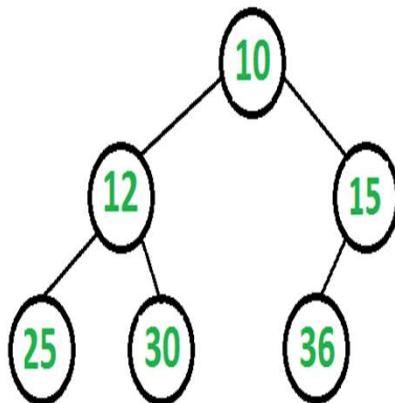
*INORDER(root → rchild)*

# Post-order and Pre-order Traversal

- During the post-order traversal algorithm, the **left subtree is explored first**, followed by nodes on the **right subtree**, and finally the **root**.
- During the pre-order traversal algorithm, the **root** is explored first, followed by nodes on the **left subtree**, and finally the nodes on the **right subtree**.

**EXERCISE:** Write the recursive algorithm for post-order and pre-order traversal algorithms

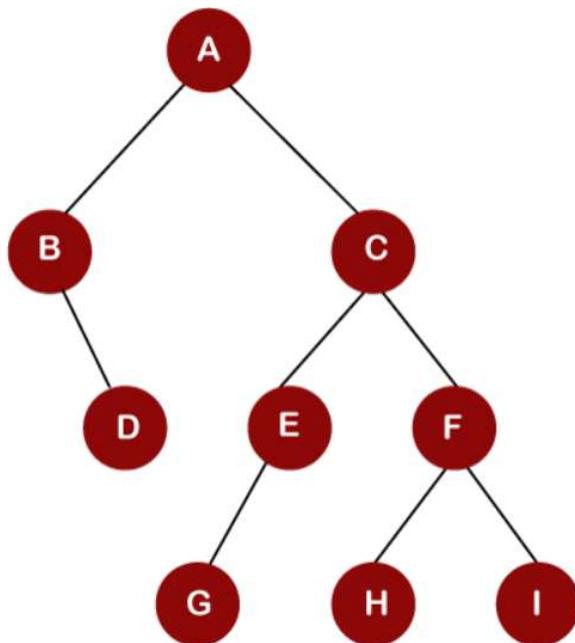
# Examples



In-order – 25 12 30 10 36 15

Pre-order – 10 12 25 30 15 36

Post-order – 25 30 12 36 15 10



In-order – B D A G E C H F I

Pre-order – A B D C E G F H I

Post-order – D B G E H I F C A

# Binary Search Tree (BST)

- A Binary Search Tree (BST) is a binary tree which has the following special properties:
  - The left subtree of a node contains only nodes with keys lesser than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree each must also be a binary search tree.

# Inserting Elements into a BST

*INSERT\_BST(root, node)*

*if root == NULL*

*root = node*

*else*

*if node → data < root → data*

*INSERT\_BST(root → lchild, node)*

*if node → data > root → data*

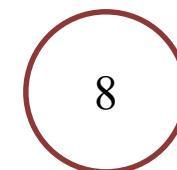
*INSERT\_BST(root → rchild, node)*

# Inserting Elements into a BST

**Current BST**

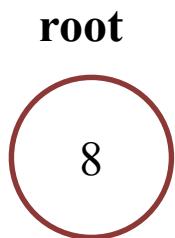
**root= NULL**

**Element to be added**

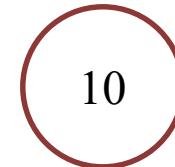


# Inserting Elements into a BST

**Current BST**

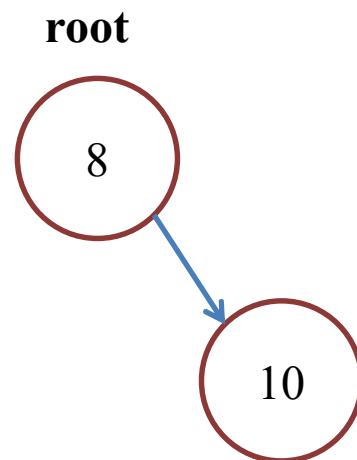


**Element to be added**

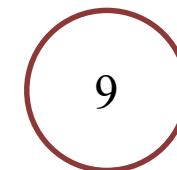


# Inserting Elements into a BST

**Current BST**

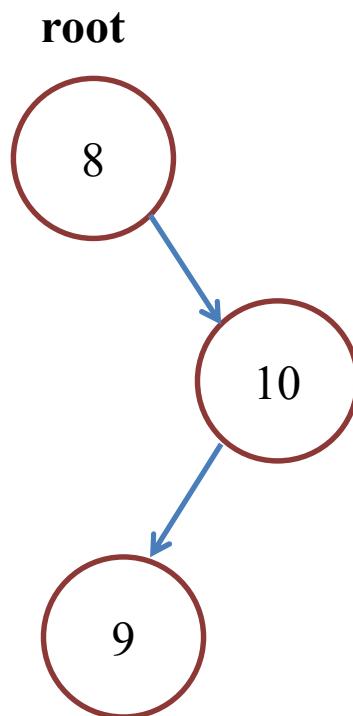


**Element to be added**

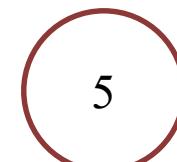


# Inserting Elements into a BST

**Current BST**

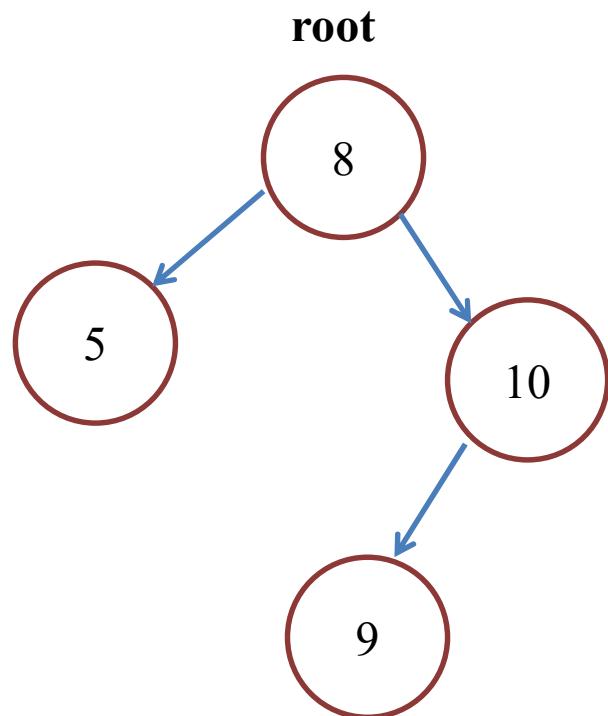


**Element to be added**

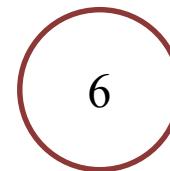


# Inserting Elements into a BST

**Current BST**

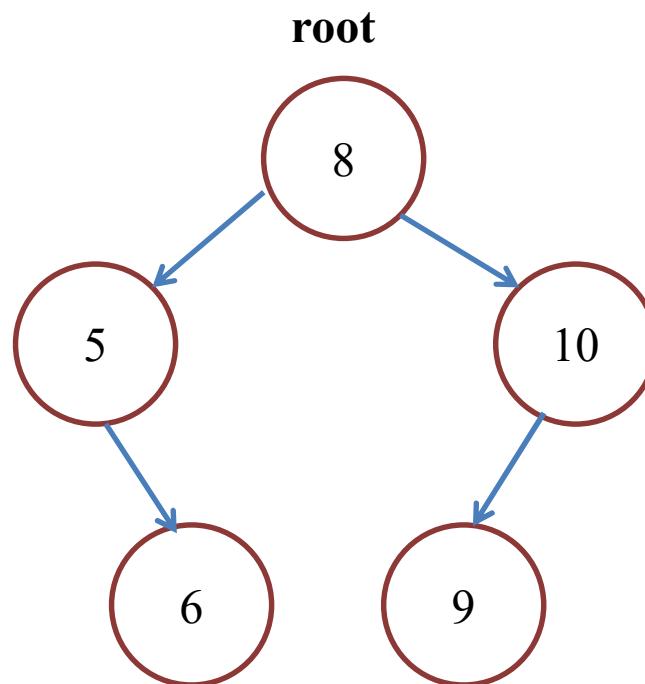


**Element to be added**



# Inserting Elements into a BST

**Current BST**



# Complexity of *INSERT\_BST*

- The *INSERT\_BST* function explores exactly one path within the tree
  - There is no backtracking or going back within the algorithm
- So the complexity of *INSERT\_BST* depends upon the maximum number of nodes in any such path within the tree
  - Which is nothing but the height of the tree
  - Complexity of *INSERT\_BST* =  $O(h)$ , where  $h$  is the height of the tree

# Searching a BST

- While searching for an element within a BST, we can employ the same strategy as in binary search
  - All elements greater than the root node are in the right subtree of the root
  - All elements greater than the root node are in the right subtree of the root
  - This definition follows recursively
  - If we reach a leaf node and still were unable to find the element, the element is not present in the BST

# Searching a BST

***SEARCH\_BST (root, key)***

*ptr = root*

***while*** *ptr* !=NULL, ***do***

***if*** *ptr* →data ==key

***return*** *ptr*

***else if*** *ptr* →data > key

*ptr* = *ptr* →lchild

***else***

*ptr* = *ptr* →rchild

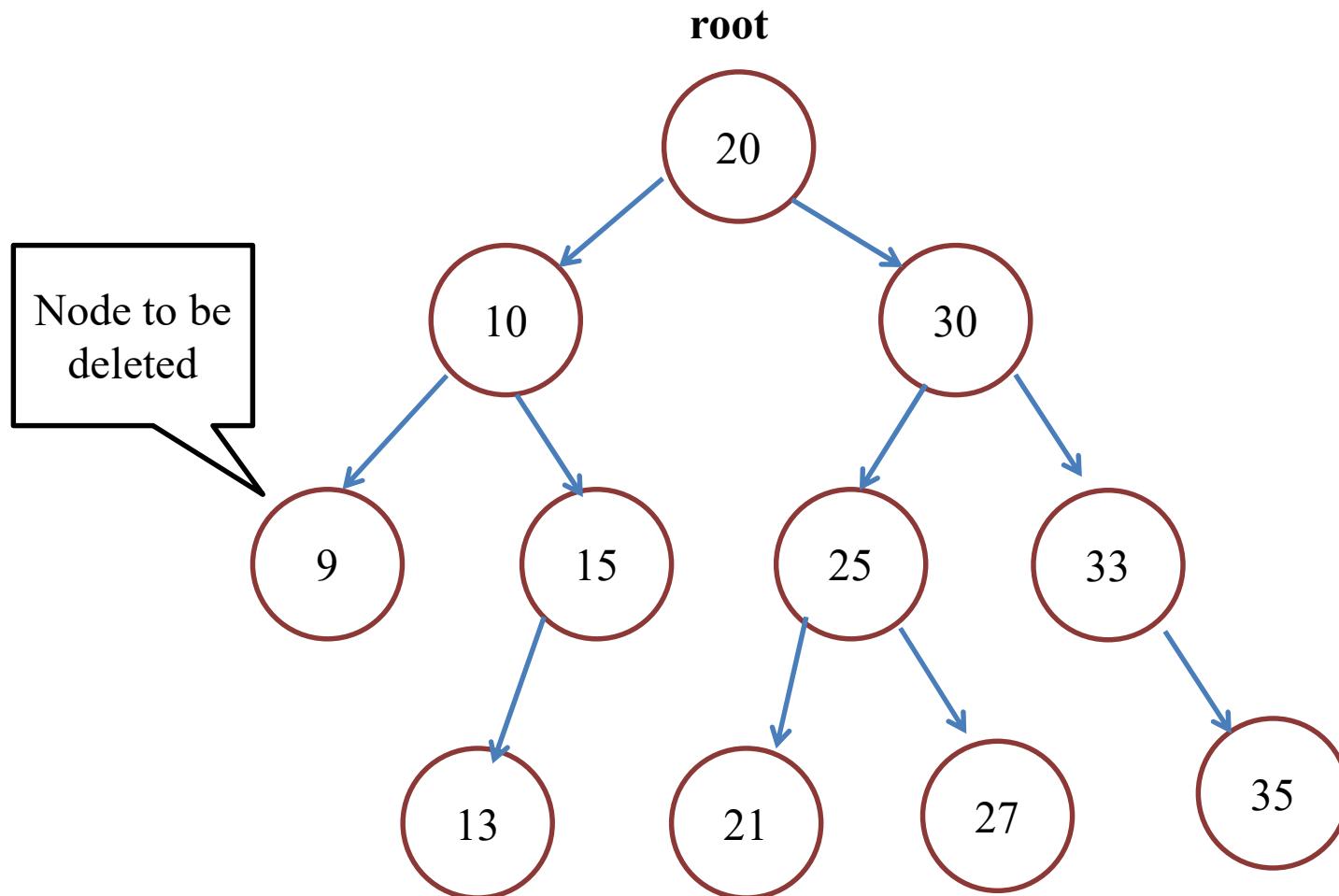
***return*** NULL

EXERCISE: What is the complexity of this algorithm?

# Deleting an Element from a BST

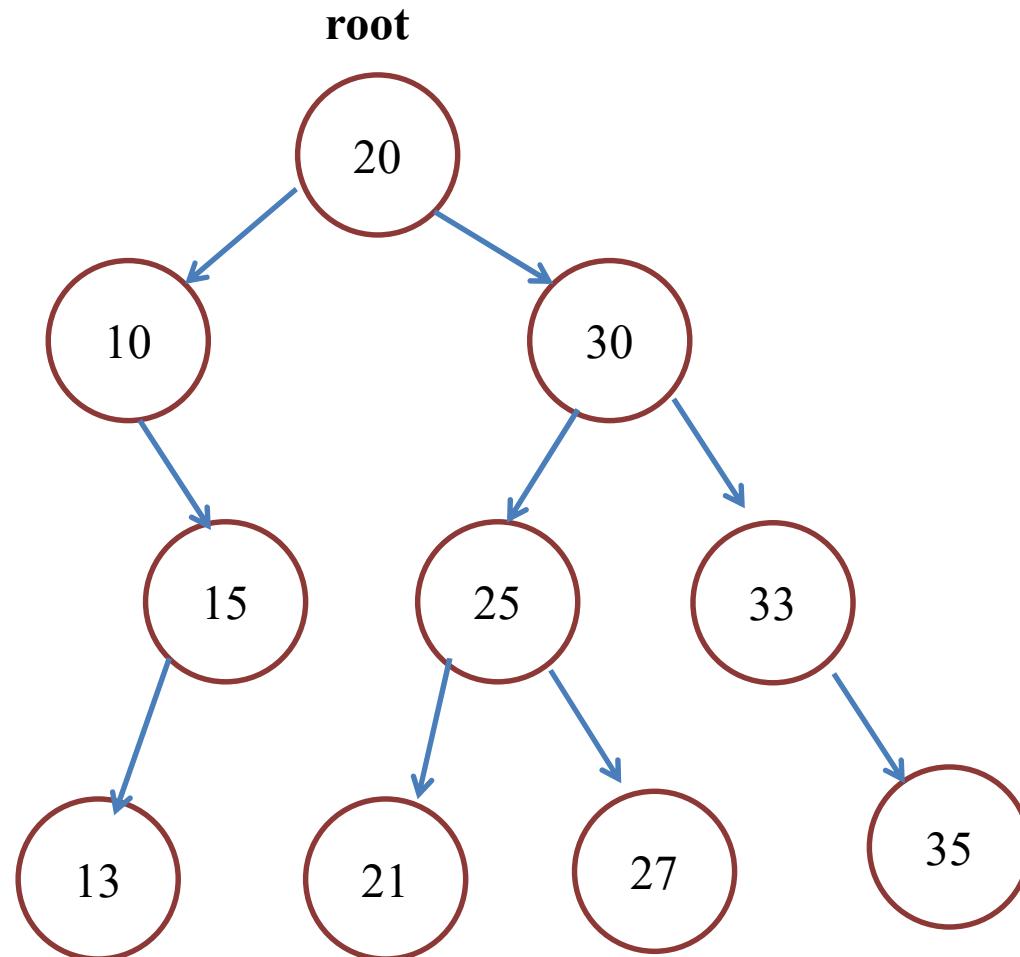
- Deletion from a BST is much more complicated than insertion or searching
  - We may need to delete a node which has children/subtrees
  - How do we delete the node and still maintain the BST property?
- Three cases
  - Deleting a leaf node – Simple, since there are no children
  - Deleting a node with one child
  - Deleting a node with two children

# Deleting an Element from a BST

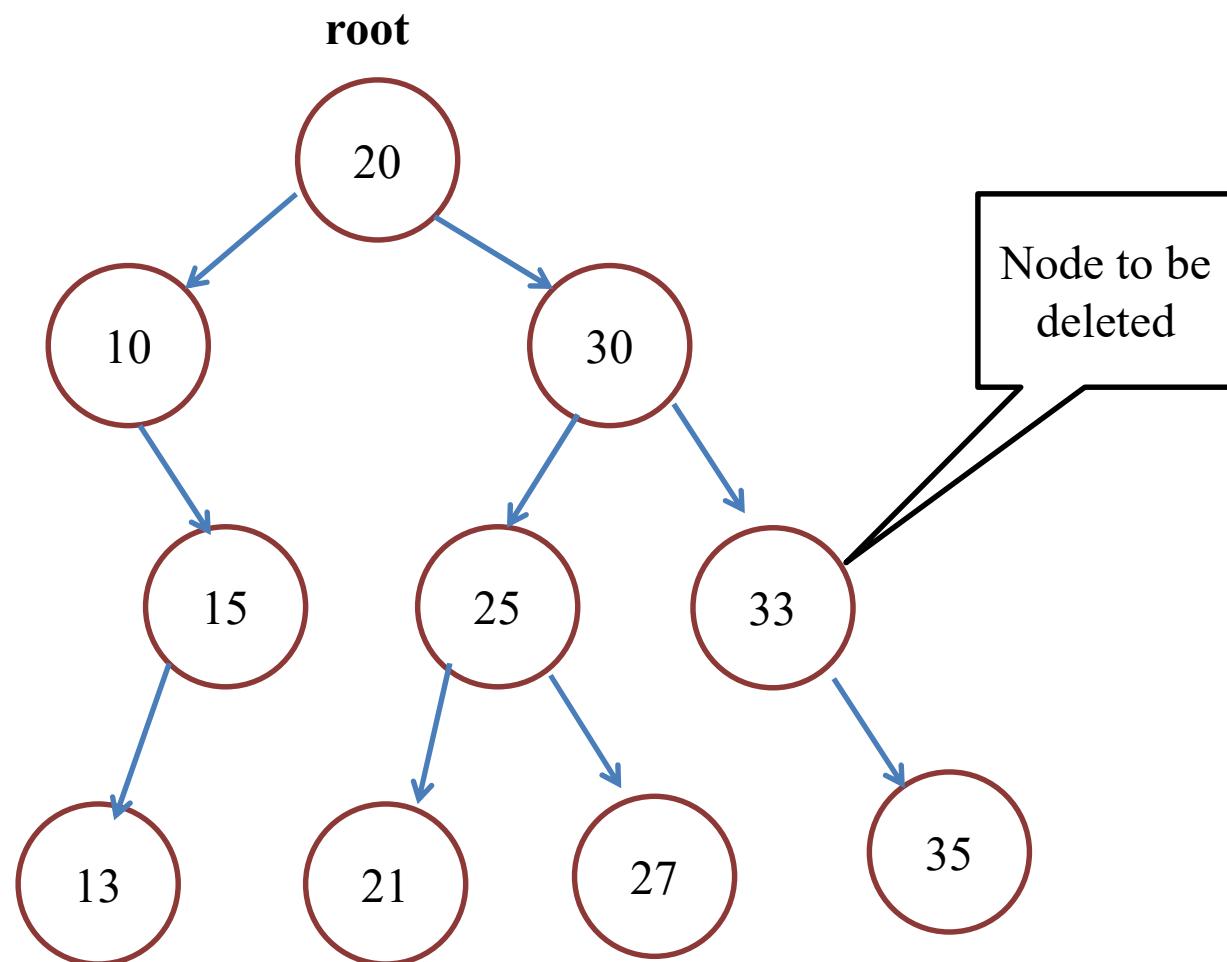


# Deleting an Element from a BST

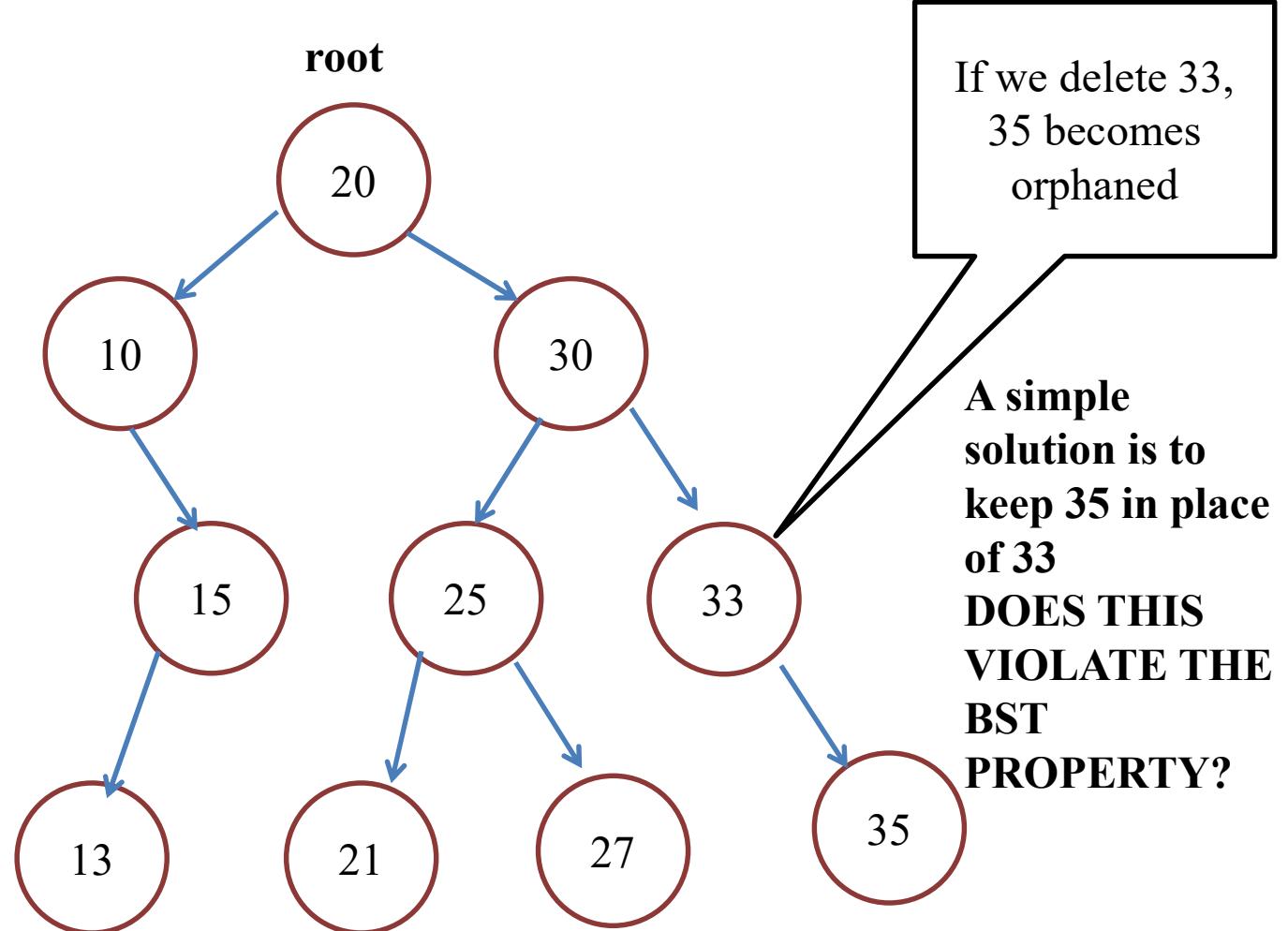
**Simple!**  
Just set the  
parent's child  
pointer as  
**NULL** and  
delete the node



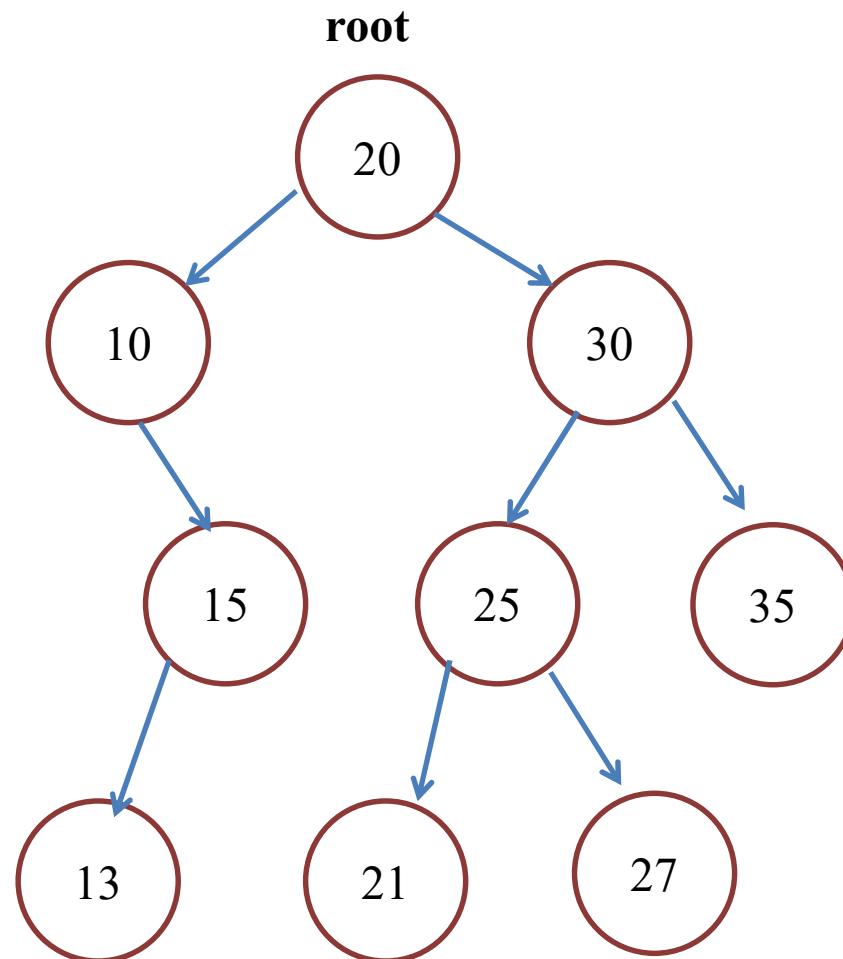
# Deleting an Element from a BST



# Deleting an Element from a BST

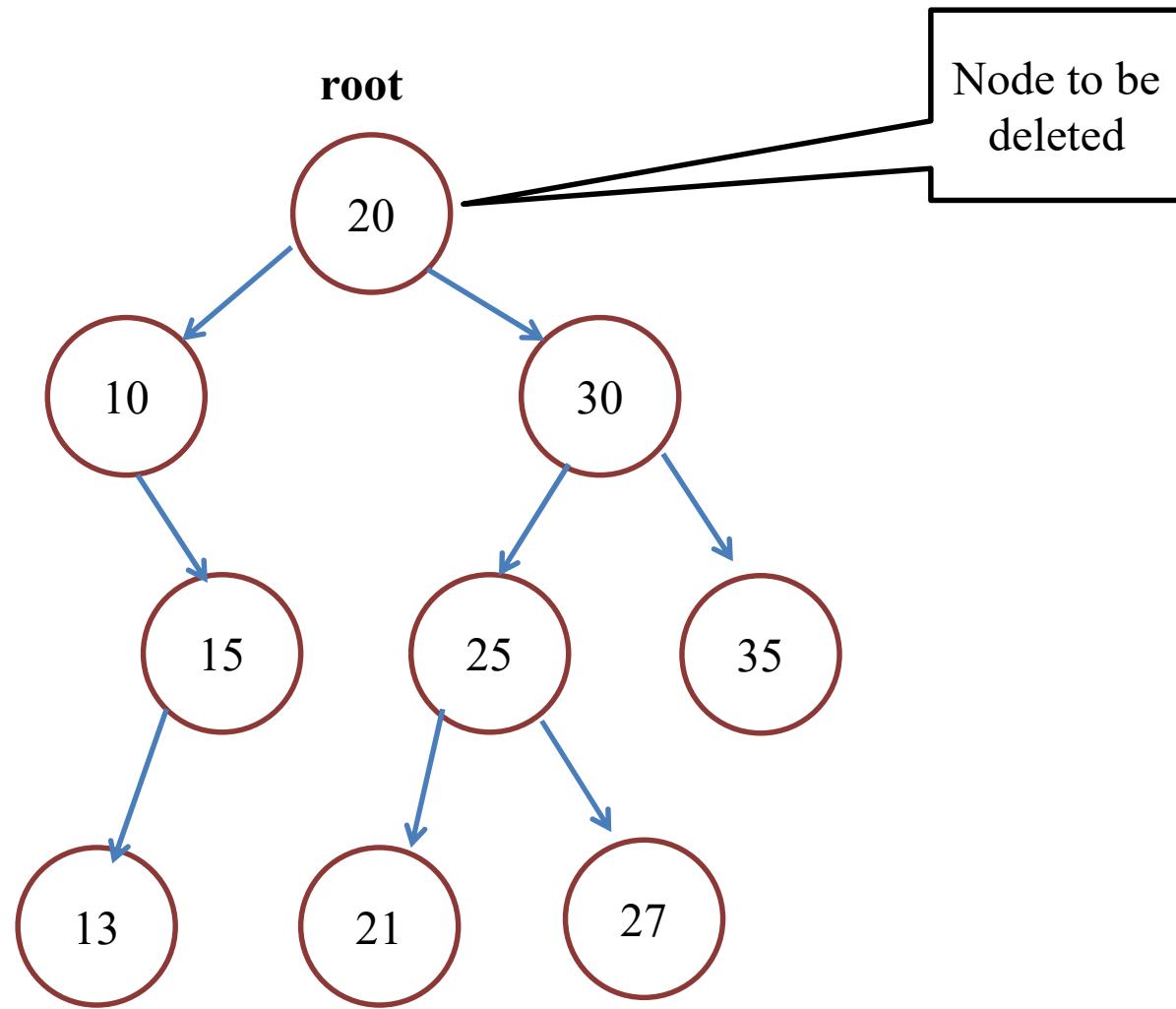


# Deleting an Element from a BST

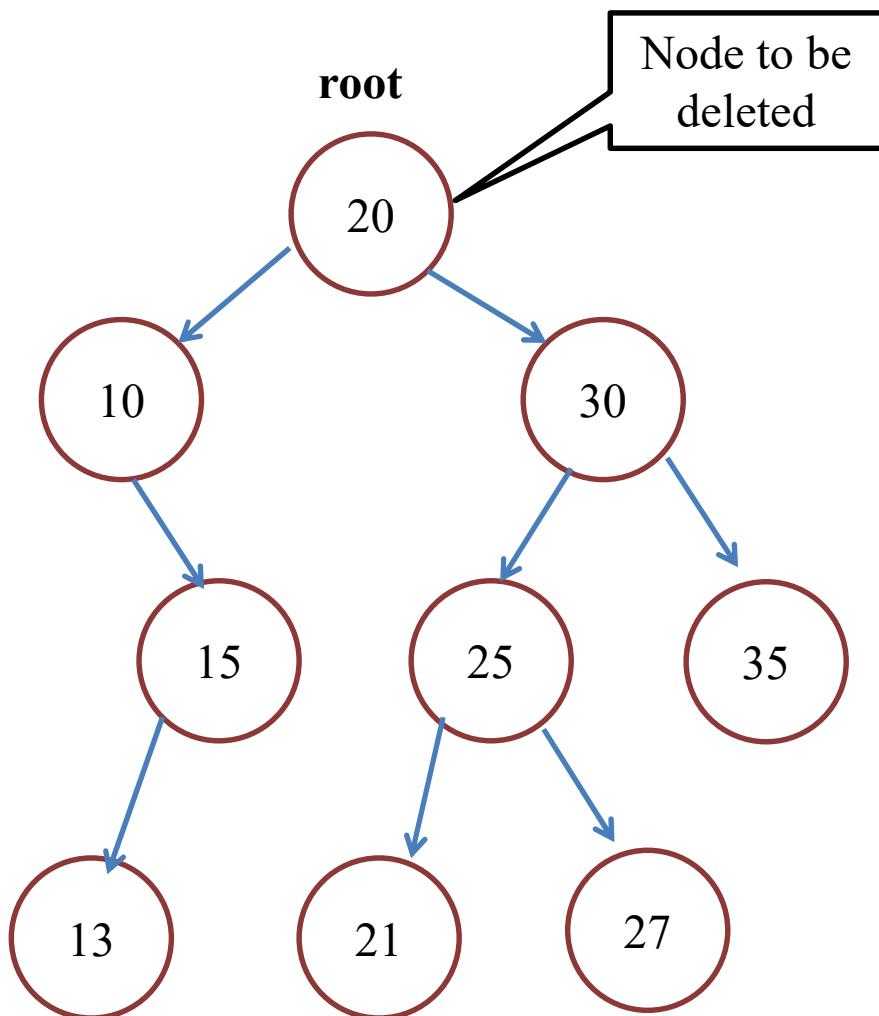


**If a node only  
has one child,  
keep the child in  
place of the node  
and delete the  
node**

# Deleting an Element from a BST



# Deleting an Element from a BST



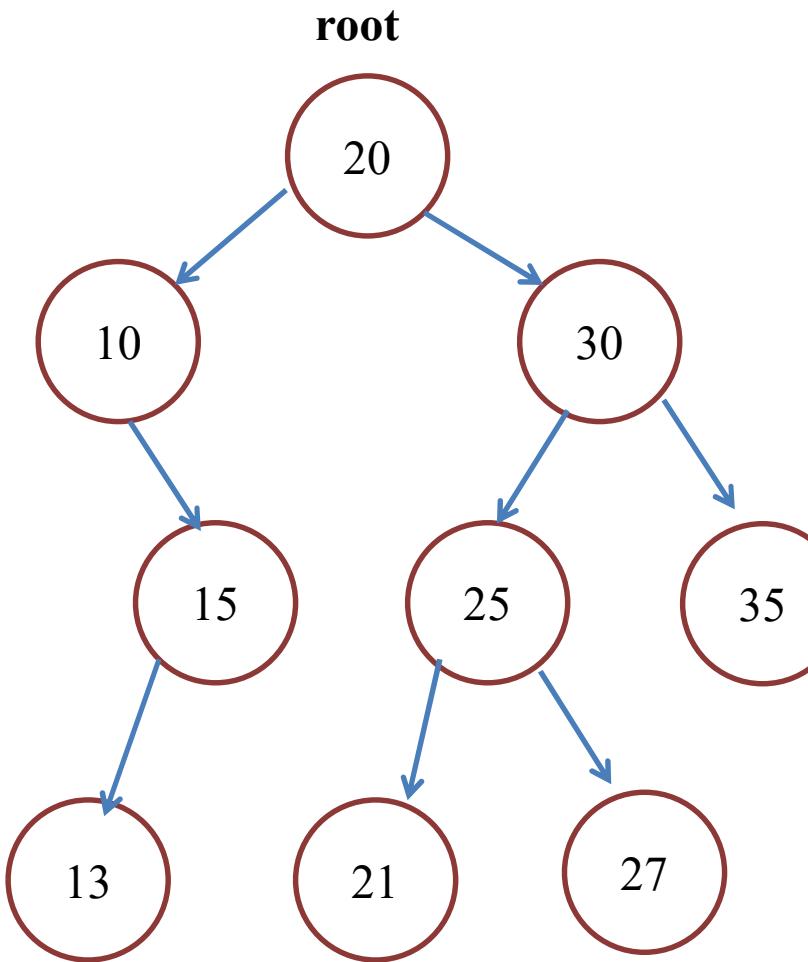
## SOLUTION:

Find a node in the BST that can take the place of node 20  
It should be greater than all other nodes in the left subtree and less than all other nodes in the right subtree once replaced

## Possible choices:

- Rightmost node in the left subtree
- Leftmost node in the right subtree

# Deleting an Element from a BST



## SOLUTION:

Find a node in the BST that can take the place of node 20

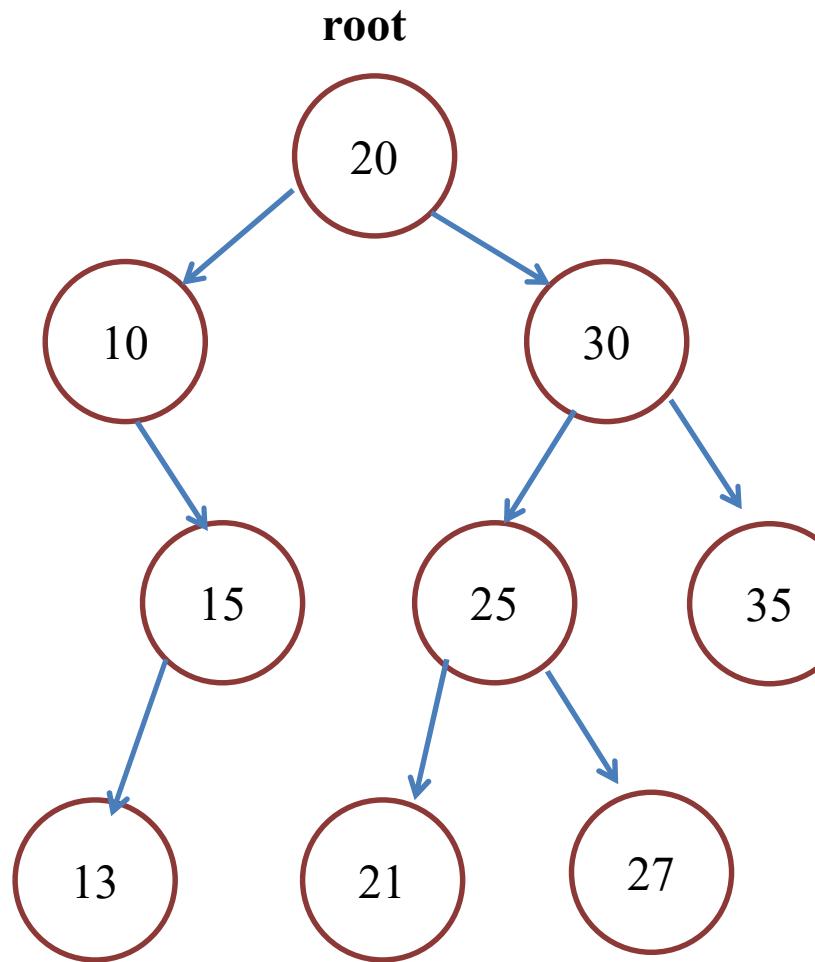
It should be greater than all other nodes in the left subtree and less than all other nodes in the right subtree once replaced

## Possible choices:

- Rightmost node in the left subtree
- Leftmost node in the right subtree

Inorder Successor of Node 20

# Deleting an Element from a BST

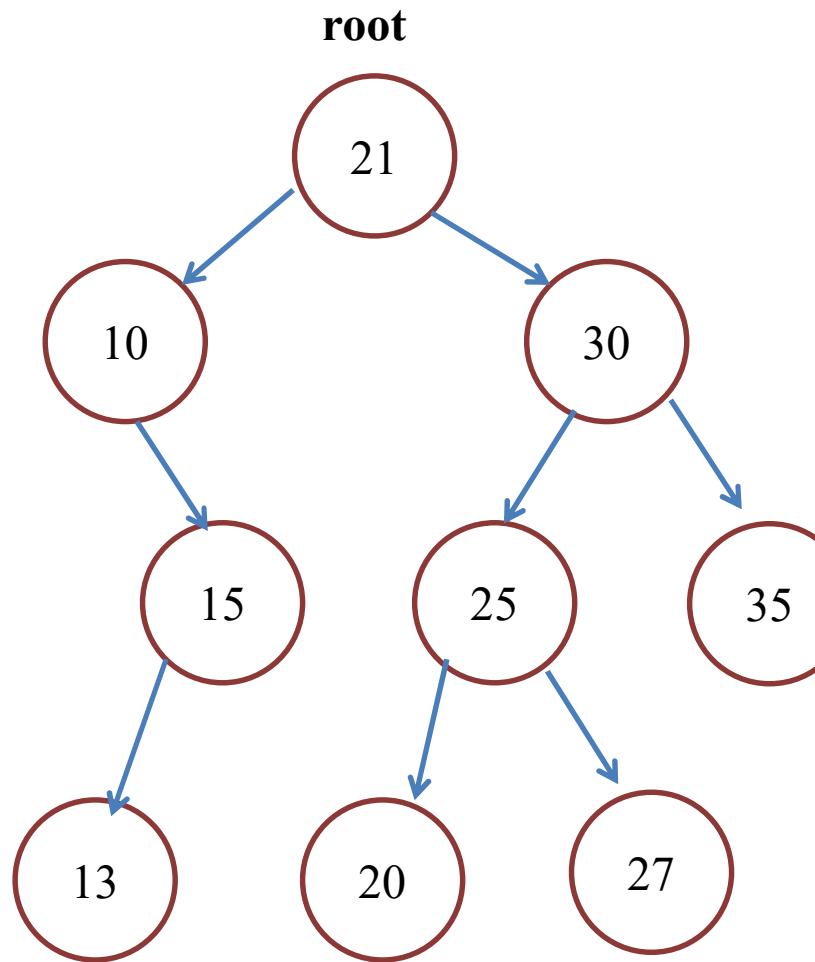


## SOLUTION:

Replace the node to be deleted with its inorder successor  
Now run the delete operation once again

Inorder Successor of Node 20

# Deleting an Element from a BST



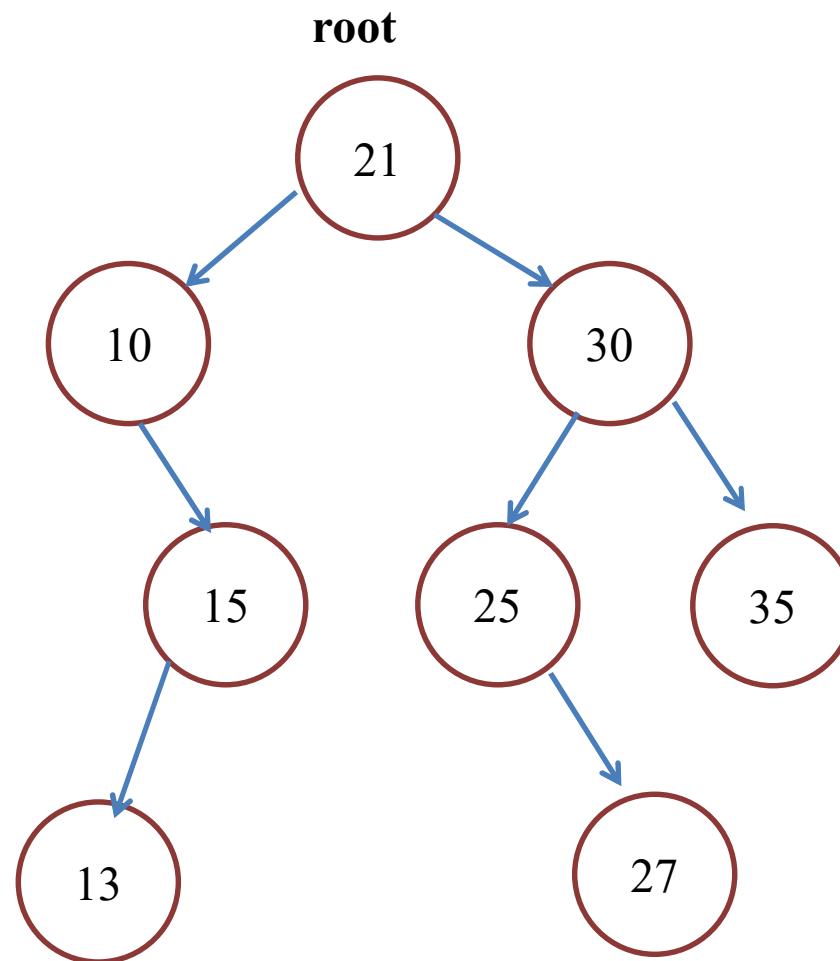
## SOLUTION:

Replace the node to be deleted with its inorder successor

Now run the delete operation once again

Now 20 is a leaf node and can be deleted easily

# Deleting an Element from a BST



# Deleting an Element from a BST

```
DELETE_BST(root, key)
    prev = curr = root
    while curr !=NULL, do
        if curr →data==key
            break
        else
            prev=curr
            if curr →data>key
                curr=curr →lchild
            else
                curr=curr →rchild
        if curr==NULL
            print “Element to be deleted does not exist”
            exit
```

# Deleting an Element from a BST

```
DELETE_BST(root, key) //Continued  
  if curr →lchild==NULL && curr →rchild==NULL  
    if curr==prev →lchild  
      prev →lchild=NULL  
    else  
      prev →rchild=NULL  
    free(curr)
```

# Deleting an Element from a BST

```
DELETE_BST(root, key)          //Continued
    else if curr →lchild==NULL || curr →rchild==NULL
        if curr →lchild==NULL
            if curr==prev →lchild
                prev →lchild=curr →rchild
            else
                prev →rchild=curr →rchild
            free(curr)
        if curr →rchild==NULL
            if curr==prev →lchild
                prev →lchild=curr →lchild
            else
                prev →rchild=curr →lchild
            free(curr)
```

# Deleting an Element from a BST

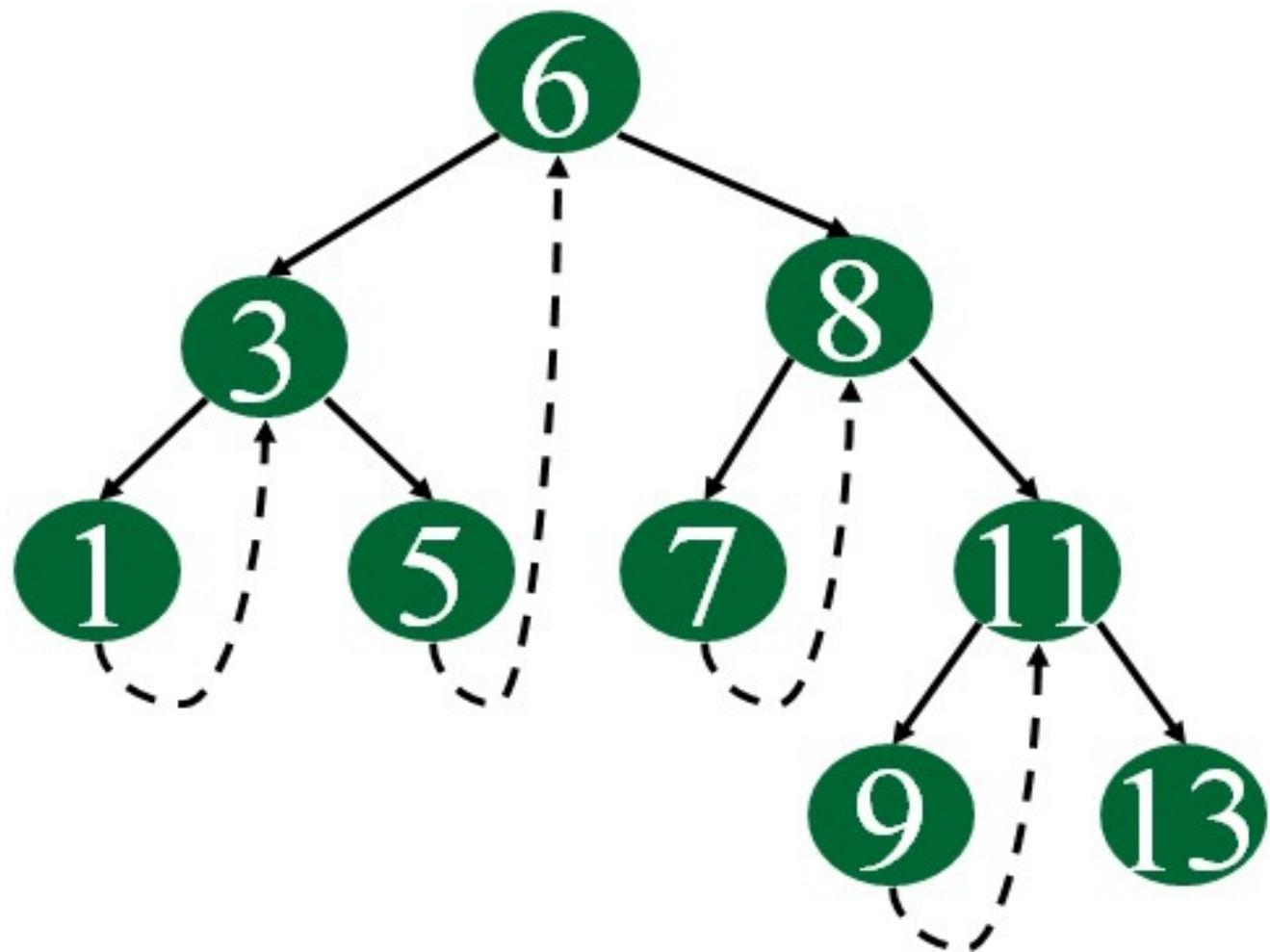
```
DELETE_BST(root, key) //Continued  
else  
    ptr=INORDER_SUCCESSOR(curr)  
    SWAP(curr →data, ptr →data)  
    DELETE_BST(root, key)
```

EXERCISE: What is the complexity of ***DELETE\_BST***?

# Threaded Binary Trees

- A threaded binary tree is a modification of a binary tree to allow in-order traversal without using recursion or stack
- Two types of threaded binary trees:
  - ***Single Threaded***: Where a NULL right pointers is made to point to the inorder successor (if successor exists)
  - ***Double Threaded***: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

# Threaded Binary Trees



# Binary Tree Construction

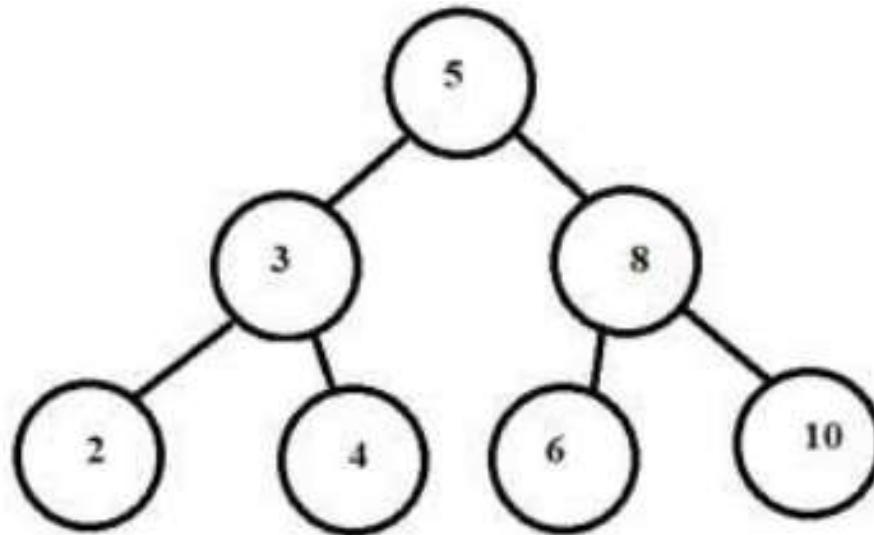
# Construct a binary search tree from inorder and preorder traversal

Inorder Traversal: 2 3 4 5 6 8 10 (Left->Root->Right)

Pre-order Traversal: 5 3 2 4 8 6 10 (Root->Left->Right)

- ❖ As we know that preorder visits the root node first then the first value always represents the root of the tree. From above sequence 5 is the root of the tree.
- ❖ *Take the elements from Pre-order (starting from left side) and find its position w.r.t Root using In-order traversal.*

# Contd...



The same approach is used for the construction of **binary tree**.

Complexity for the construction of BST =  $O(n \log n)$

Complexity for the construction of Binary Tree =  $O(n^2)$

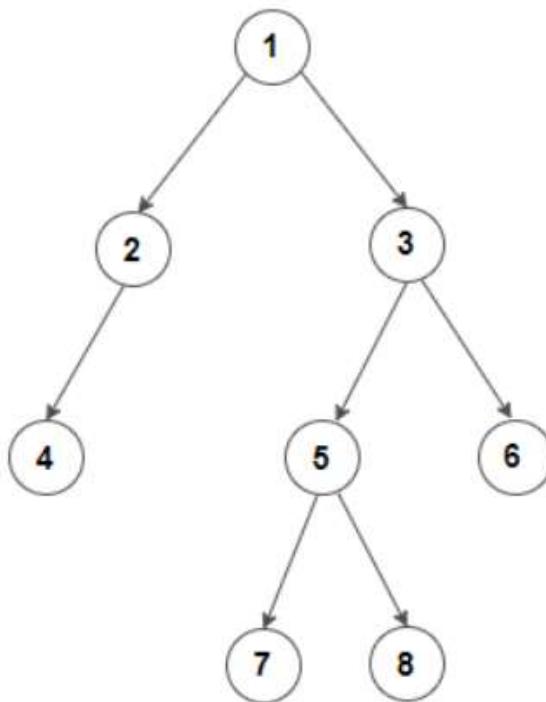
# Construct Binary Tree from Inorder and Postorder Traversal

In-order Traversal: 4, 2, 1, 7, 5, 8, 3, 6 (Left-> Root-> Right)

Post-order Traversal: 4, 2, 7, 8, 5, 6, 3, 1 (Left->Right-> Root)

- ❖ As we know that post-order visits the root node at last then the last value always represents the root of the tree. From above sequence 1 is the root of the tree.
- ❖ *Take the elements from Post-order (starting from right side) and find its position w.r.t Root using In-order traversal.*

# Contd...



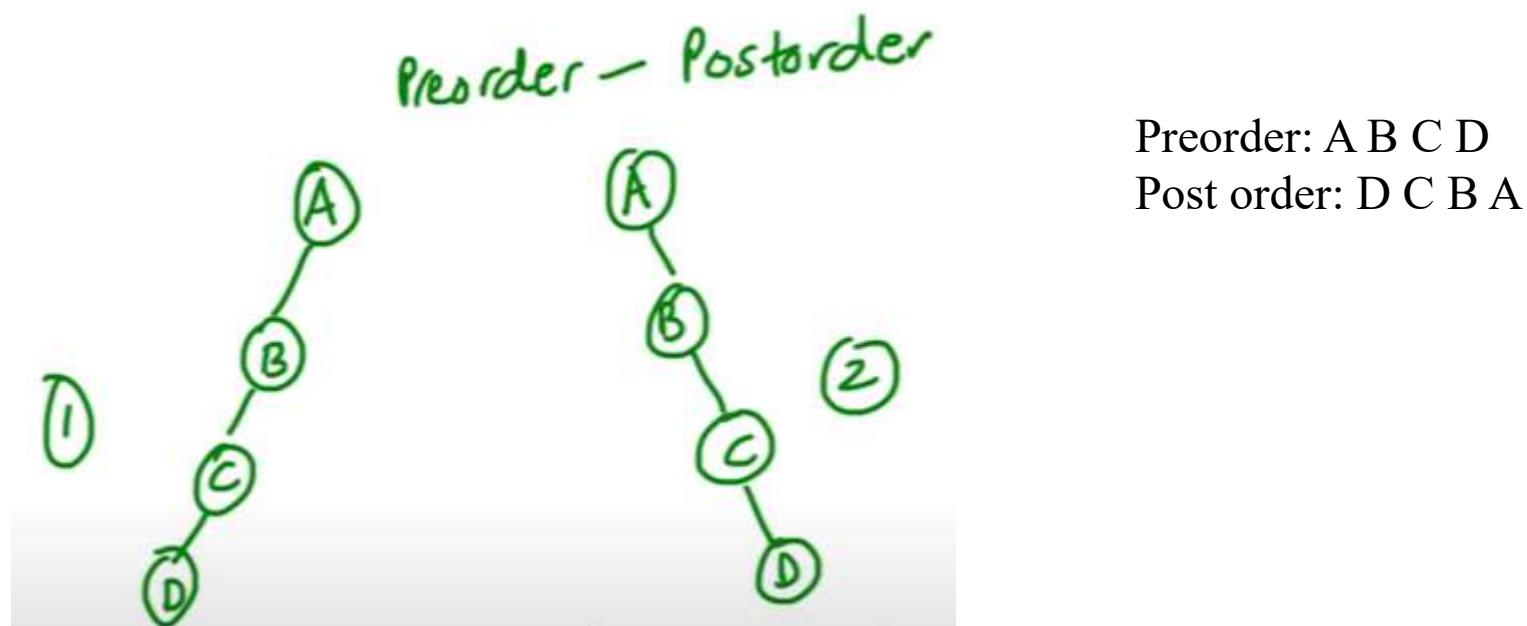
The same approach is used for the construction of **binary search tree**.

Complexity for the construction of BST =  $O(n \log n)$

Complexity for the construction of Binary Tree =  $O(n^2)$

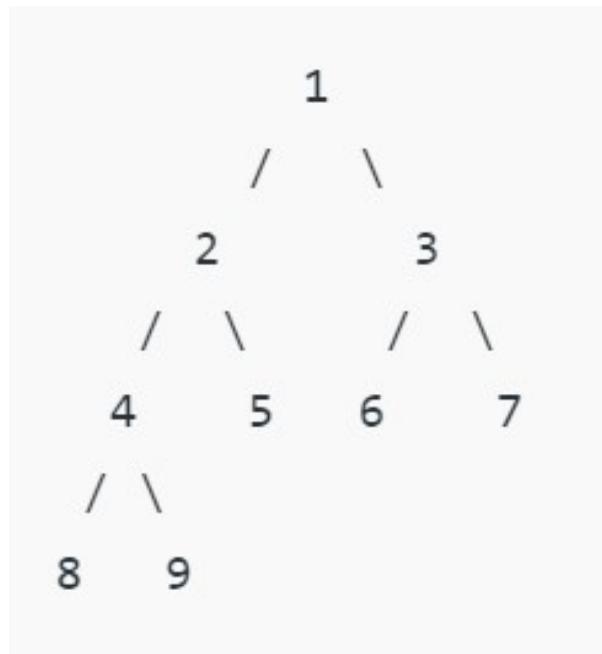
# Construct Binary Tree from Preorder and Postorder Traversal

- Cannot construct a **unique binary tree**
- Can construct a unique **full binary tree**



# Contd...

- Pre-order: 1, 2, 4, 8, 9, 5, 3, 6, 7 (Root->Left->Right)
- Post-order: 8, 9, 4, 5, 2, 6, 7, 3, 1 (Left->Right->Root)



# Number of trees with n keys

- $n = 3 \{1, 2, 3\}$

How many BST can you draw?

5 BST we can draw.

- $n= 4$

Number of BST ??

- For n

Generalized formula???

# Hashing Algorithms

Dr. Amit Praseed

# The Concept of Hashing

- Hashing is a technique which computes a special value (called “hash”) for different key values



- Consider the possibility that we have a hash function that converts a data value to the corresponding memory address (or array index) where it is stored
  - Allows us to do  $O(1)$  searching

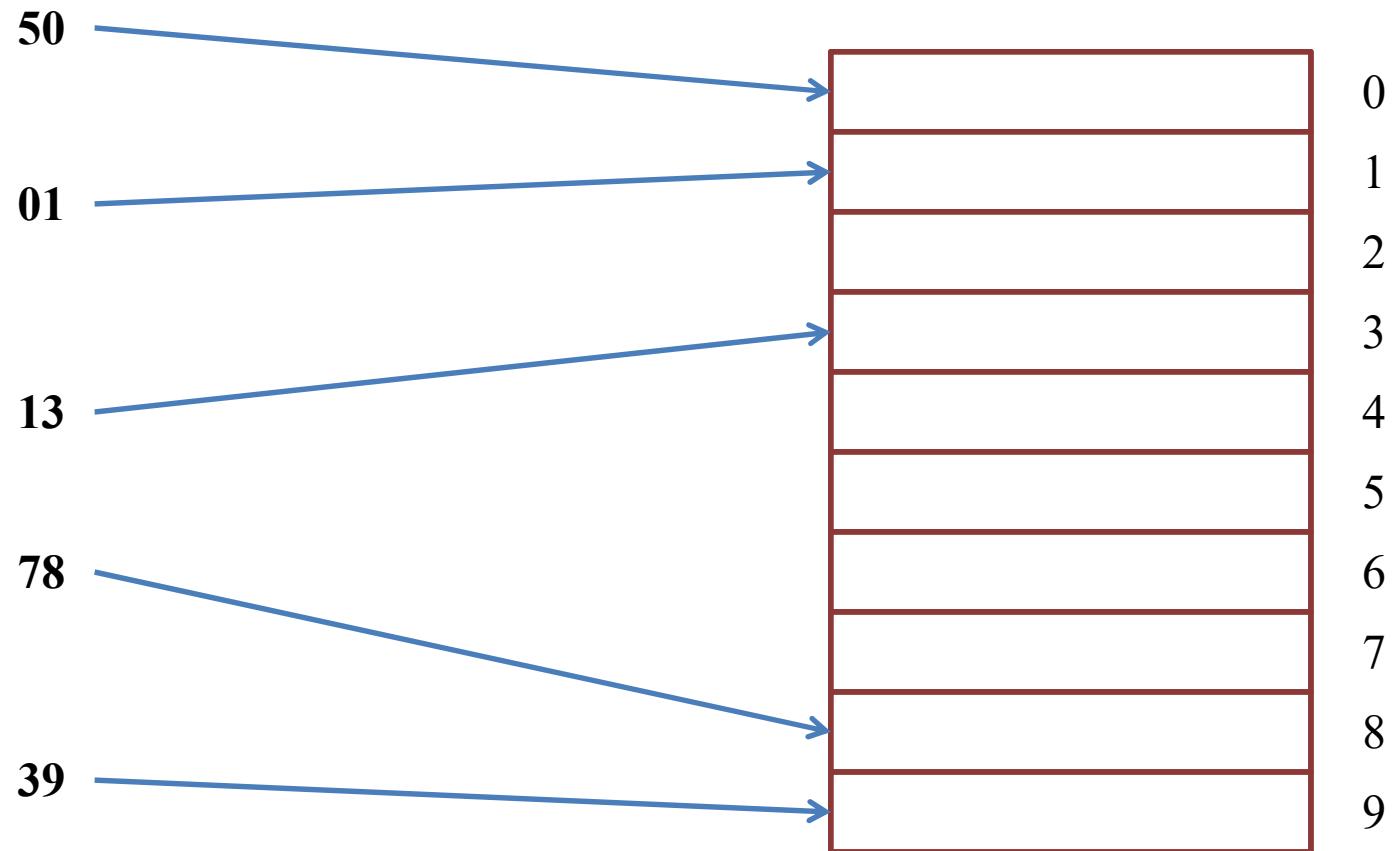
# A Simple Example of Hashing

- A very simple (non) example of hashing is as follows:
  - We want to store data values from 1 to 100
  - Reserve memory space for 100 data elements, possibly in an array  $arr[100]$
  - Store data element  $i$  in array index  $arr[i-1]$
  - This allows us to do  $O(1)$  searching
  - Leads to wastage of space, especially if not all data values in the range are present

# A Simple Example of Hashing

- Let us reduce the space requirement:
  - We want to store data values from 1 to 100
  - Reserve memory space for 10 data elements, possibly in an array  $arr[10]$
  - This data structure is called a hash table
  - Store data element  $i$  in array index  $i \% 10$
  - This is called a Modular Hashing Algorithm

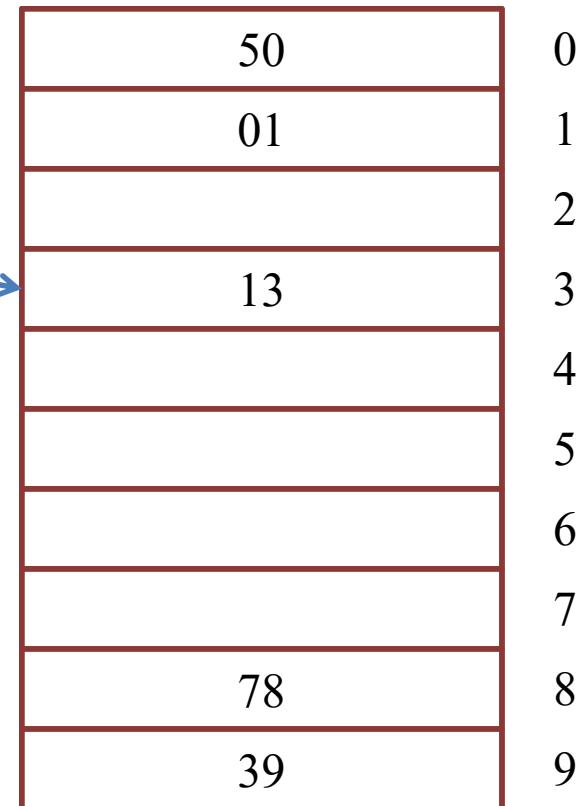
# A Modular Hashing Algorithm



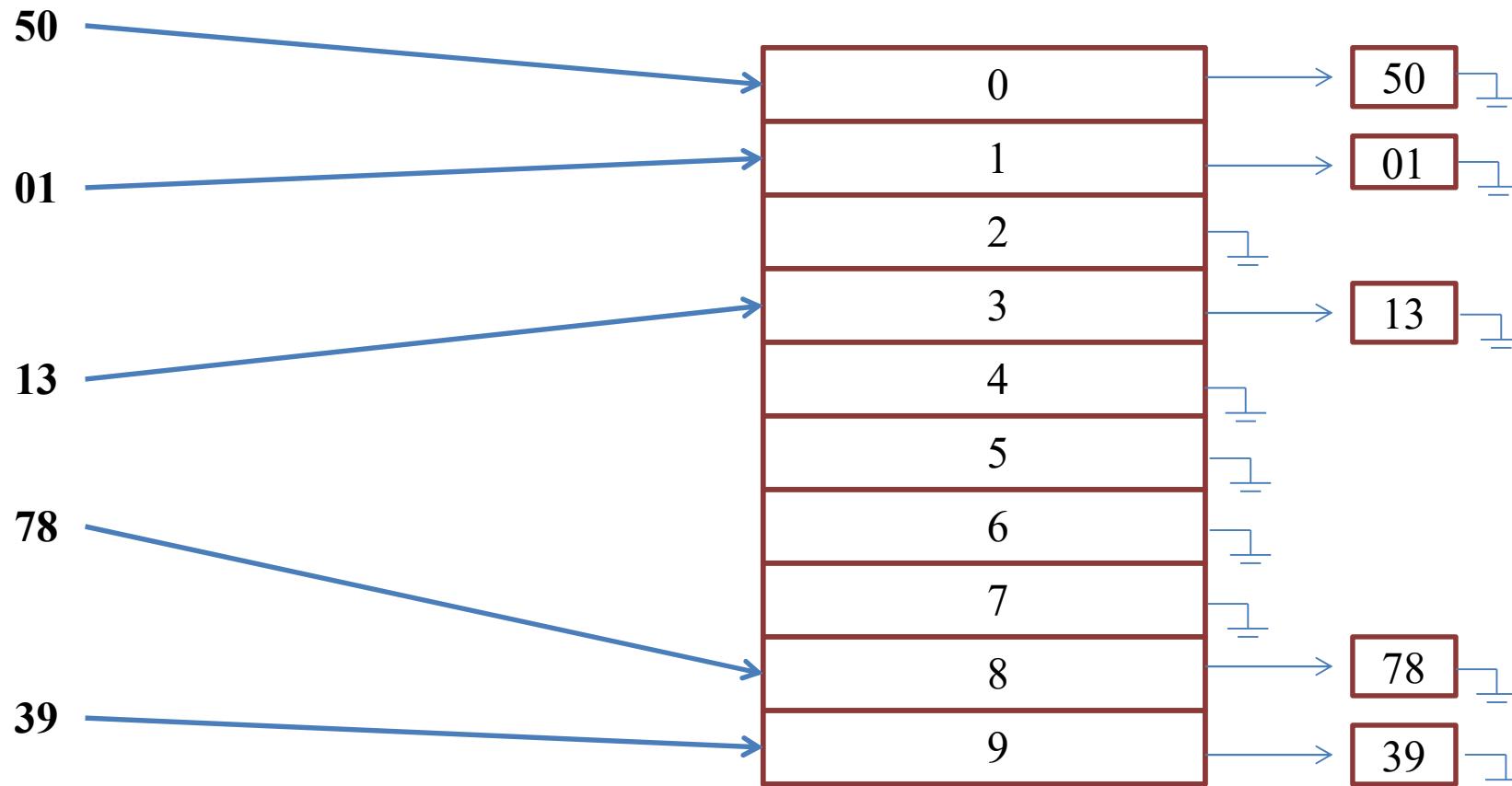
# A Modular Hashing Algorithm

33

- Since there are 100 elements to be mapped to only 10 memory locations, more than one data element will map to the same memory location
- This is called **Collision**
- Ideally, you should choose a hash function that does not have collisions, but it is difficult in practice
- Thus, we settle for hash functions which minimize collisions, and we use certain **collision resolution strategies** when they arise

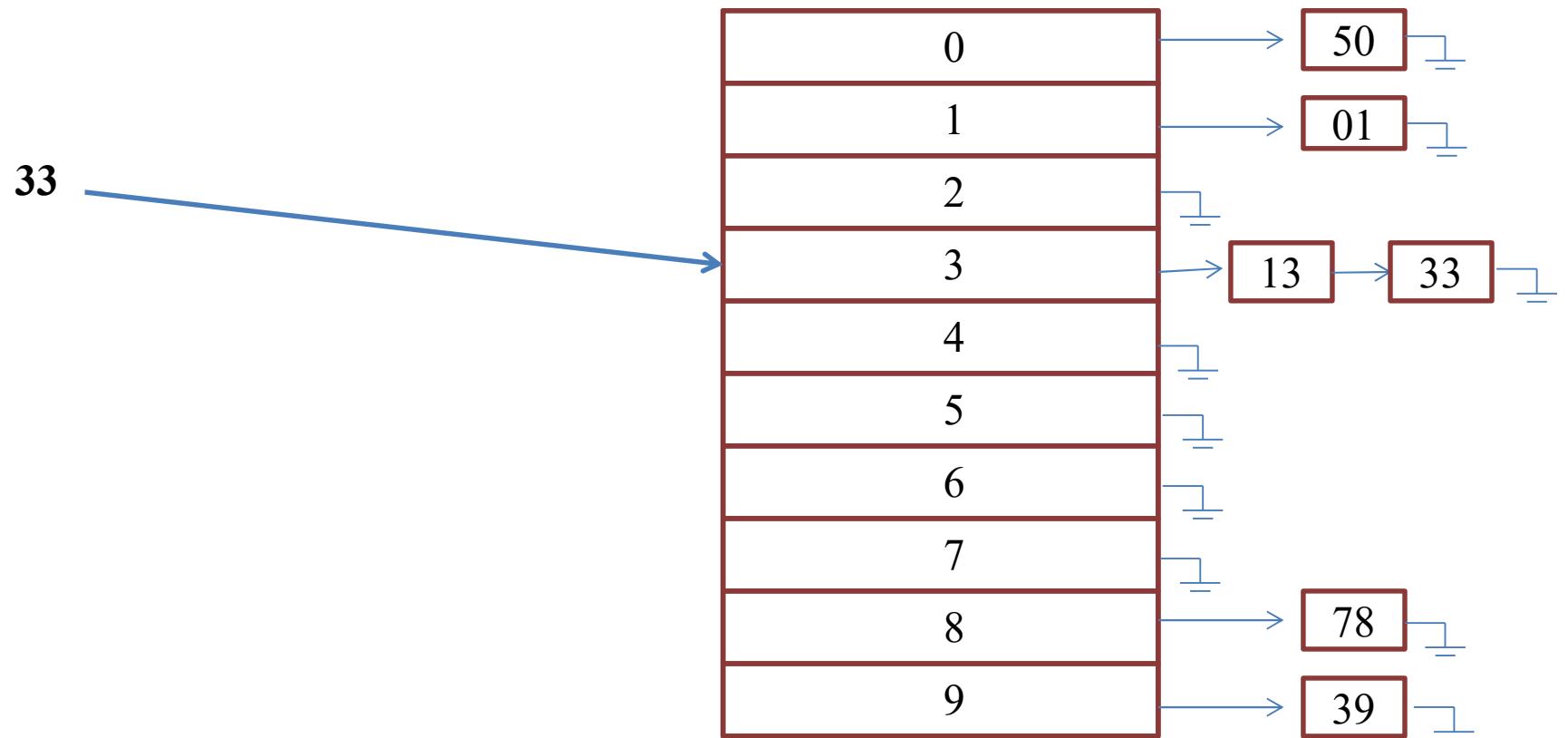


# Separate Chaining



Instead of mapping to a memory location, the hash function maps to the head pointer of a linked list. The linked list contains all the data items that have the same hash value

# Separate Chaining



**When there is a collision, the new data item is simply added to the linked list corresponding to the hash value**

# Open Addressing

- Open addressing is a collision resolution strategy that does not rely on linked lists
- It relies on the assumption that there will be empty space within the hash table, which has not been used by any data value
- Suppose a data value  $x$  hashes to a location  $H(x)$ , but suffers a collision
  - Successively try to put  $x$  in locations  $(H(x) + F(i))$  till an empty space is identified
  - The function  $F(i)$  decides on the open addressing strategy used

# Linear Probing

- This is a very simple open addressing strategy
  - $F(i) = i$
- In linear probing, if an element  $x$  is hashed to a particular location  $H(x)$ , but suffers a collision, successive locations,  $H(x)+1, H(x)+2\dots$ , are searched and the element  $x$  is stored in the first unused position

# Quadratic Probing

- In the quadratic probing strategy
  - $F(i) = i^2$
- In quadratic probing, if an element  $x$  is hashed to a particular location  $H(x)$ , but suffers a collision, successive locations,  $H(x)+1$ ,  $H(x)+4$ ,  $H(x)+9\dots$ , are searched and the element  $x$  is stored in the first unused position

# Rehashing

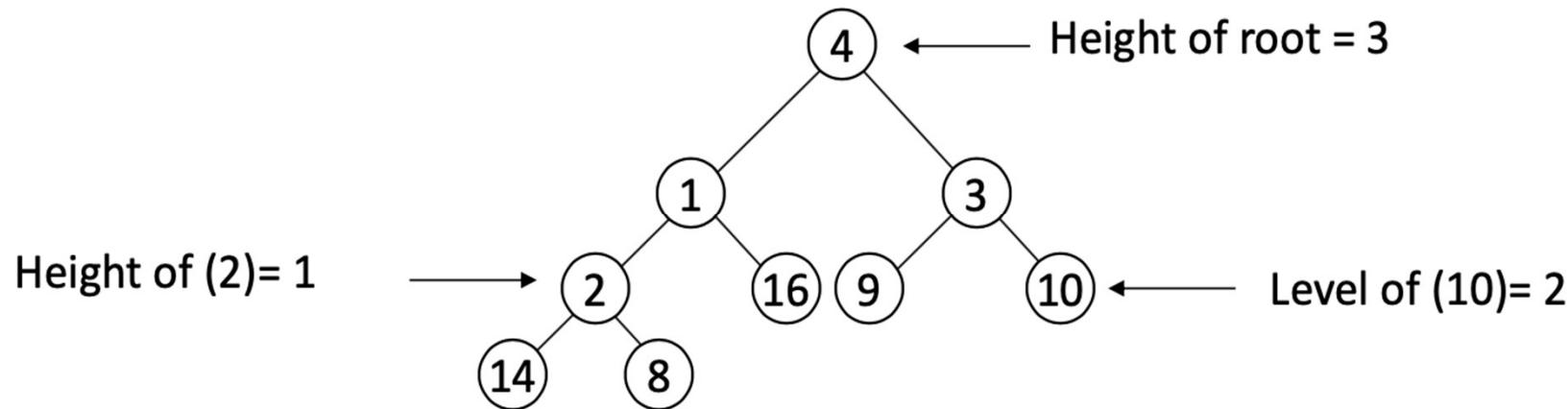
- As a hash table gets full, it becomes complex to insert further elements
- Rehashing suggests that we create a new hash table with double the size of the original, and then re-hash all the elements in the original table to the new table
- Complexity is  $O(n)$
- Needs to be used sparingly

# Heaps

Dr. Sreeja S R

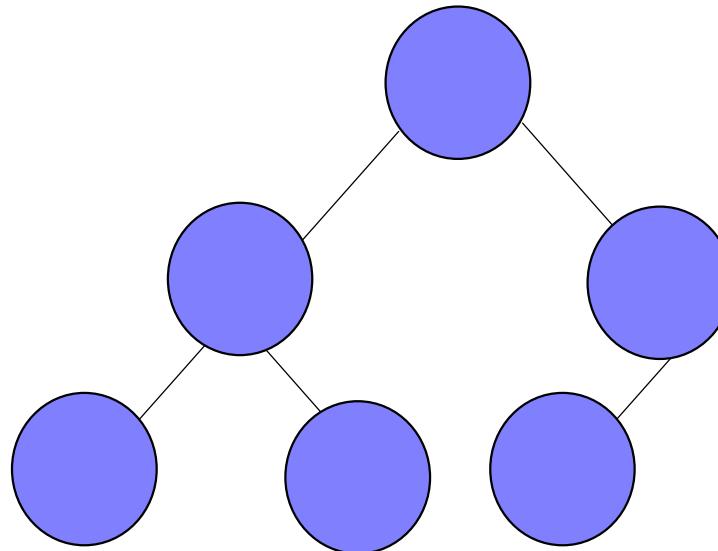
## Useful properties of binary tree

- There are **at most**  $2^l$  nodes at level (or depth)  $l$  of a binary tree
- A binary tree with height  $d$  has **at most**  $2^{d+1} - 1$  nodes
- A binary tree with  $n$  nodes has height **at least**  $\lfloor \log n \rfloor$



# Heaps

A heap is a certain kind of complete binary tree.



When a complete binary tree is built, its first node must be the root.

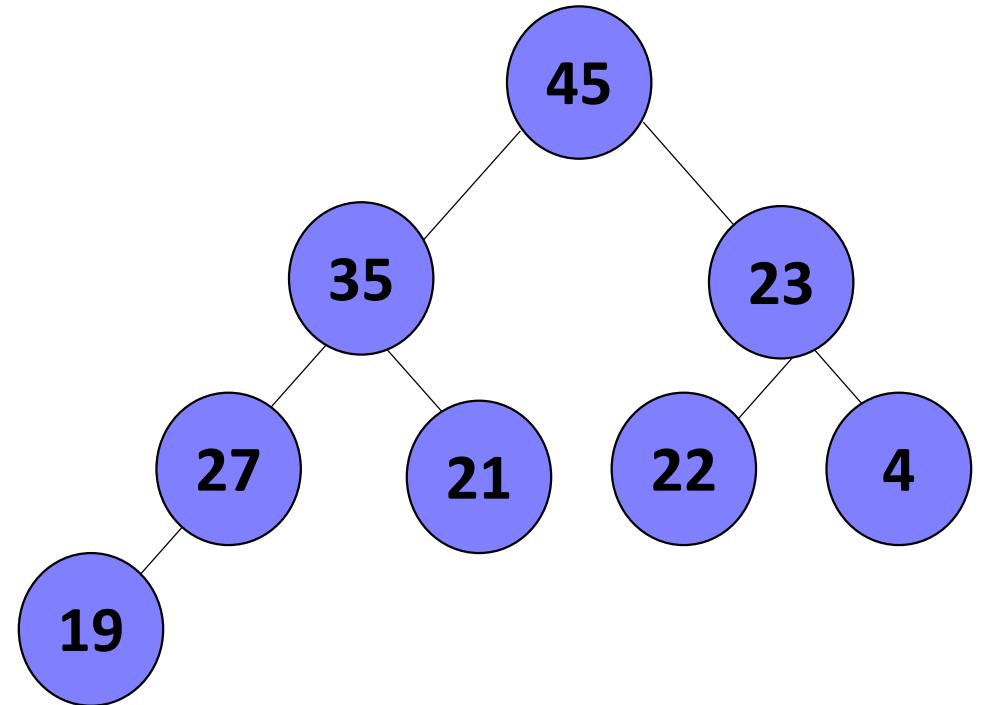
The second node is always the left child of the root.

The third node is always the right child of the root.

The next nodes always fill the next level from left-to-right.

# Heaps

- Each node in a heap contains a key that can be compared to other nodes' keys.
- Notice that this is not a binary search tree, but the keys do follow some resemblance of order.
- Can you see what rule is being enforced here?
- The *heap property* requires that each node's key is  $\geq$  the keys of its children.
- This is a handy property because the biggest node is always at the top. Because of this, a heap can easily implement a priority queue (where we need quick access to the highest priority item).



# Heap types

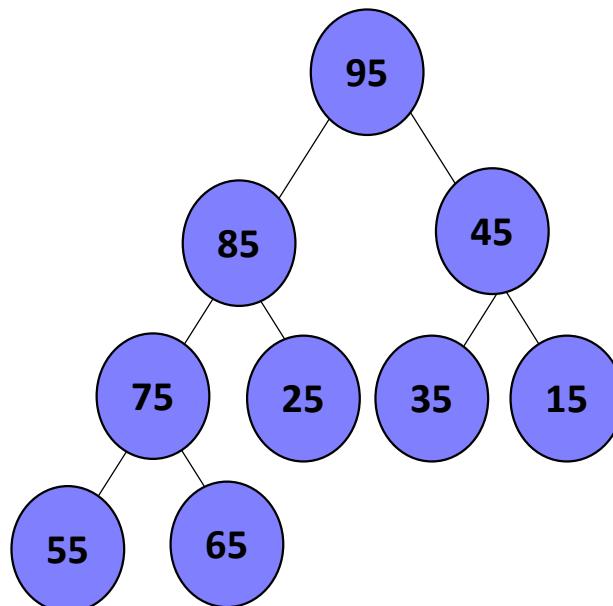
**Max-heaps** (largest element at root), have the *max-heap property*:

- for all nodes  $i$ , excluding the root:  $A[PARENT(i)] \geq A[i]$

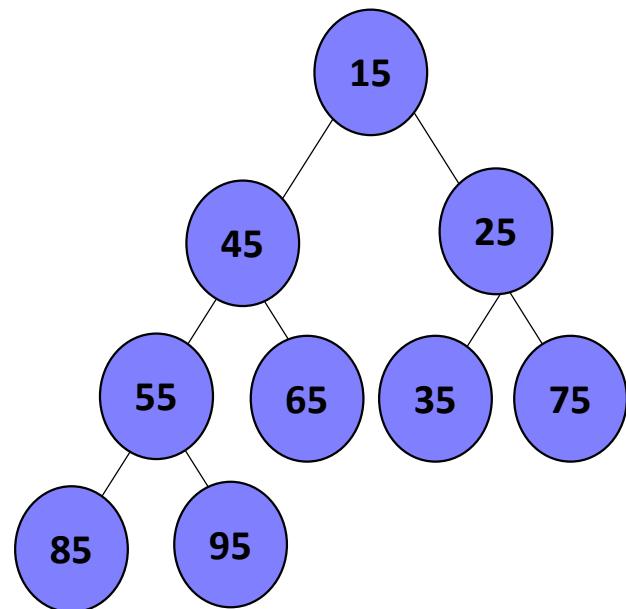
**Min-heaps** (smallest element at root), have the *min-heap property*:

- for all nodes  $i$ , excluding the root:  $A[PARENT(i)] \leq A[i]$

Max-heaps

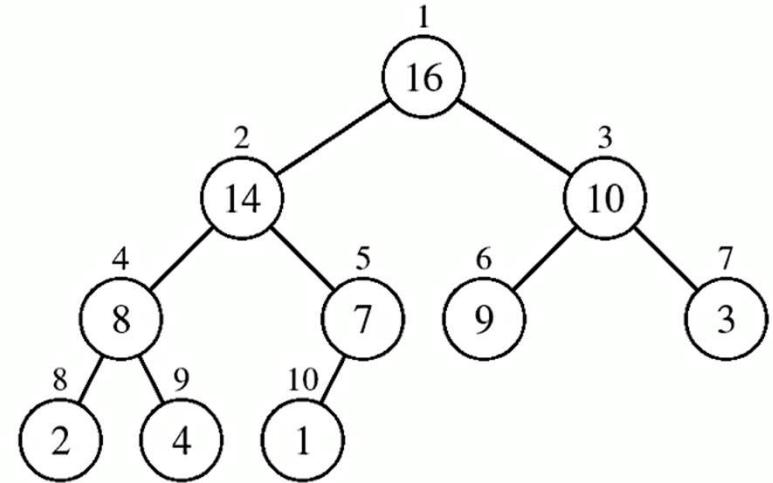
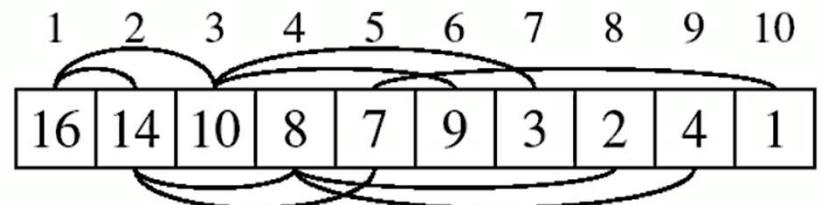


Min-heaps



# Array representation of Heaps

- A heap can be stored as an array  $A$ .
  - Root of tree is  $A[1]$
  - Left child of  $A[i] = A[2i]$
  - Right child of  $A[i] = A[2i + 1]$
  - Parent of  $A[i] = A[\lfloor i/2 \rfloor]$
- The elements in the subarray  $A[\lfloor n/2 \rfloor + 1 .. n]$  are leaves



No pointers required! Height of a binary heap is  $O(\log n)$

# Heap Operations

*MAX\_HEAPIFY* : Maintain/Restore the max-heap property

*INSERT* : insert a new element, and reorder the heap to maintain the heap property.

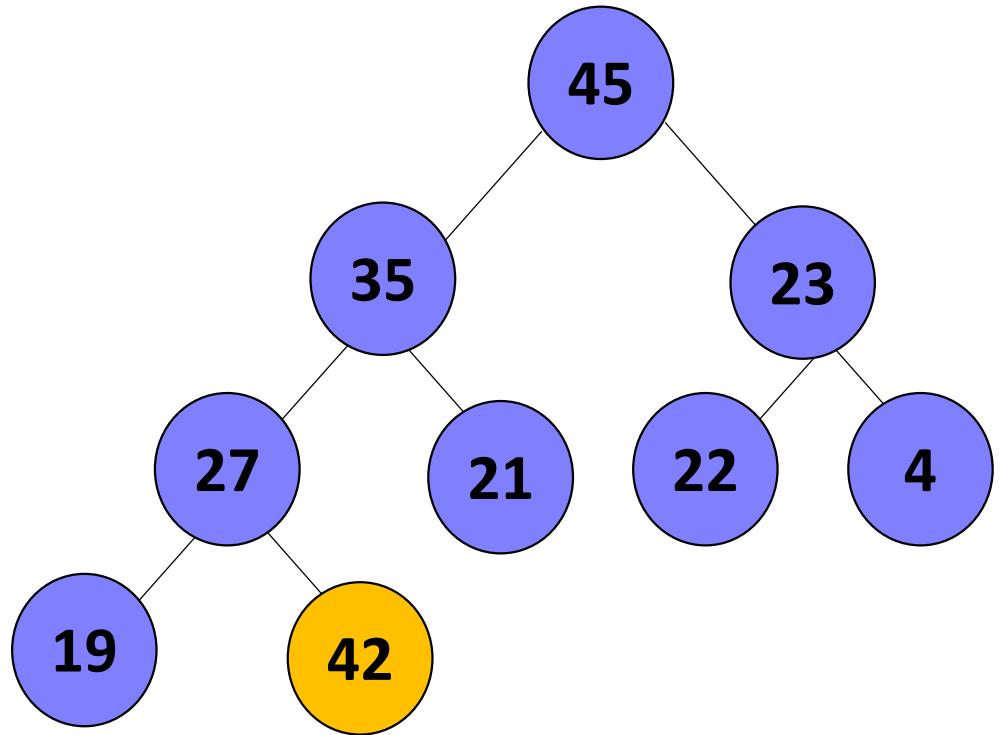
*BUILD\_HEAP* : produce a max-heap/min-heap from an unordered array

*EXTRACT\_MAX/EXTRACT\_MIN* : remove the largest (smallest) element from the heap and reorder the heap to maintain the heap property.

Find max, delete, update are other auxiliary operations in heap.

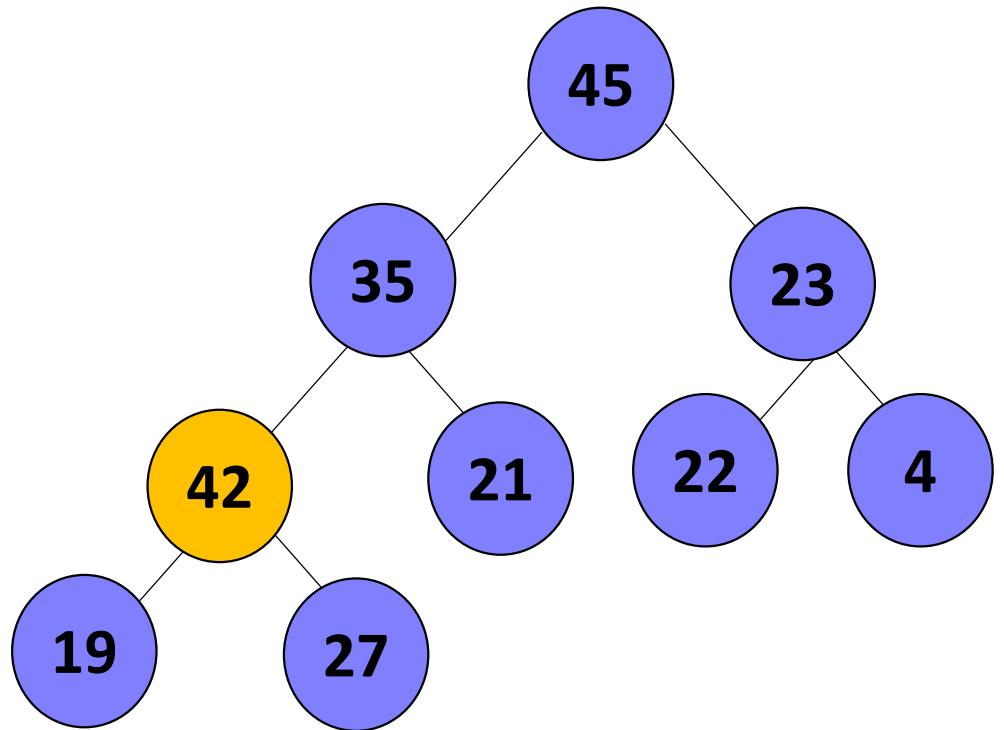
## Inserting an element to a heap

- Put the new node in the next available spot in the heap.
- The heap property is no longer valid. The new node 42 is bigger than its parent 27.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



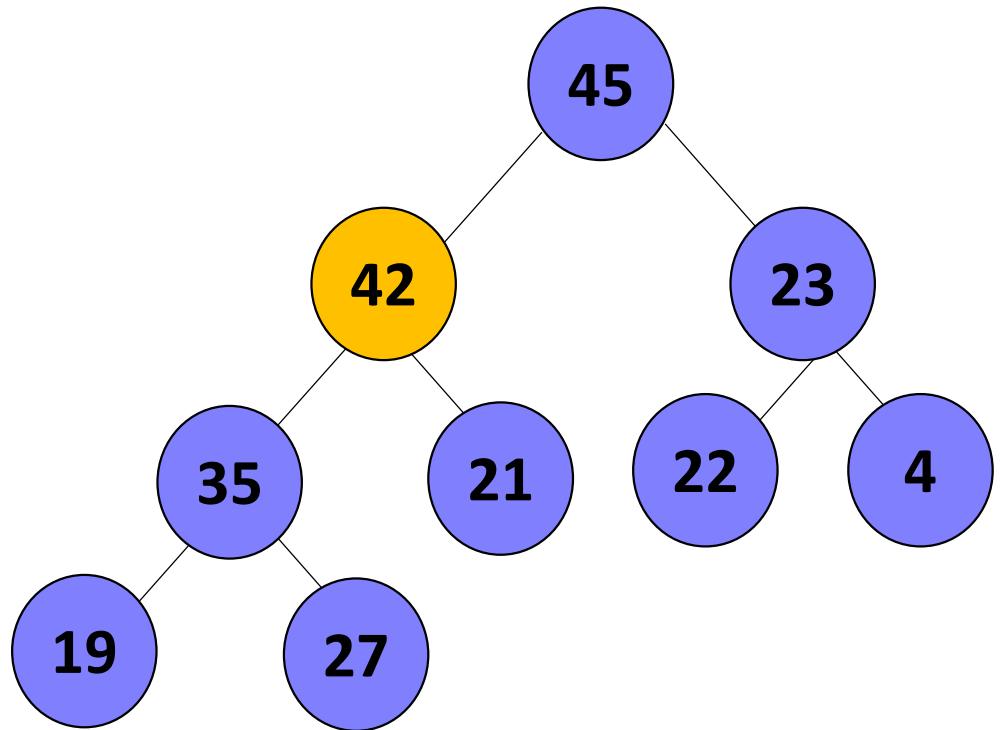
## Inserting an element to a heap

- The heap property is no longer valid. The new node 42 is bigger than its parent 35.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



## Inserting an element to a heap

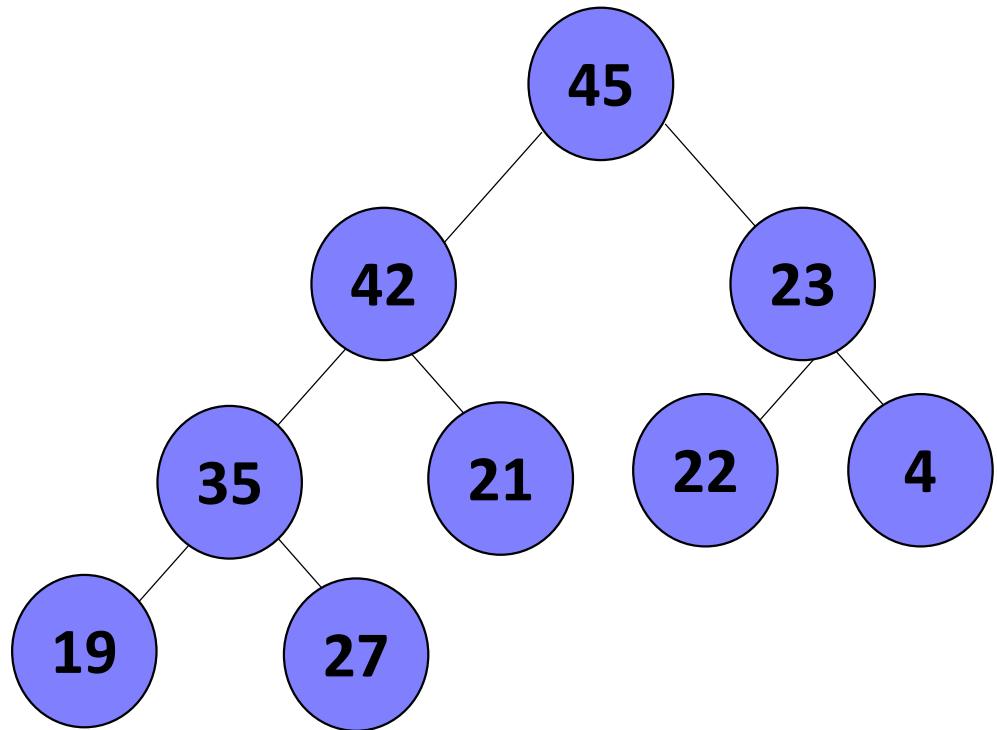
- Can we stop now? So, what are the conditions that can stop the pushing upward.



## Inserting an element to a heap

In general, there are two conditions that can stop the pushing upward:

- The parent has a key that is  $\geq$  new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



# Maintaining the Heap Property

**MAX-HEAPIFY (A, i, n)**

```
1. l ← LEFT(i)
2. r ← RIGHT(i)
3. if l ≤ n and A[l] > A[i]
4.     then largest ← l
5.     else largest ← i
6. if r ≤ n and A[r]>A[largest]
7.     then largest ← r
8. if largest ≠ i
9.     then exchange A[i] and A[largest]
10.    MAX-HEAPIFY(A, largest, n)
```

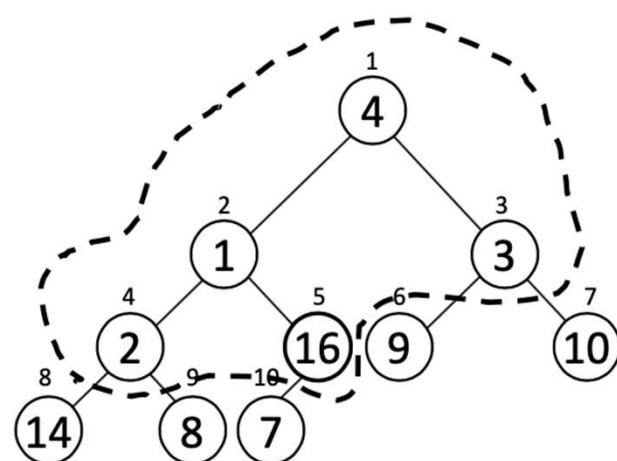
Running time of MAX-HEAPIFY is  $O(\log n)$

# Building a Heap

- Convert an array  $A[1 \dots n]$  into a max-heap ( $n = \text{length}[A]$ )
- The elements in the subarray  $A[(\lfloor n/2 \rfloor + 1) \dots n]$  are leaves
- Apply **MAX-HEAPIFY** on elements between 1 and  $\lfloor n/2 \rfloor$

*Alg:* BUILD-MAX-HEAP( $A$ )

1.  $n = \text{length}[A]$
2. **for**  $i \leftarrow \lfloor n/2 \rfloor$  **downto** 1
3.     **do** MAX-HEAPIFY( $A, i, n$ )



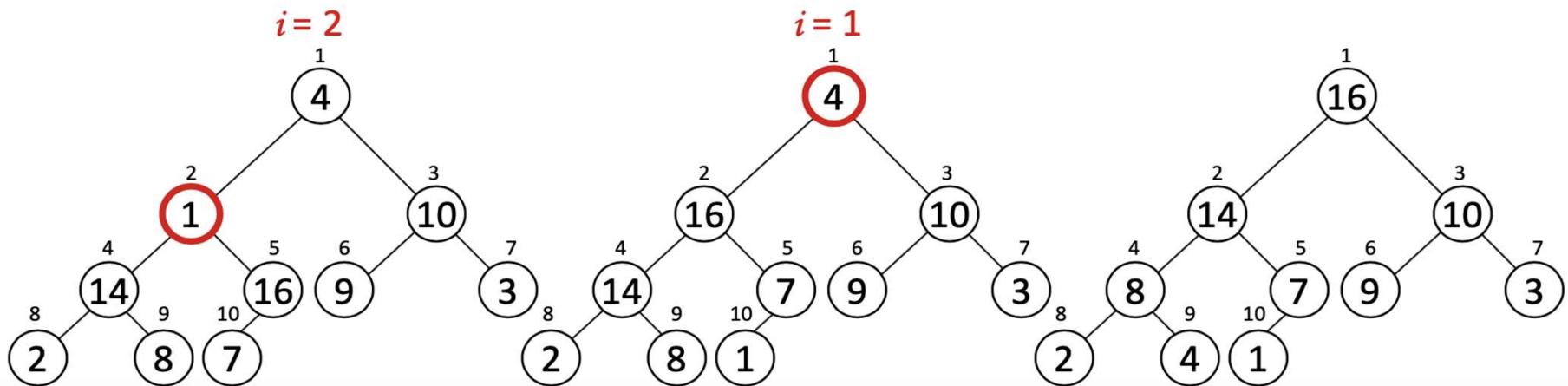
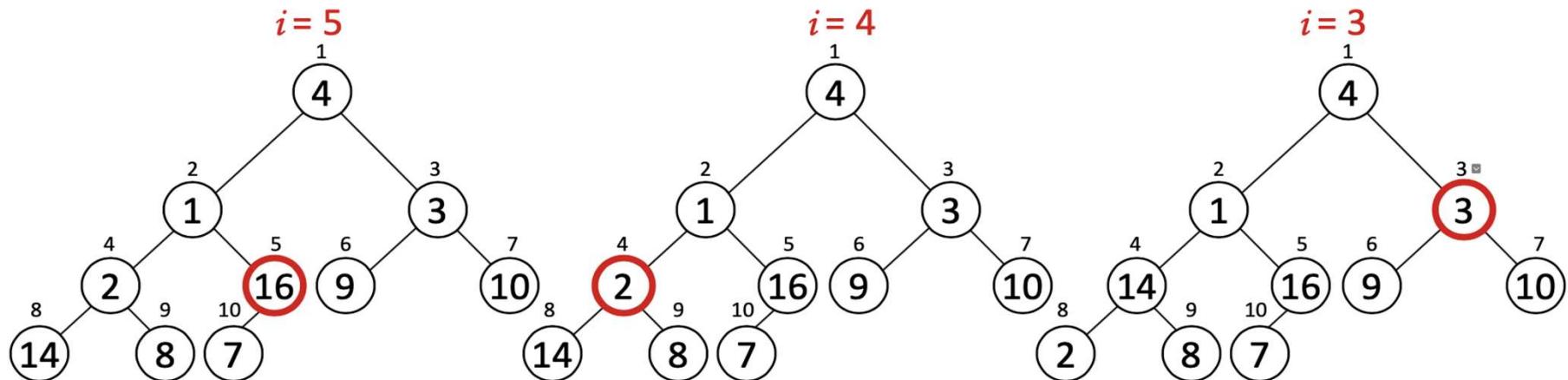
A: 

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

# Example:

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



## Running Time of BUILD-MAX-HEAP

*Alg:* BUILD-MAX-HEAP( $A$ )

1.  $n = \text{length}[A]$
  2. **for**  $i \leftarrow \lfloor n/2 \rfloor$  **downto** 1
  3.     **do** MAX-HEAPIFY( $A, i, n$ )
- $\longrightarrow O(\log n)$        $O(n)$

Running time of BUILD-MAX-HEAP:  $O(n \log n)$

## Running Time of BUILD-MAX-HEAP

- We have a loop of  $n/2$  times, and each time we call **MAX-HEAPIFY** which runs in  $(\log n)$ . This implies a bound of  $O(n \log n)$ . This is correct, but is a loose bound! We can do better.
- **Key observation:** Each time **MAX-HEAPIFY** is run within the loop, it is not run on the entire tree. We run it on subtrees, which have a lower height, so these subtrees do not take  $\log n$  time to run. Since the tree has more nodes on the leaf, most of the time the heaps are small compared to the size of  $n$ .

So, Running time of BUILD-MAX-HEAP:  $O(n)$

# HEAP\_EXTRACT\_MAX

## Goal:

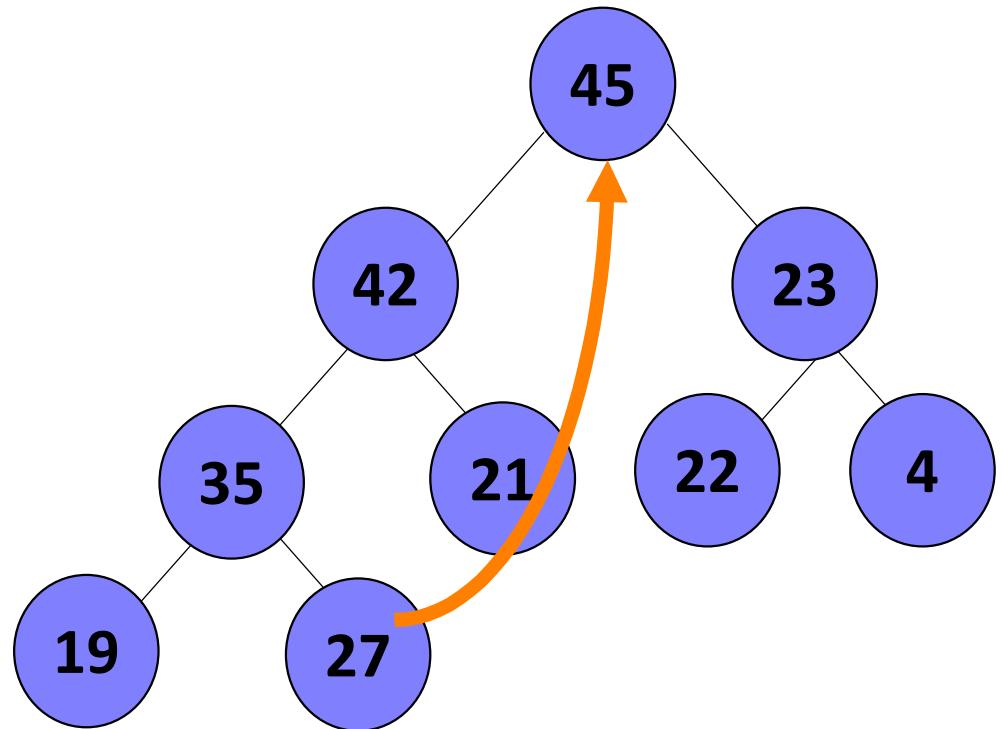
- Extract the largest element of the heap (i.e., return the max value) and also remove that element from the heap.

## Idea:

- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size  $n-1$

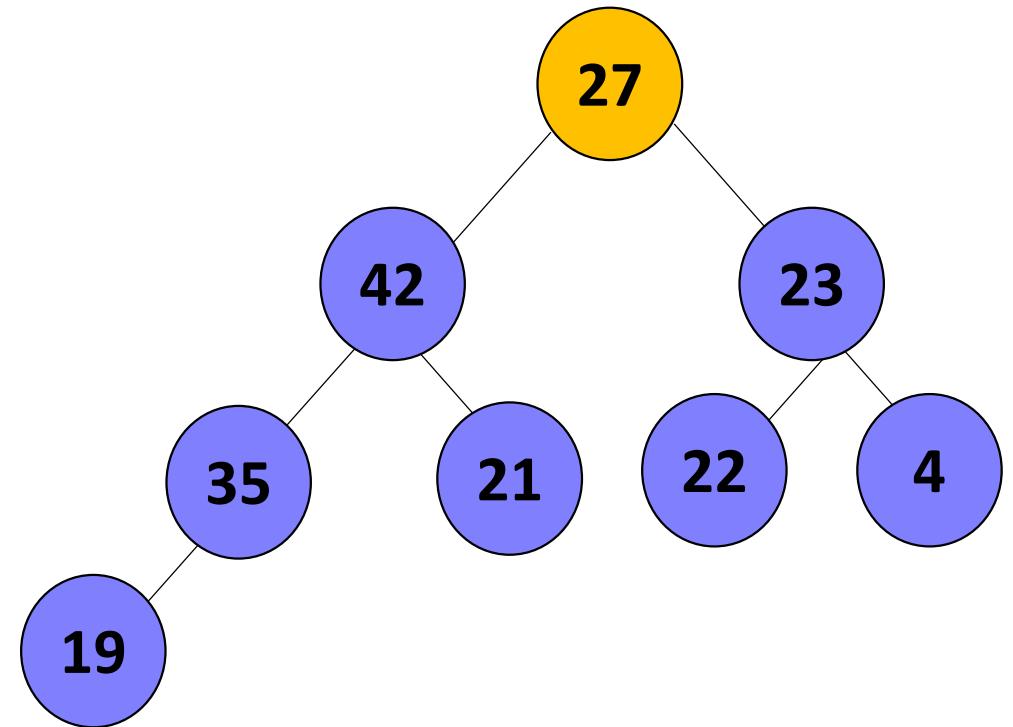
## HEAP\_EXTRACT\_MAX

- The first step of the removal is to move the last node of the tree onto the root.
- Decrease the size of the heap by 1 element
- In this example, 27 should be moved to root.



## HEAP\_EXTRACT\_MAX

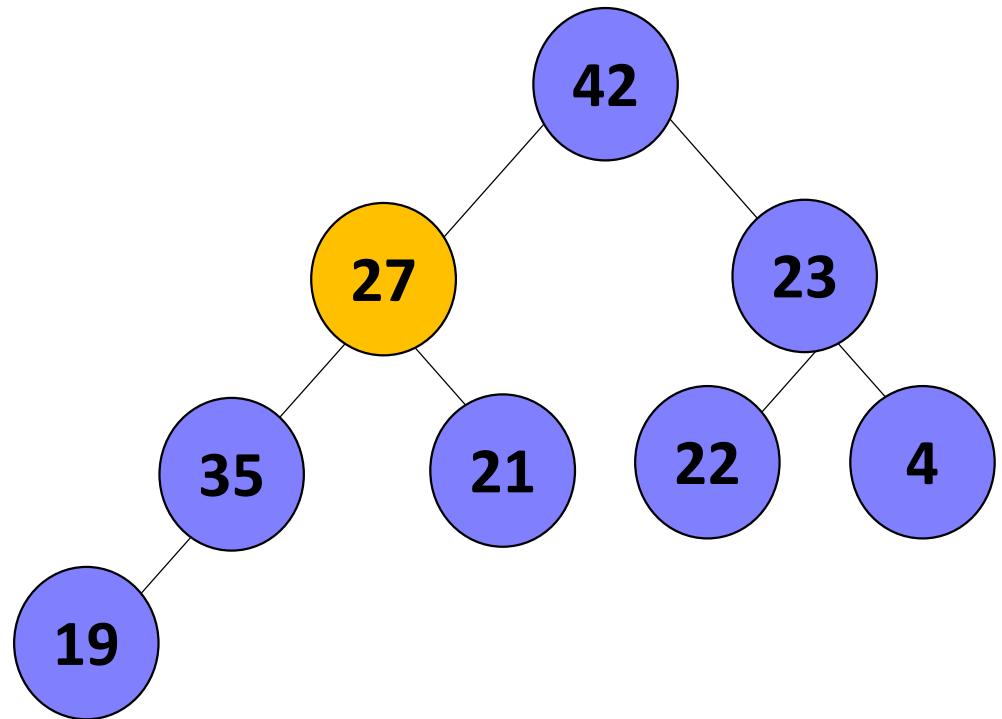
- Now the 27 is on top of the heap, and the original root (45) is no longer around. But the heap property is once again violated.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



Can you guess what the downward pushing is called? [reheapification downward](#).

## HEAP\_EXTRACT\_MAX

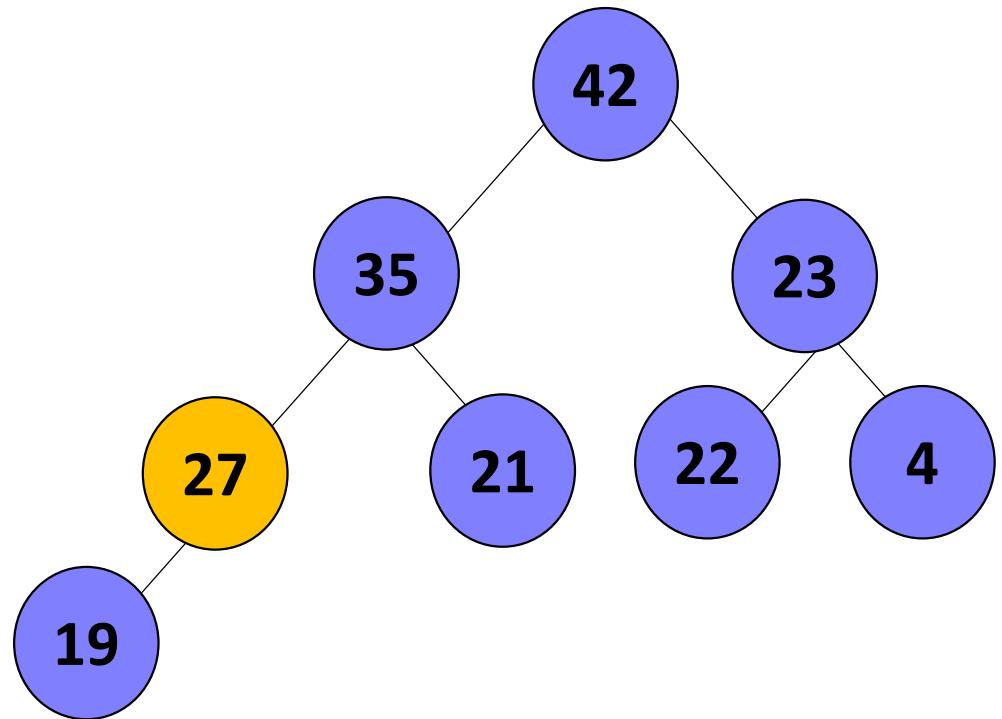
- When a node is pushed downward it is important to swap it with its largest child.
- Should we continue with the reheapification downward?



## HEAP\_EXTRACT\_MAX

Reheapification downward can stop under two circumstances:

1. The children all have keys that are  $\leq$  the out-of-place node.
2. The out-of-place node reaches a leaf.



## HEAP\_EXTRACT\_MAX

**Alg:** HEAP-EXTRACT-MAX(A, n)

1. **If**  $n < 1$   
**then error** "heap underflow"
2.  $\max \leftarrow A[1]$
3.  $A[1] \leftarrow A[n]$
4. MAX-HEAPIFY(A, 1, n-1)
5. **return** max

What is the Running time of HEAP\_EXTRACT\_MAX?

## Uses of Heaps

- It can be used to improve running times for several network optimization algorithms.
- It can be used to assist in dynamically-allocating memory partitions.
- A heapsort is considered to be one of the best sorting methods being in-place with no quadratic worst-case scenarios. (will see in next lecture)
- Finding the min, max, both the min and max, median, or even the k-th largest element can be done in linear time using heaps.

# Thank you!

# Heap sort and Priority Queue

Dr. Sreeja S R

# Heap-Sort

## Goal:

- Sort an array using heap representation

## Idea:

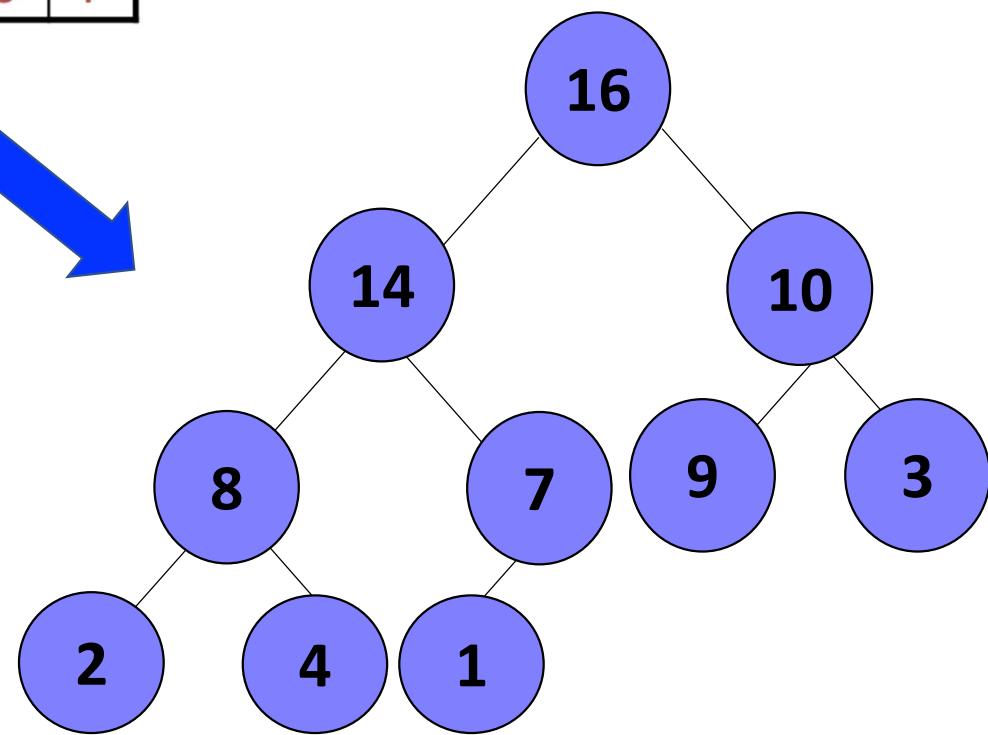
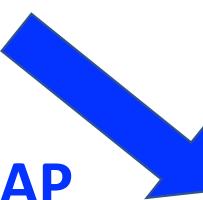
1. **BUILD-MAX-HEAP** from unordered array
2. Find maximum element  $A[1]$
3. Swap elements  $A[n]$  and  $A[1]$ :  
now max element is at the end of the array!
4. Discard node  $n$  from heap (by decrementing heap-size variable)
5. New root may violate max heap property, but its children are max heaps. Run **MAX-HEAPIFY** to fix this.
6. Go to Step 2 unless heap is empty.

## Heap-Sort - Example

A

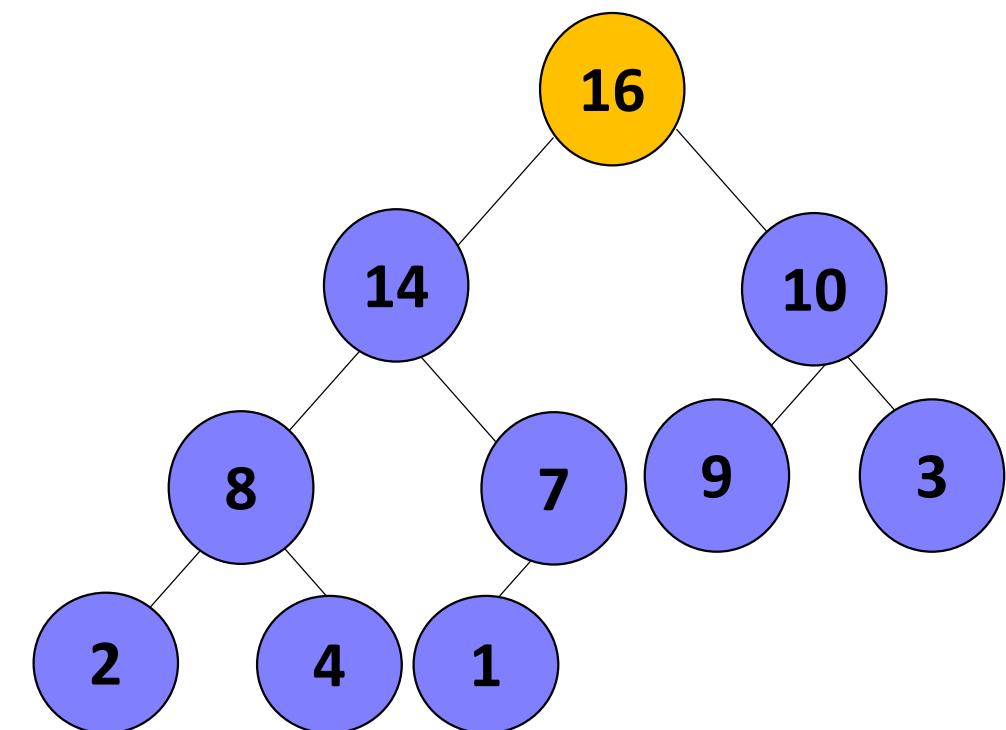
4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

1. BUILD-MAX-HEAP

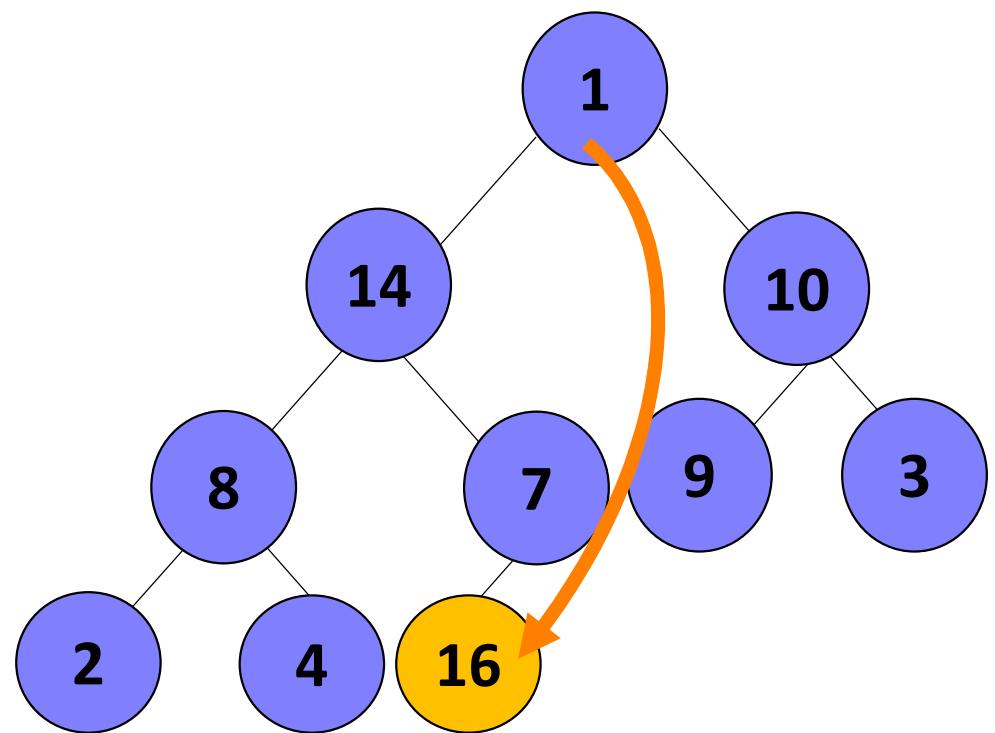


## Heap-Sort - Example

2. Find maximum element  $A[1]$

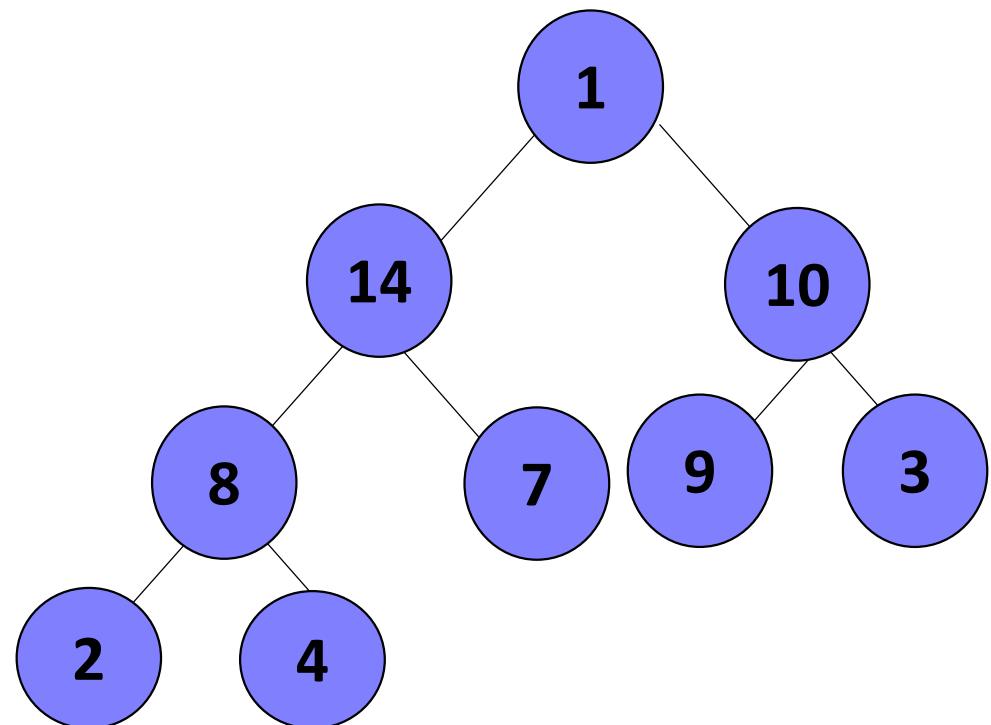


3. Swap elements  $A[n]$  and  $A[1]$



## Heap-Sort - Example

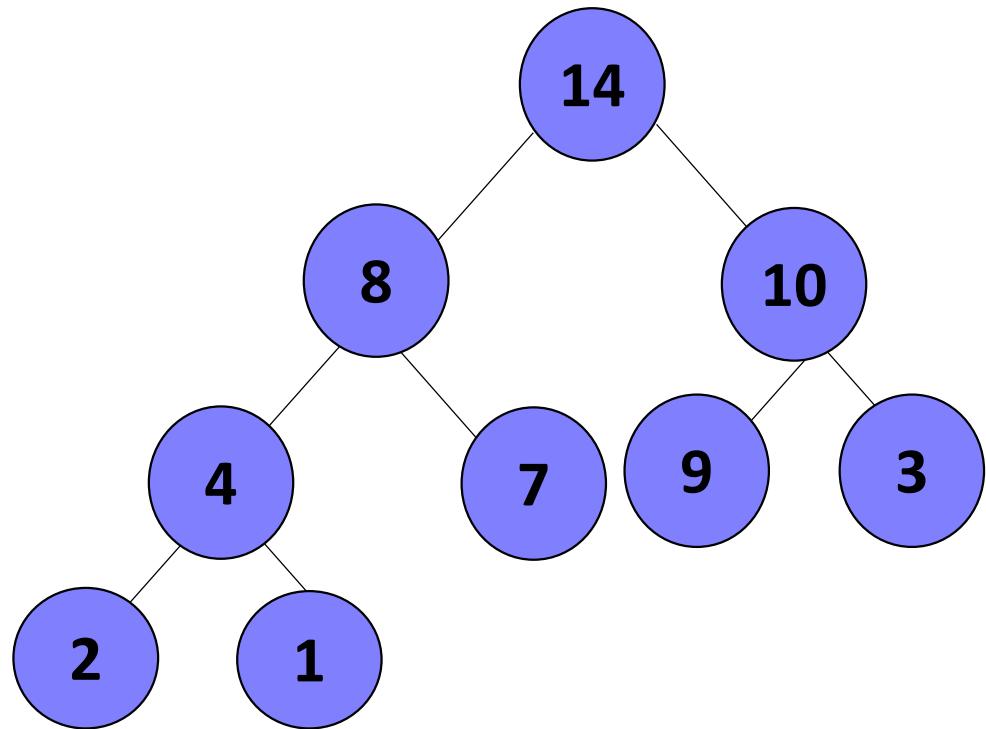
4. Discard node  $n$  from heap



16

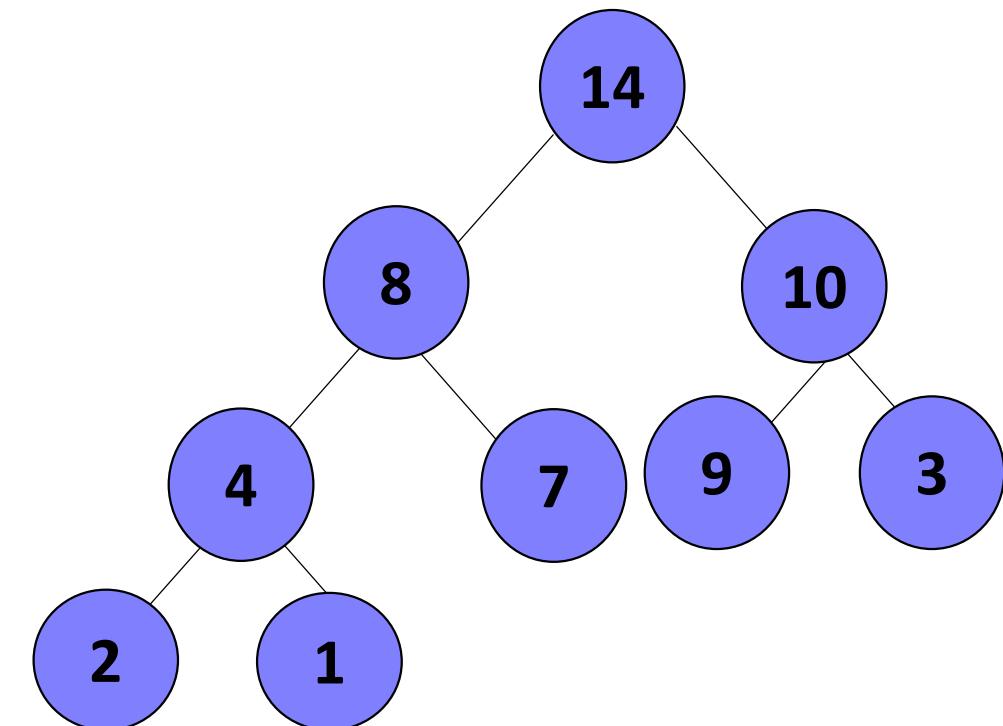
not part of heap

5. Run **MAX-HEAPIFY**



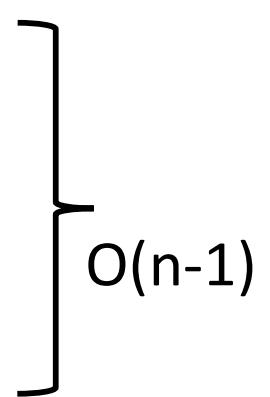
## Heap-Sort - Example

6. Repeat the process until heap is empty



The elements which is removed from  
the heap

## Running Time of HEAPSORT

1.  $\text{BUILD-MAX-HEAP}(A)$   $O(n)$
  2. **for**  $i \leftarrow \text{length}[A]$  **downto** 2
  3.     **do exchange**  $A[1] \leftrightarrow A[i]$
  4.      $\text{MAX-HEAPIFY}(A, 1, i - 1)$   $\longrightarrow O(\log n)$
- 

Running time of HEAPSORT:  $O(n \log n)$

# HEAPSORT

```
void heapify(int arr[], int n, int i)
{
    // Find largest among root, left child and right child
    int tmp;
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    // Swap and continue heapifying if root is not largest
    if (largest != i)
    {
        //swap(arr[i], arr[largest]);
        tmp=arr[i];
        arr[i]=arr[largest];
        arr[largest]=tmp;
        printf("\n\tMaxHeap: ");
        printArray(arr,n);
        heapify(arr, n, largest);
    }
}
```

# HEAPSORT

```
void heapSort(int arr[], int n)
{
    int i,tmp;
    // Build max heap
    for (i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Heap sort
    for (i=n-1; i>=0; i--)
    {
        //swap(arr[0], arr[i]);
        tmp=arr[0];
        arr[0]=arr[i];
        arr[i]=tmp;
        printf("\n\tHeap Sort::");
        printArray(arr,n);
        // Heapify root element to get highest element at root again
        heapify(arr, i, 0);
    }
}
```

# HEAPSORT

```
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i<n; ++i)
        printf(" %d ", arr[i]);
        printf("\n");
}

int main()
{
    int arr[] = {15,19,10,7,17,16};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf( "UnSorted array is \n");
    printArray(arr, n);

    heapSort(arr, n);

    printf( "Sorted array is \n");
    printArray(arr, n);
    return 0;
}
```

# Priority Queues

## Properties:

- Each element is associated with a value (priority)
- The key with the highest (or lowest) priority is extracted first.

## Major operations:

- $\text{insert}(S, x)$  : insert element  $x$  into set  $S$
- $\text{max}(S)$  : return element of  $S$  with largest key
- $\text{extract\_max}(S)$  : return element of  $S$  with largest key and remove it from  $S$
- $\text{increase\_key}(S, x, k)$  : increase the value of element  $x$ 's key to new value  $k$

# Priority Queue problem

- Where might we want to use heaps? Consider the Priority Queue problem
  - Given a sequence of objects with varying degrees of priority, and we want to deal with the highest-priority item first.
- Managing air traffic control
  - Want to do most important tasks first.
  - Jobs placed in queue with priority, controllers take off queue from top
- Scheduling jobs on a processor
  - Critical applications need high priority
- Event-driven simulator with time of occurrence as key.
  - Use min-heap, which keeps smallest element on top, get next occurring event.

# Running time of priority queue with different representations

## □ Priority queue using **linked list**

- Remove a key -  $O(1)$
- Insert a key -  $O(n)$
- Increase key -  $O(n)$
- Extract max key -  $O(1)$

## □ Priority queue using **heaps**

- Remove a key -  $O(\log n)$
- Insert a key -  $O(\log n)$
- Increase key -  $O(\log n)$
- Extract max key -  $O(\log n)$

# Thank you!

# Sorting

Dr. Sreeja S R

# Today's Discussion...

- Introduction
- Different sorting algorithms
- Sorting by Comparison
  - Insertion Sort
  - Selection Sort

# Introduction

# Sorting – The Task

- Given an array

$x[0], x[1], \dots, x[\text{size}-1]$

reorder entries so that

$x[0] \leq x[1] \leq \dots \leq x[\text{size}-1]$

Here, List is in non-decreasing order.

- We can also sort a list of elements in non-increasing order.

## Sorting – Example

- Original list:
  - **10, 30, 20, 80, 70, 10, 60, 40, 70**
- Sorted in non-decreasing order:
  - **10, 10, 20, 30, 40, 60, 70, 70, 80**
- Sorted in non-increasing order:
  - **80, 70, 70, 60, 40, 30, 20, 10, 10**

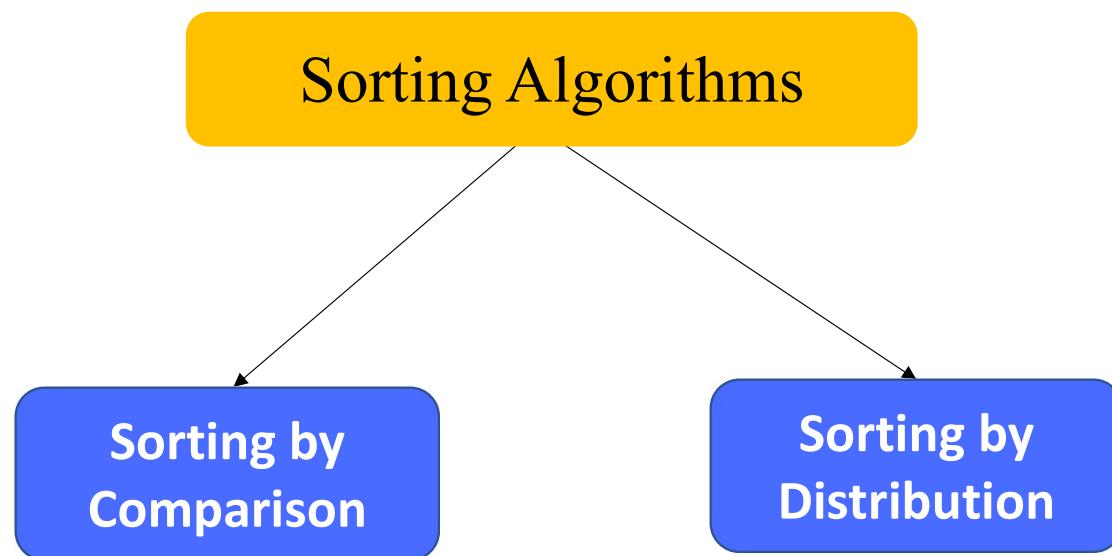
# Issues in Sorting

Many issues are there in sorting techniques

- How to rearrange a given set of data?
- Which data structures are more suitable to store data prior to their sorting?
- How fast the sorting can be achieved?
- How sorting can be done in a memory constraint situation?
- How to sort various types of data?

# Sorting Algorithms

# Sorting Algorithm types



# Sorting by Comparison

- Basic operation involved in this type of sorting technique is comparison. A data item is **compared** with other items in the list of items in order to find its place in the sorted list.
  - Insertion
  - Selection
  - Exchange
  - Enumeration

# Sorting by Comparison

## Sorting by comparison – Insertion:

- From a given list of items, one item is considered at a time. The item chosen is then inserted into an appropriate position relative to the previously sorted items. The item can be inserted into the same list or to a different list.

e.g.: [Insertion sort](#)

## Sorting by comparison – Selection:

- First the smallest (or largest) item is located and it is separated from the rest; then the next smallest (or next largest) is selected and so on until all item are separated.

e.g.: [Selection sort](#), [Heap sort](#)

# Sorting by Comparison

## Sorting by comparison – Exchange:

- If two items are found to be out of order, they are interchanged. The process is repeated until no more exchange is required.

e.g.: Bubble sort, Shell Sort, Quick Sort

## Sorting by comparison – Enumeration:

- Two or more input lists are **merged** into an output list and while merging the items, an input list is chosen following the required sorting order.

e.g.: Merge sort

# Sorting by Distribution

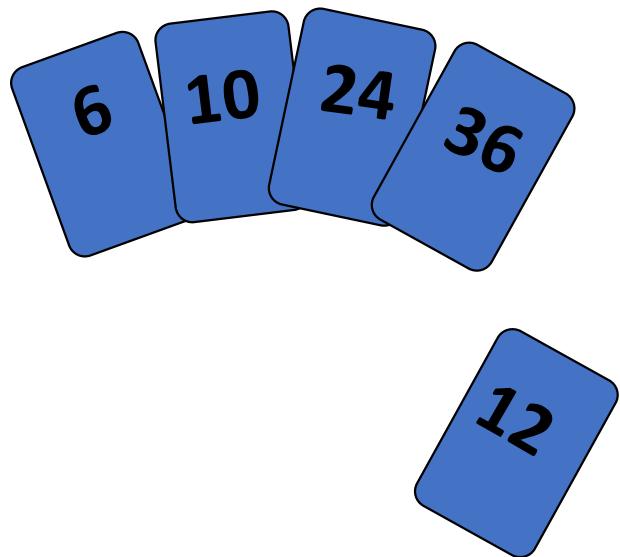
- No key comparison takes place
- All items under sorting are distributed over an auxiliary storage space based on the constituent element in each and then grouped them together to get the sorted list.
- Distributions of items based on the following choices
  - ✓ **Radix** - An item is placed in a space decided by the bases (or radix) of its components with which it is composed of.
  - ✓ **Counting** - Items are sorted based on their relative counts.
  - ✓ **Hashing** - Items are hashed, that is, dispersed into a list based on a hash function.

# Insertion Sort

# Insertion Sort

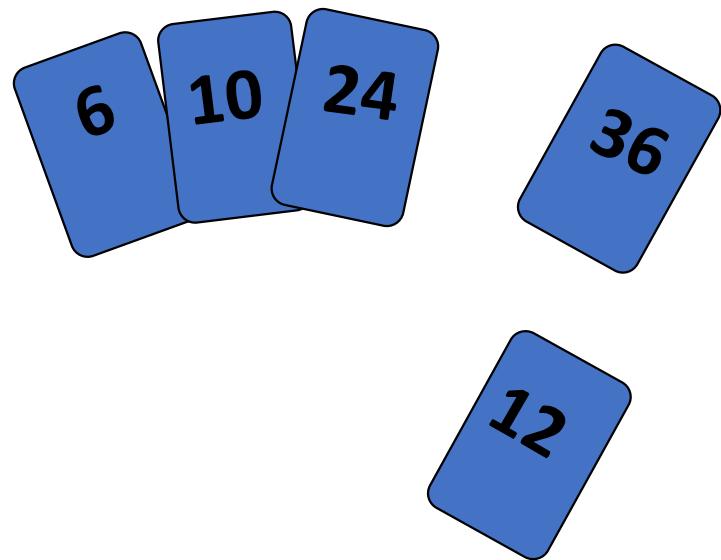
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

## Insertion Sort

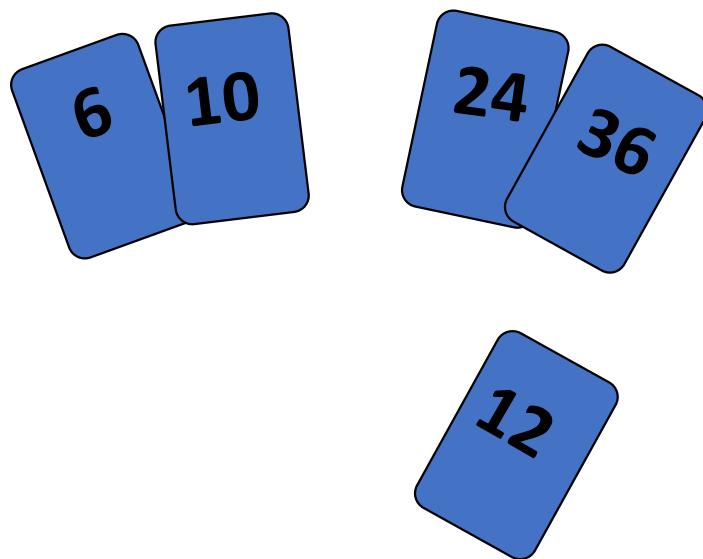


To insert 12, we need to make room for it by moving first 36 and then 24.

# Insertion Sort



# Insertion Sort

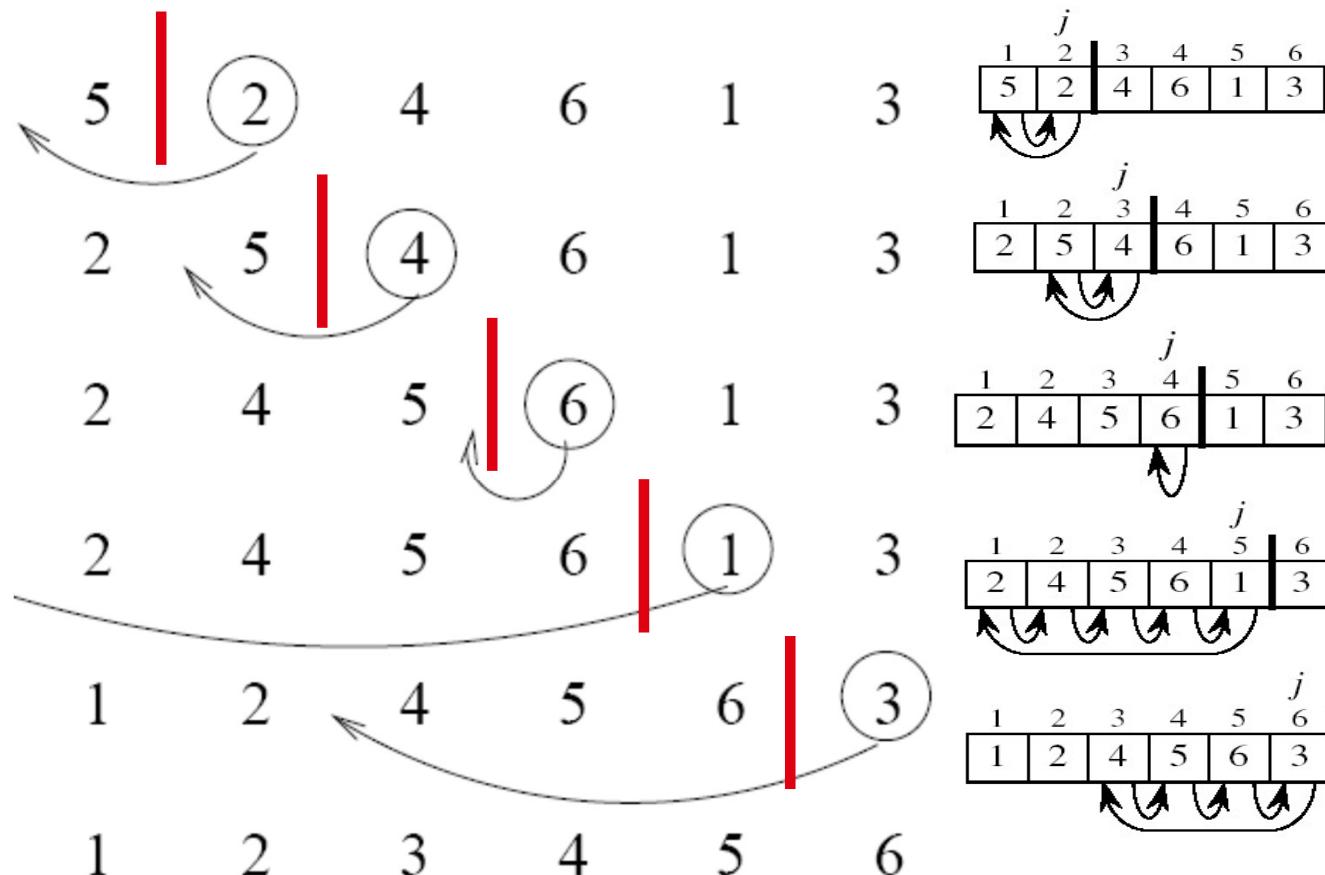


## Insertion Sort

For the given array A of size n, do the sorting as follows:

- Consider A[0] is sorted.
- Insert A[1] in the sorted array A[0]. So now A[0],A[1] are sorted
- Insert A[2] in the sorted array A[0],A[1]. So now A[0],A[1],A[2] are sorted
- Insert A[3] in the sorted array A[0],A[1],A[2]. So now A[0],A[1],A[2],A[3] are sorted
- .....
- Insert A[i] in the sorted array A[0],A[1],...,A[i-1]. So now A[0],A[1],...A[i] are sorted
- Continue until  $i = n-1$  (outer loop)

# Insertion Sort



# Insertion Sort

*Alg.:* INSERTION-SORT( $A$ )

**for**  $j \leftarrow 2$  **to**  $n$

**do**  $key \leftarrow A[j]$

        ▷ Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$

$i \leftarrow j - 1$

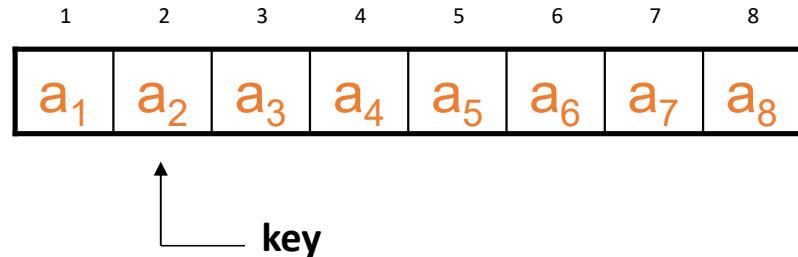
**while**  $i > 0$  and  $A[i] > key$

**do**  $A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow key$

- Insertion sort – sorts the elements in place



# Insertion Sort: Complexity Analysis

INSERTION-SORT( $A$ )	cost	times
<b>for</b> $j \leftarrow 2$ <b>to</b> $n$	$c_1$	$n$
<b>do</b> $\text{key} \leftarrow A[j]$	$c_2$	$n-1$
▷ Insert $A[j]$ into the sorted sequence $A[1 \dots j-1]$	$0$	$n-1$
$i \leftarrow j - 1$	$c_4$	$n-1$
<b>while</b> $i > 0$ and $A[i] > \text{key}$	$c_5$	$\sum_{j=2}^n t_j$
<b>do</b> $A[i + 1] \leftarrow A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
$i \leftarrow i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
$A[i + 1] \leftarrow \text{key}$	$c_8$	$n-1$

$t_j$ : # of times the while statement is executed at iteration  $j$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

# Comparisons and Exchanges in Insertion Sort

INSERTION-SORT( $A$ )

for  $j \leftarrow 2$  to  $n$

cost    times

$c_1$      $n$

do key  $\leftarrow A[j]$

$c_2$      $n-1$

Insert  $A[j]$  into the sorted sequence  $A[1 \dots j-1]$

0     $n-1$

$i \leftarrow j - 1$

$\approx n^2/2$  comparisons

$c_4$      $n-1$

while  $i > 0$  and  $A[i] > \text{key}$

$c_5$      $\sum_{j=2}^n t_j$

do  $A[i + 1] \leftarrow A[i]$

$c_6$      $\sum_{j=2}^n (t_j - 1)$

$i \leftarrow i - 1$

$\approx n^2/2$  exchanges

$c_7$      $\sum_{j=2}^n (t_j - 1)$

$A[i + 1] \leftarrow \text{key}$

$c_8$      $n-1$

# Insertion Sort: Summary of Complexity Analysis

Case	Comparisons	Sorting order	Complexity	Remarks
Case 1	$C(n) = (n - 1)$	Input list is in sorted order	$T(n) = O(n)$	Best case
Case 2	$C(n) = \frac{n(n - 1)}{2}$	Input list is sorted in reverse order	$T(n) = O(n^2)$	Worst case
Case 3	$C(n) = \frac{(n - 1)(n + 4)}{4}$	Input list is in random order	$T(n) = O(n^2)$	Average case

# Insertion Sort - Summary

- Advantages
  - Good running time for “almost sorted” arrays  $O(n)$
- Disadvantages
  - $O(n^2)$  running time in **worst** and **average** case
  - $\approx n^2/2$  comparisons and exchanges

# Selection Sort

# Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

# Selection Sort

*Alg.:* SELECTION-SORT( $A$ )

```
n ← length[ $A$ ]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if  $A[i] < A[\text{smallest}]$ 
                then smallest ← i
exchange  $A[j]$   $\leftrightarrow A[\text{smallest}]$ 
```



# Selection Sort - Example

X:  3 12 -5 6 142 21 -17 45

X:  -17 12 -5 6 142 21 3 45

X:  -17 -5 12 6 142 21 3 45

X:  -17 -5 3 6 142 21 12 45

X:  -17 -5 3 6 142 21 12 45

X:  -17 -5 3 6 12 21 142 45

X:  -17 -5 3 6 12 21 142 45

X:  -17 -5 3 6 12 21 45 142

X:  -17 -5 3 6 12 21 45 142

# Selection Sort: Complexity Analysis

*Alg.:* SELECTION-SORT( $A$ )

	cost	times
$n \leftarrow \text{length}[A]$	$c_1$	1
<b>for</b> $j \leftarrow 1$ <b>to</b> $n - 1$	$c_2$	$n$
<b>do</b> $\text{smallest} \leftarrow j$	$c_3$	$n-1$
<b>for</b> $i \leftarrow j + 1$ <b>to</b> $n$	$c_4$	$\sum_{j=1}^{n-1} (n - j + 1)$
<b>do if</b> $A[i] < A[\text{smallest}]$	$c_5$	$\sum_{j=1}^{n-1} (n - j)$
<b>then</b> $\text{smallest} \leftarrow i$	$c_6$	$\sum_{j=1}^{n-1} (n - j)$
<b>exchange</b> $A[j] \leftrightarrow A[\text{smallest}]$	$c_7$	$n-1$

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4 \sum_{j=1}^{n-1} (n - j + 1) + c_5 \sum_{j=1}^{n-1} (n - j) + c_6 \sum_{j=2}^{n-1} (n - j) + c_7(n-1) = \Theta(n^2)$$

# Selection Sort: Summary of Complexity Analysis

Case	Comparisons	Sorting order	Complexity	Remarks
Case 1	$c(n) = \frac{n(n - 1)}{2}$	Input list is in sorted order	$T(n) = O(n^2)$	Best case
Case 2	$c(n) = \frac{n(n - 1)}{2}$	Input list is sorted in reverse order	$T(n) = O(n^2)$	Worst case
Case 3	$c(n) = \frac{n(n - 1)}{2}$	Input list is in random order	$T(n) = O(n^2)$	Average case

# Any question?

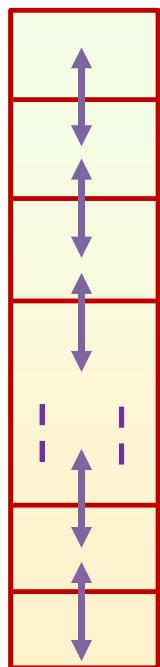


# Sorting

Dr. Sreeja S R

# Bubble Sort

# Bubble Sort



In every iteration  
heaviest element drops  
at the bottom.

The bottom  
moves upward.

The sorting process proceeds in several passes.

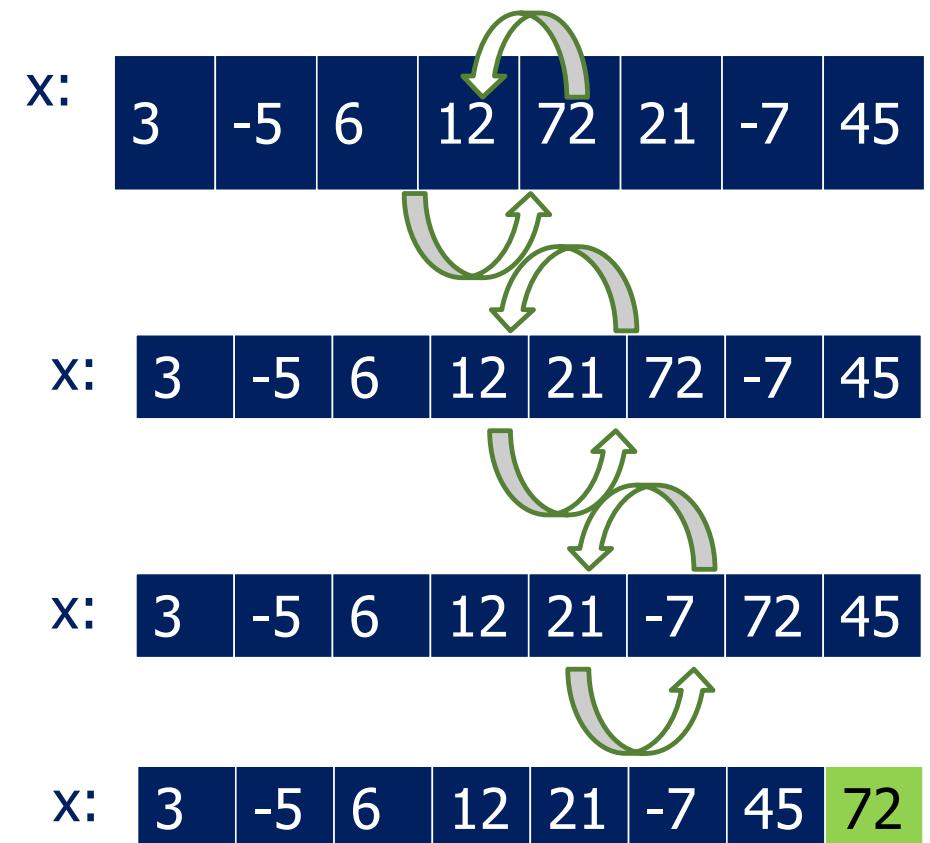
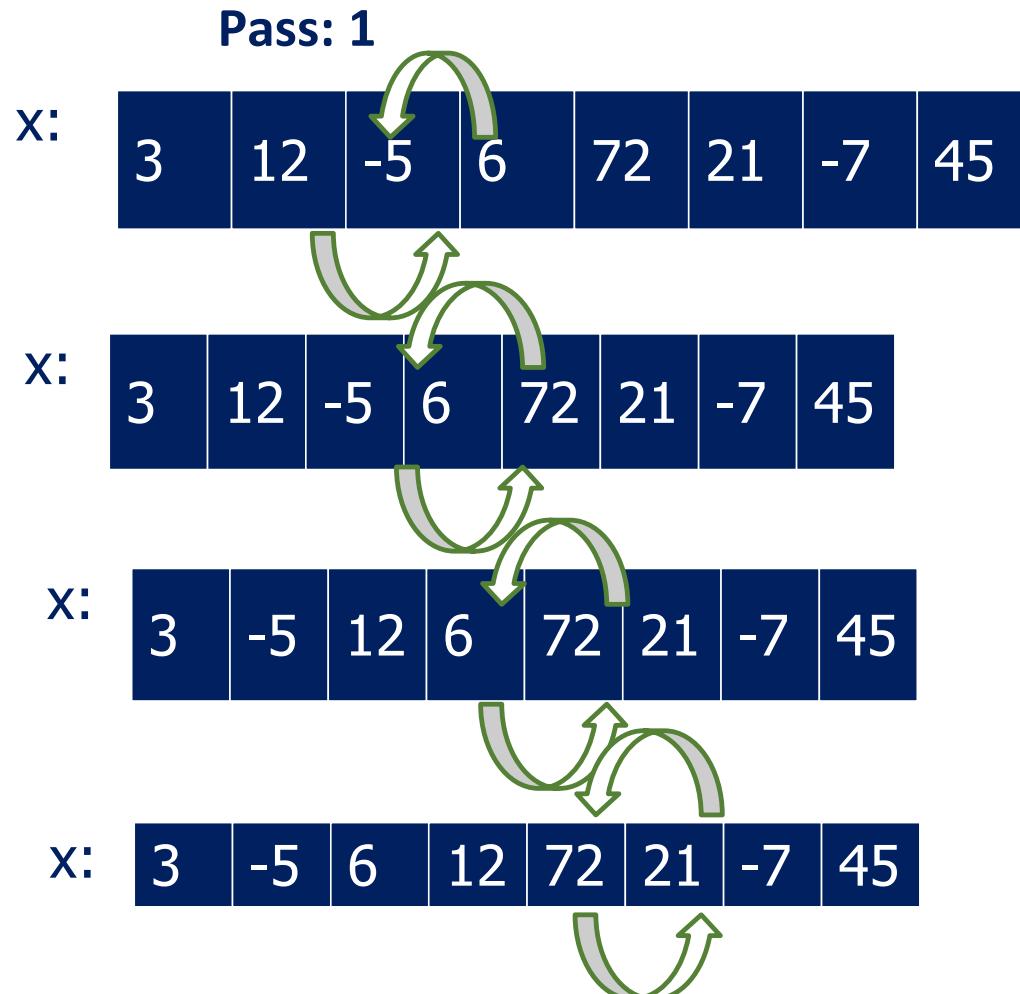
- In every pass we go on comparing neighbouring pairs, and swap them if out of order.
- In every pass, the largest of the elements under considering will *bubble* to the top (i.e., the right).

# Bubble Sort

## How the passes proceed?

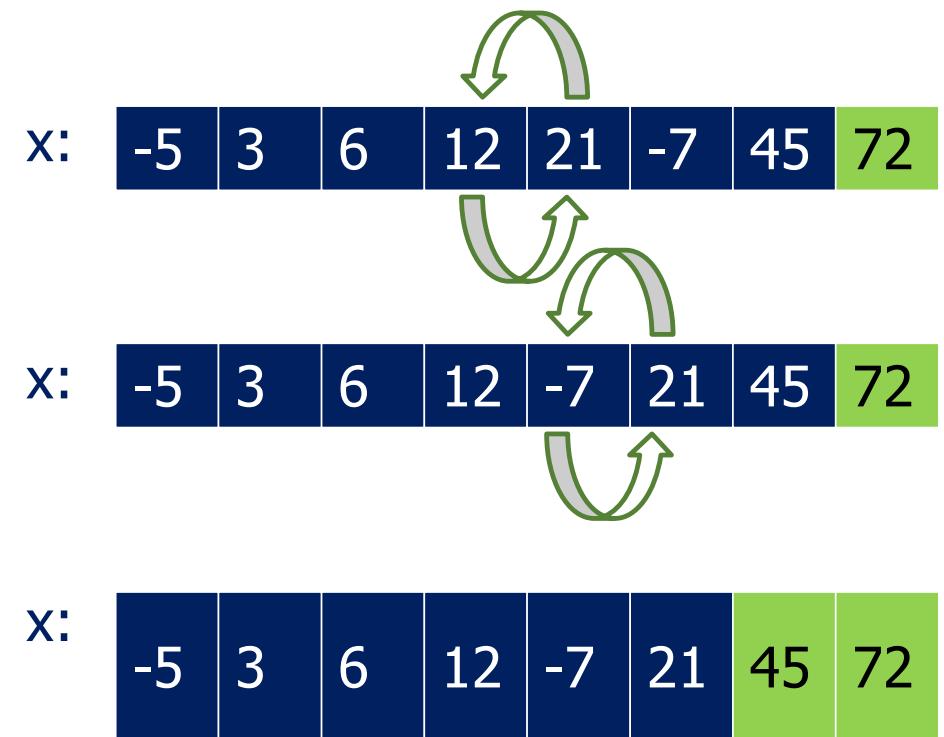
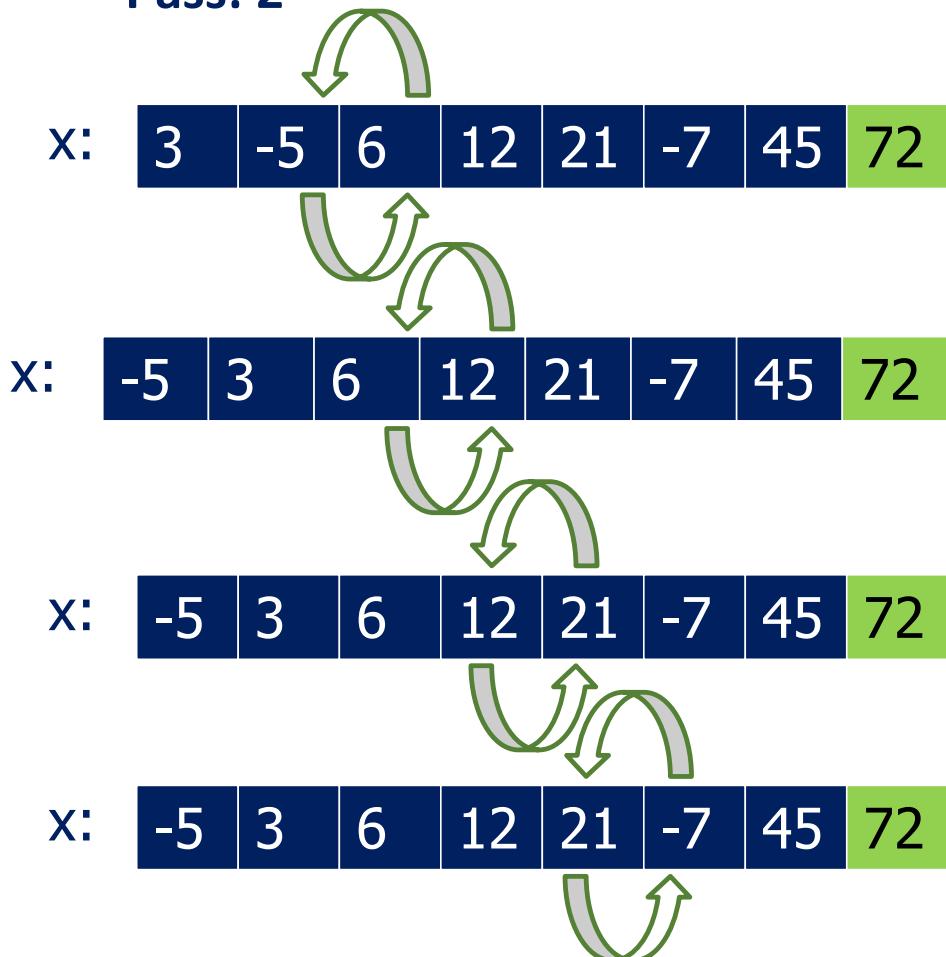
- In pass 1, we consider index 0 to n-1.
- In pass 2, we consider index 0 to n-2.
- In pass 3, we consider index 0 to n-3.
- .....
- .....
- In pass n-1, we consider index 0 to 1.

## Bubble Sort - Example



## Bubble Sort - Example

Pass: 2



# Bubble Sort

*Alg.:* BUBBLESORT(A)

```
for i ← length[A] to 1
    do for j ← 1 downto i
        do if A[j] > A[j +1]
            then exchange A[j] ↔ A[j+1]
```

# Bubble Sort – Complexity Analysis

*Alg.:* BUBBLESORT(A)

```
for i ← length[A] to 1
    do for j ← 1 downto i
```

Comparisons:  $\approx n^2/2$

**do if  $A[j] > A[j + 1]$**

Exchanges:  $\approx n^2/2$

**then exchange  $A[j] \leftrightarrow A[j+1]$**

Thus,  $T(n) = O(n^2)$

# Bubble Sort: Summary of Complexity analysis

Case	Comparisons	Remarks	Complexity	Remarks
Case 1	$n-1, 0(\text{swapping})$	Input list is in sorted order	$O(n)$	Best case
Case 2	$n(n-1)/2$	Input list is sorted in reverse order	$O(n^2)$	Worst case
Case 3		Input list is in random order	$O(n^2)$	Average case

# Bubble Sort

**How do you make best case with  $(n-1)$  comparisons only?**

- By maintaining a variable **flag**, to check if there has been any swaps in a given pass.
- If not, the array is already sorted.

# Efficient Sorting algorithms

Two of the most popular sorting algorithms are based on **divide-and-conquer** approach.

- Quick sort (will be covered next week)
- Merge sort

Basic concept of divide-and-conquer method:

```
sort (list)
{
    if the list has length greater than 1
    {
        Partition the list into lowlist and highlist;
        sort (lowlist);
        sort (highlist);
        combine (lowlist, highlist);
    }
}
```

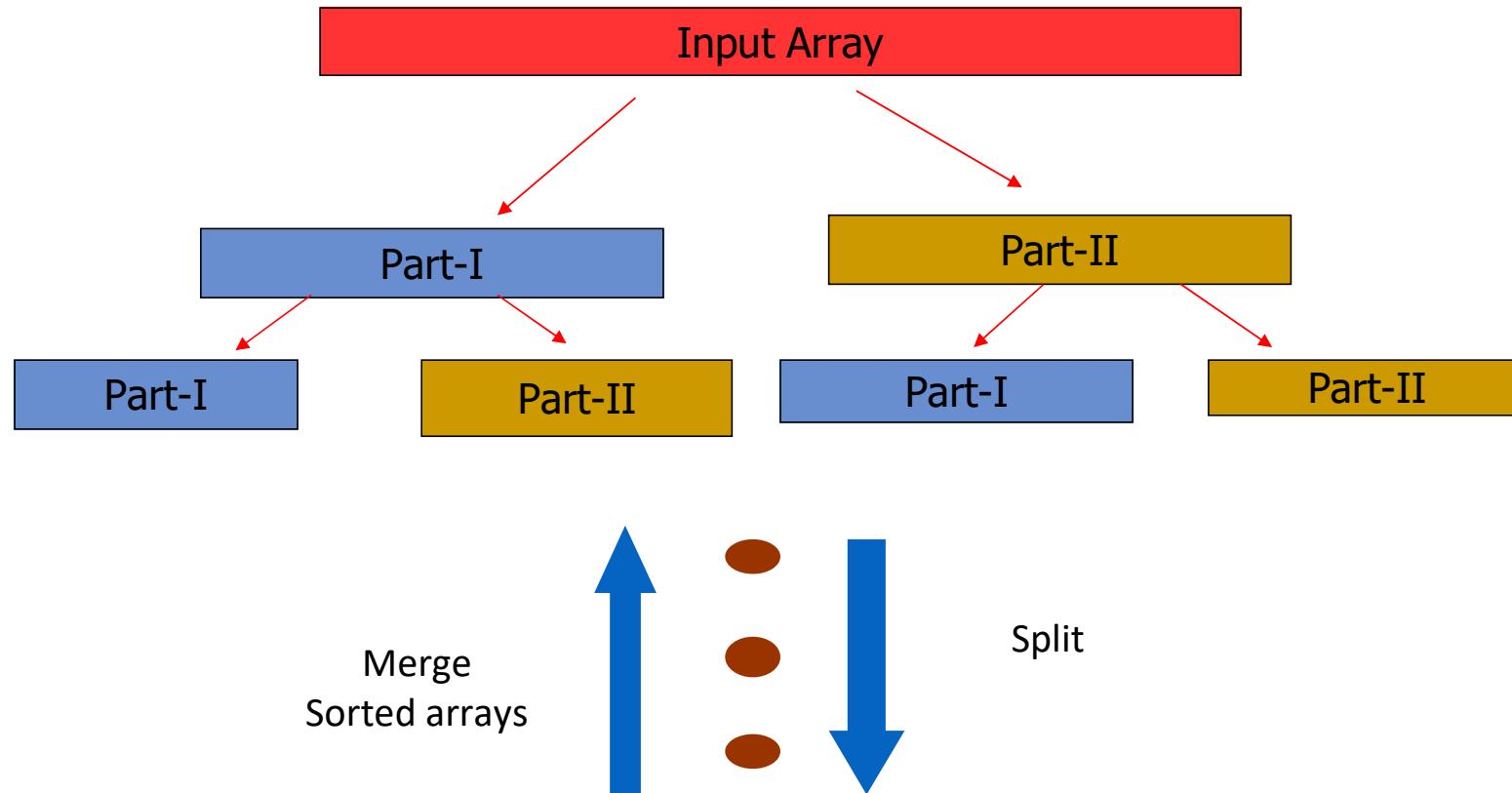
# Merge Sort

# Merge Sort Approach

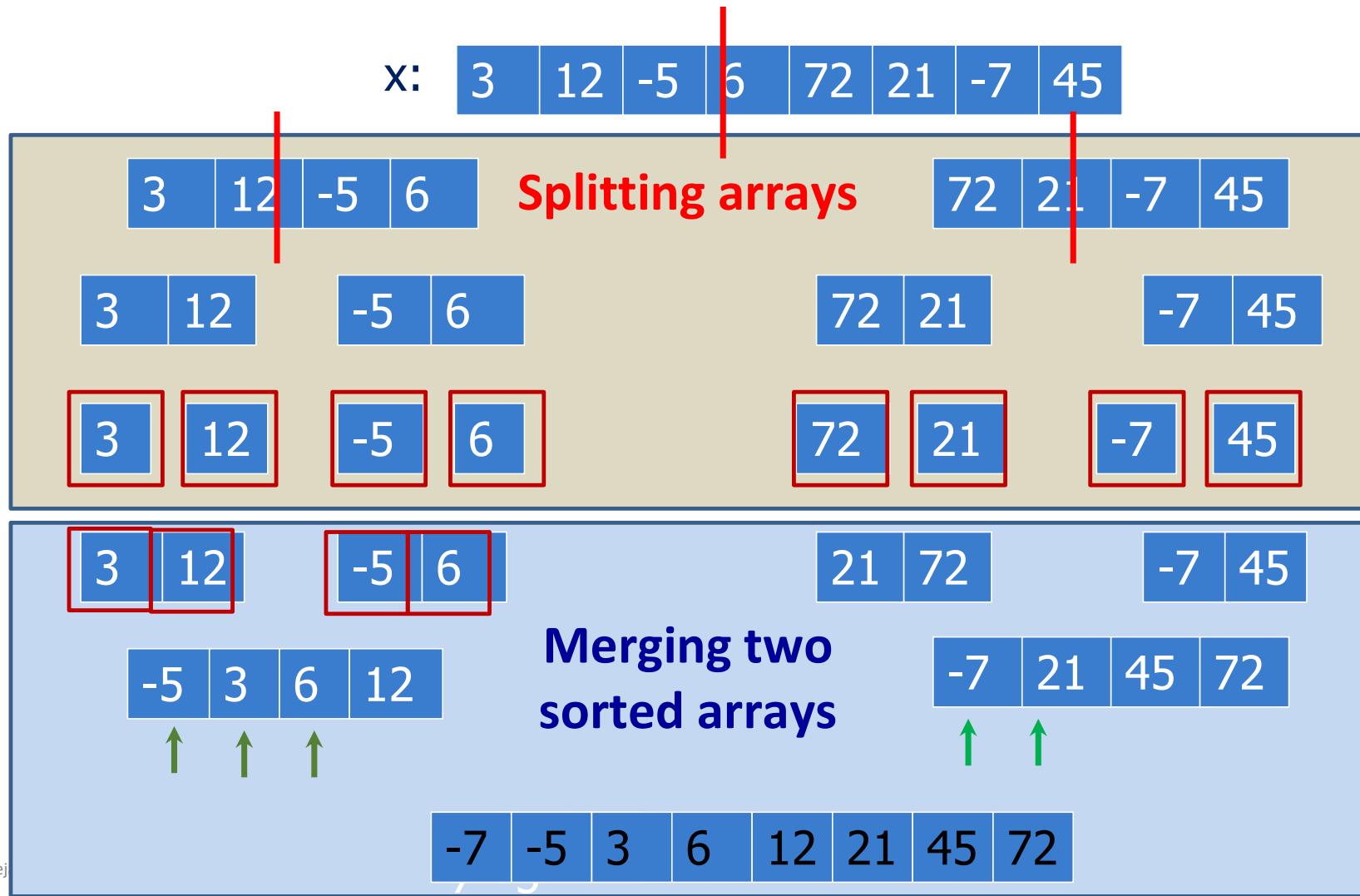
To sort an array  $A[p \dots r]$ :

- **Divide**
  - Divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

# Merge Sort – How it Works?



# Merge Sort - Example



# Merge Sort

*Alg.:* MERGE-SORT( $A, p, r$ )

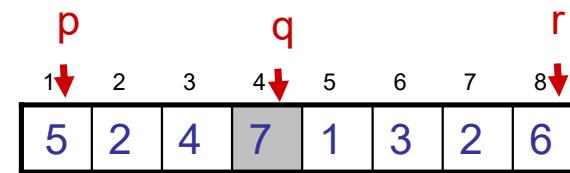
if  $p < r$

then  $q \leftarrow \lfloor(p + r)/2\rfloor$  ← DIVIDE

MERGE-SORT( $A, p, q$ ) ← CONQUER

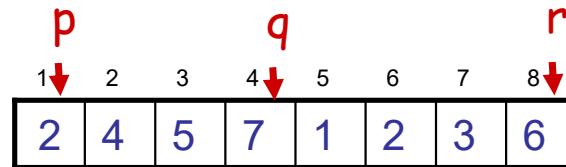
MERGE-SORT( $A, q + 1, r$ ) ← CONQUER

MERGE( $A, p, q, r$ ) ← COMBINE



- Initial call: MERGE-SORT( $A, 1, n$ )

## MERGE(A, p, q, r)

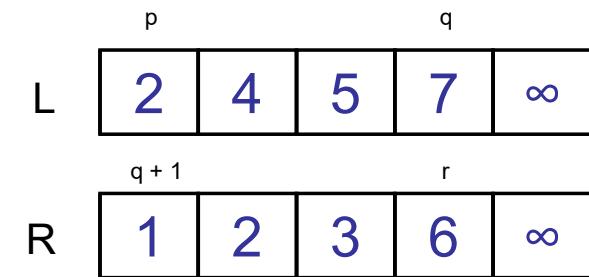
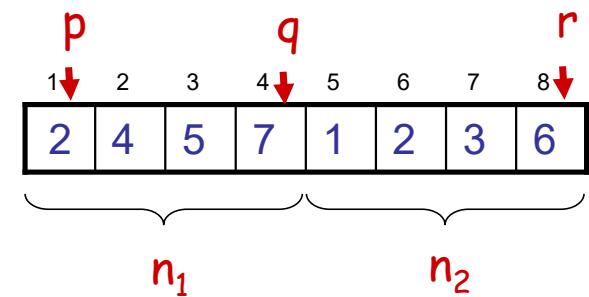


- **Input:** Array  $A$  and indices  $p, q, r$  such that  $p \leq q < r$ 
  - Subarrays  $A[p \dots q]$  and  $A[q + 1 \dots r]$  are sorted
- **Output:** One single sorted subarray  $A[p \dots r]$

# MERGE(A, p, q, r)

*Alg.:* MERGE(A, p, q, r)

1. Compute  $n_1$  and  $n_2$
2. Copy the first  $n_1$  elements into  $L[1 \dots n_1 + 1]$  and the next  $n_2$  elements into  $R[1 \dots n_2 + 1]$
3.  $L[n_1 + 1] \leftarrow \infty$ ;  $R[n_2 + 1] \leftarrow \infty$
4.  $i \leftarrow 1$ ;  $j \leftarrow 1$
5. **for**  $k \leftarrow p$  **to**  $r$
6.   **do if**  $L[i] \leq R[j]$
7.     **then**  $A[k] \leftarrow L[i]$
8.       *i*  $\leftarrow i + 1$
9.     **else**  $A[k] \leftarrow R[j]$
10.      *j*  $\leftarrow j + 1$



# MERGE-SORT Running Time

- **Divide:**
  - compute  $q$  as the average of  $p$  and  $r$ :  $D(n) = O(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size  $n/2 \Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an  $n$ -element subarray takes  $O(n)$  time  $\Rightarrow C(n) = O(n)$

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

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

# Merge Sort - Discussion

- Advantages:
  - Guaranteed to run in  $O(n \log n)$
- Disadvantage
  - Requires extra space  $\approx N$

# Choosing suitable sorting algorithms

Problem1: Sort a file of huge records with tiny keys  
(Ex: reorganize your MP-3 files)

Which method to use?

- A. insertion sort
- B. selection sort
- C. bubble sort
- D. merge sort

# Choosing suitable sorting algorithms

Problem1: Sort a file of huge records with tiny keys  
(Ex: reorganize your MP-3 files)

- Insertion sort or bubble sort?
  - NO, too many exchanges
- Selection sort?
  - YES, it takes linear time for exchanges
- Merge sort or custom method?
  - Probably not: selection sort simpler, does less swaps

## Choosing suitable sorting algorithms

Problem2: Sort a huge randomly-ordered file of small records  
(Ex: Process transaction record for a phone company)

Which method to use?

- A. insertion sort
- B. selection sort
- C. bubble sort
- D. merge sort

# Choosing suitable sorting algorithms

Problem2: Sort a huge randomly-ordered file of small records  
(Ex: Process transaction record for a phone company)

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion sort?
  - NO, quadratic time for randomly-ordered keys
- Merge sort?
  - YES, it is designed for this problem

# Choosing suitable sorting algorithms

Problem3: sort a file that is already almost in order

EX: Re-sort a huge database after a few changes

Which method to use?

- A. insertion sort
- B. selection sort
- C. bubble sort
- D. merge sort

# Choosing suitable sorting algorithms

Problem3: sort a file that is already almost in order

EX: Re-sort a huge database after a few changes

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, bad for some definitions of *almost in order*
- Insertion sort?
  - YES, takes linear time for most definitions of almost in order
- Merge sort?
  - Probably not: insertion sort simpler and faster

# Any question?



# SORTING ALGORITHMS

---

# Outline

---

- Quick sort
- Sorting in linear time
  - Decision Tree, Counting Sort
  - Radix Sort
  - Bucket Sort
- Choosing Suitable Sorting Algorithms

# Quick sort

---

- Divide-and-conquer
  - Partition array into two sub-arrays, recursively sort.
  - All of first sub-array < all of second < all of second sub-array

**Divide:** Partition (separate) the array  $A[p..r]$  into two (possibly empty) subarrays  $A[p..q-1]$  and  $A[q+1..r]$ . . Index  $q$  is computed here, which is called **pivot**.

- Each element in  $A[p..q-1] < A[q]$ .
- $A[q] <$  each element in  $A[q+1..r]$ .
- Index  $q$  is computed as part of the partitioning procedure.

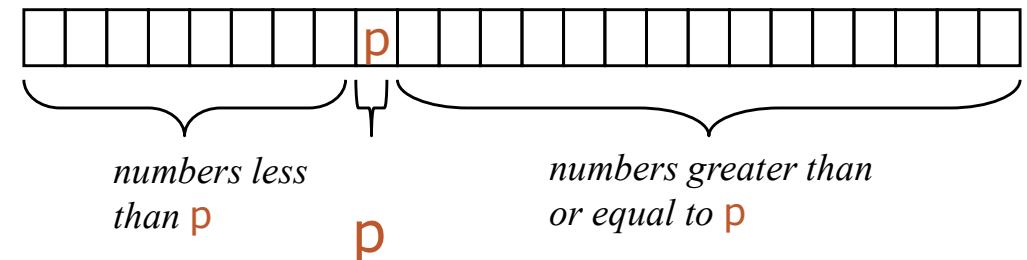
**Conquer:** Sort the two subarrays by recursive calls to quicksort.

**Combine:** The subarrays are sorted in place – no work is needed to combine them

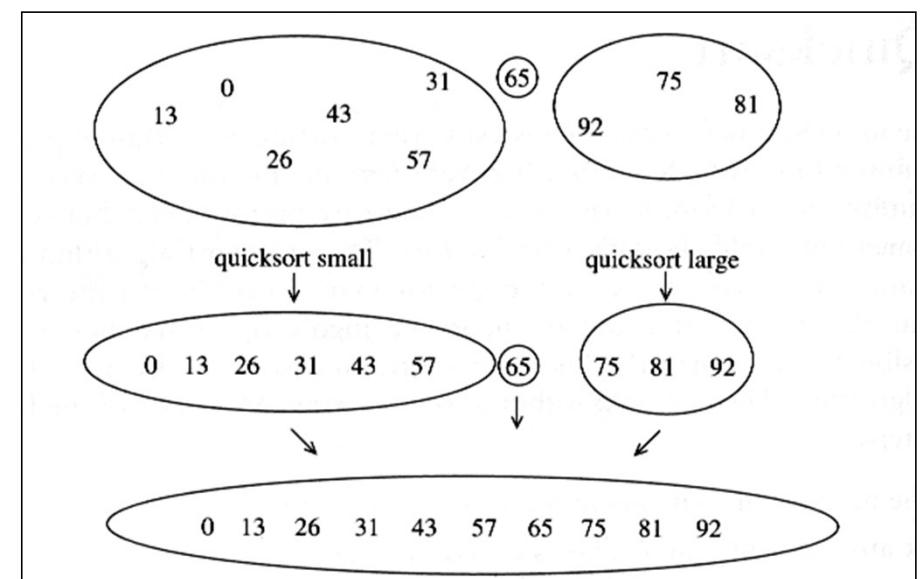
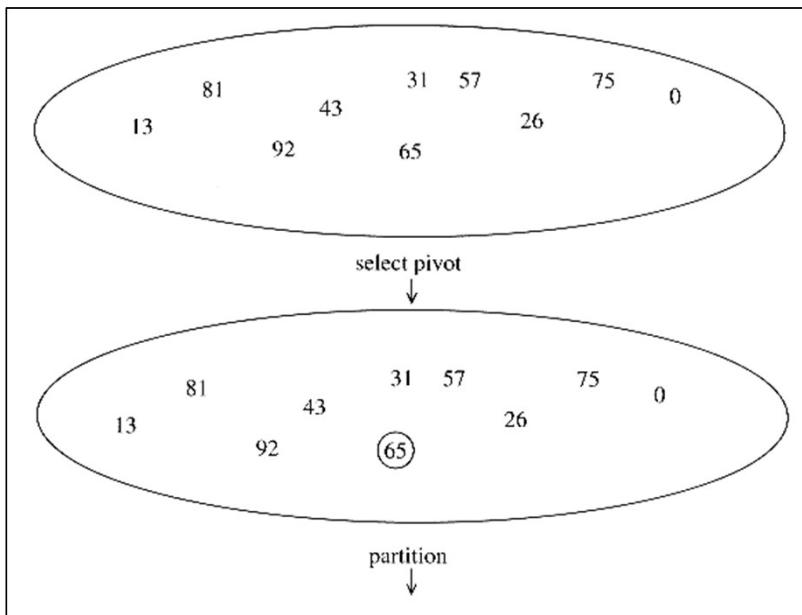
# Basic idea

---

- At every step, we select a pivot element in the list (usually the first element).
  - Move all numbers less than  $p$  to the beginning of the array
  - Move all numbers greater than (or equal to)  $p$  to the end of the array
- 
- Quicksort the numbers less than  $p$
  - Quicksort the numbers greater than or equal to  $p$



# Example of Quick Sort...



## Example<sup>1</sup>

Input: 45 -56 78 90 -3 -6 123 0 -3 45 69 68

45 -56 78 90 -3 -6 123 0 -3 45 69 68

-6 -56 -3 0 -3

45

123 90 78 45 69 68

-56 -6 -3 0 -3

68 90 78 45 69

123

-3 0 -3

45

68

78 90 69

-3 0

69 78 90

Output: -56 -6 -3 -3 0 45 45 68 69 78 90 123

# Quick Sort: Algorithm

---

```
Quicksort (A[L..R])
```

```
    if L < R then
```

```
        pivot = Partition( A[L..R] )
```

```
        Quicksort (A[1..p-1])
```

```
        Quicksort (A[p+1...R])
```

```
Partition (A[L..R])
```

```
    p ← A[L]; i ← L; j ← R + 1
```

```
    while (i < j) do {
```

```
        repeat i ← i + 1 until A[i] ≥ p
```

```
        repeat j ← j - 1 until A[j] ≤ p
```

```
        if (i < j) then swap(A[i], A[j])
```

```
}
```

```
    swap(A[j],A[L])
```

```
    return j
```

# Quick Sort: Complexity analysis

---

Case-1: The pivot is the smallest element, all the time -> Partition is always unbalanced

**Worst-Case Scenario**

$$\begin{aligned} T(N) &= T(N-1) + cN \\ T(N-1) &= T(N-2) + c(N-1) \\ T(N-2) &= T(N-3) + c(N-2) \\ &\vdots \\ T(2) &= T(1) + c(2) \\ T(N) &= T(1) + c \sum_{i=2}^N i = O(N^2) \end{aligned}$$

Worst-Case Scenario

# Quick Sort: Complexity Analysis

---

- Partition is perfectly balanced- >Pivot is always in the middle (median of the array).

**Best-Case Scenario**

$$T(N) = T(N/2) + T(N/2) + cN = 2T(N/2) + cN$$

This recurrence is similar to the merge sort recurrence.  $O(N \log N)$

# Complexity Analysis

---

**Best Case:** every partition splits half of the array

$$O(n \log_2 n)$$

**Worst Case:** one array is empty; one has all elements

$$O(n^2)$$

<sup>1</sup>Average case:

$$O(n \log_2 n)$$

<sup>1</sup>Proof: pp 272–273, Data Structures and Algorithm Analysis by M. A. Weiss, 2<sup>nd</sup> edition

# SORTING IN LINEAR TIME

---

# The decision-tree model

---

- Algorithms that can sort  $n$  numbers in  $O(n \log n)$  time:

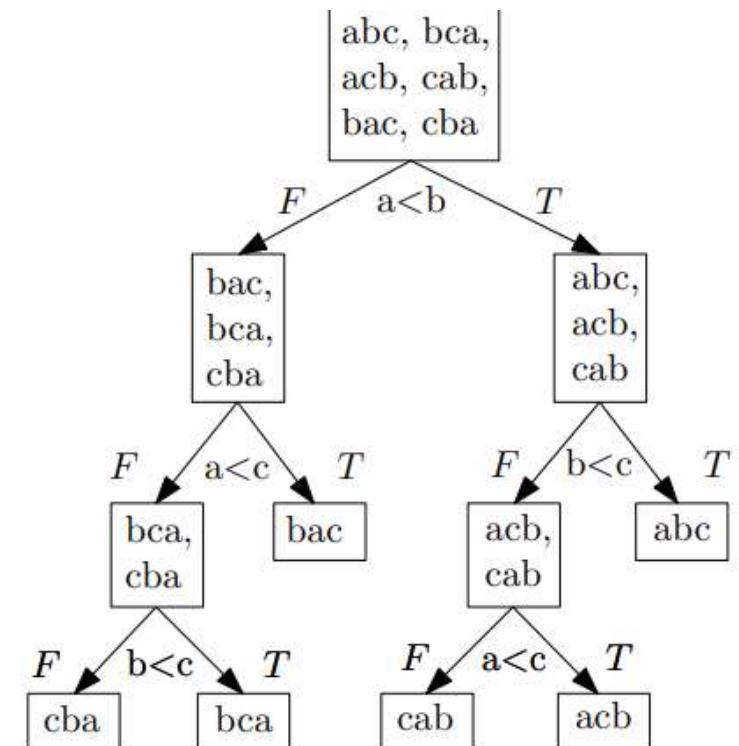
- Merge sort and heapsort
- quicksort achieves it on best and average case

- ✓ Comparison Sorts

- In a comparison sort algorithm, the sorted order is determined by a sequence of comparisons between pairs of elements.
- Comparison sorts can be viewed abstractly in terms of decision trees
- Using a decision tree, we show that every comparison sort requires  $\Omega(n \log n)$  comparisons in the worst-case.

# Decision Tree

1. Represents every sequence of comparisons that an algorithm might make on an input of size  $n$
2. Nodes annotated with the orderings consistent with the comparisons made so far.
3. Edges denote the result of a single comparison. Total order at leaves



Algorithm: Insertion sort. Instance ( $n = 3$ ):  
the numbers a, b, c.

# Lower bound for the worst case:

- The worst-case number of comparisons for a comparison sort corresponds to the height of its decision tree.

➤ **Claim :**

The depth of a decision tree for a given value of  $n$  is  
 $\Omega(n \log n)$ .

➤ **Proof:**

There are  $n!$  leaves. A tree of height  $h$  has at most  $2^{h+1}$  nodes. So

$$2^{h+1} \geq n!$$

$$\begin{aligned} h + 1 &\geq \log_2 n! = \log_2 (1 \cdot 2 \cdot \dots \cdot n) \\ &= \log_2 1 + \log_2 2 + \dots + \log_2 n > (n/2) \log_2 \end{aligned}$$

$$(n/2)$$

$$h \in \Omega(n \log n)$$

# Lower Bound (cont'd)

## Theorem

Every comparison sort requires  $\Omega(n \log n)$  comparisons in the worst-case.

## Proof

Given a comparison sort, we look at the decision tree it generates on a inputs of size  $n$ .

- Each path from root to leaf is one possible sequence of comparisons.
- Length of the path is the number of comparisons for that instance.
- Height of the tree is the worst-case path length (number of comparisons)

Height of the tree is  $\Omega(n \log n)$  by the previous claim. Hence, every comparison sort requires  $\Omega(n \log n)$  comparisons.

# Linear sorting algorithms

---

- Linear sorting algorithms
  - Counting Sort
  - Radix Sort
  - Bucket sort
- These apply only when the input has a special structure, e.g., inputs are integers.
- Make certain assumptions about the data
- Linear sorts are NOT “comparison sorts”

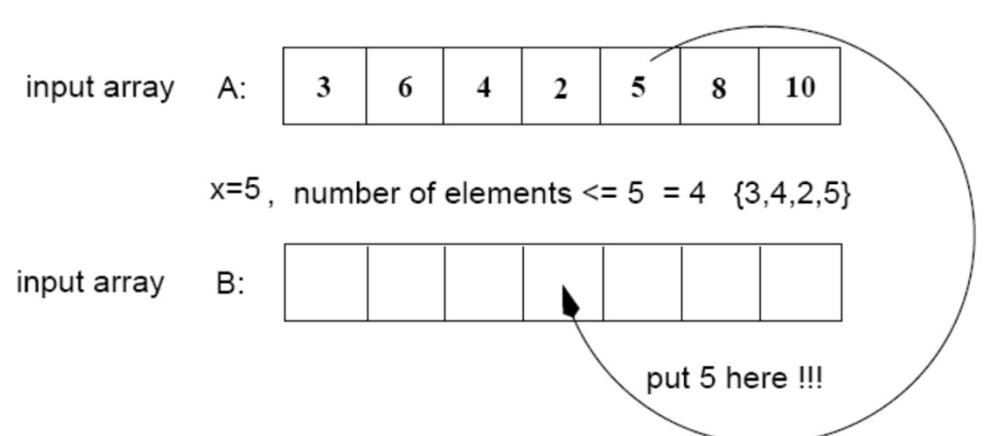
# Counting Sort

## Assumptions:

- $n$  integers which are in the range [0 ...  $r$ ]
- $r$  is in the order of  $n$ , that is,  $r=O(n)$

## Idea:

- For each element  $x$ , find the number of elements  $\leq x$
- Place  $x$  into its correct position in the output array



# Example

## Step 1

Find the number of times  $A[i]$  appears in  $A$

input array A:

3	6	4	1	3	4	1	4
---	---	---	---	---	---	---	---

allocate C

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

i=1, A[1]=3

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

C[A[1]]=C[3]=1

i=2, A[2]=6

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

C[A[2]]=C[6]=1

i=3, A[3]=4

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

C[A[3]]=C[4]=1

.

.

i=8, A[8]=4

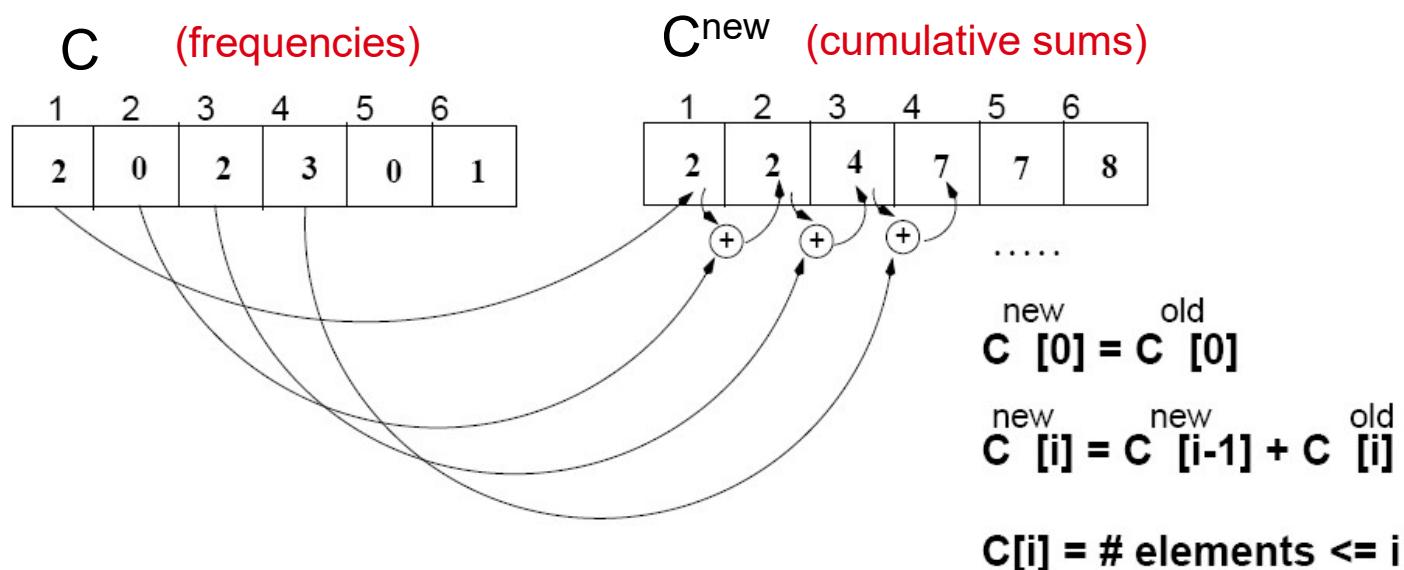
1	2	3	4	5	6
2	0	2	3	0	1

C[A[8]]=C[4]=3

$C[i] = \text{number of times element } i \text{ appears in } A$

# Example

Step 2 Find the number of elements  $\leq A[i]$ ,



## Another Example

---

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2) Modify the count array such that each element at each index stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in the output sequence.

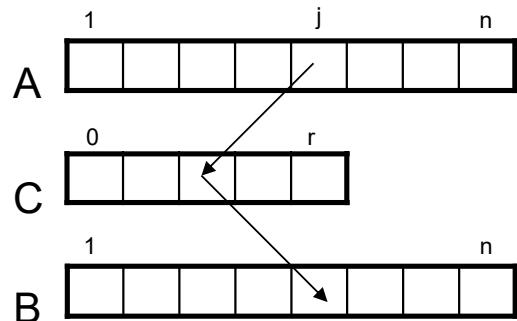
3) Output each object from the input sequence followed by decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place next data 1 at an index 1 smaller than this index.

# Algorithm

*Alg.:* COUNTING-SORT( $A, B, n, r$ )



1.      **for**  $i \leftarrow 0$  **to**  $r$       }  $O(r)$   
2.          **do**  $C[i] \leftarrow 0$
3.      **for**  $j \leftarrow 1$  **to**  $n$       }  $O(n)$   
4.          **do**  $C[A[j]] \leftarrow C[A[j]] + 1$   
5.           $C[i]$  contains the number of elements equal to  $i$
6.      **for**  $i \leftarrow 1$  **to**  $r$       }  $O(r)$   
7.          **do**  $C[i] \leftarrow C[i] + C[i - 1]$   
8.           $C[i]$  contains the number of elements  $\leq i$
9.      **for**  $j \leftarrow n$  **downto** 1  
10.         **do**  $B[C[A[j]]] \leftarrow A[j]$   
11.             $C[A[j]] \leftarrow C[A[j]] - 1$

---

Overall time:  $O(n + r)$

# Analysis of Counting Sort

---

- Overall time:  $O(n + r)$
- In practice we use COUNTING sort when  $r = O(n)$

⇒ running time is  $O(n)$

# SORTING ALGORITHMS

---

# Outline

---

- Sorting in linear time

- Radix Sort
- Bucket Sort
- Choosing Suitable Sorting Algorithms

# Radix Sort

---

- Represents keys as  $d$ -digit(s) numbers in some base- $k$
- (e.g., decimal representation  $k=10$ )
- Every digit has  $k$  values ( $k = 10$  for decimal,  $k = 2$  for binary)

$\text{key} = x_1x_2\dots x_d \text{ where } 0 \leq x_i \leq k-1$

Example:  $\text{key}=15$

$\text{key}_{10} = 15, d=2, k=10 \text{ where } 0 \leq x_i \leq 9$

$\text{key}_2 = 1111, d=4, k=2 \text{ where } 0 \leq x_i \leq 1$

# Basic idea

---

- **Assumptions**

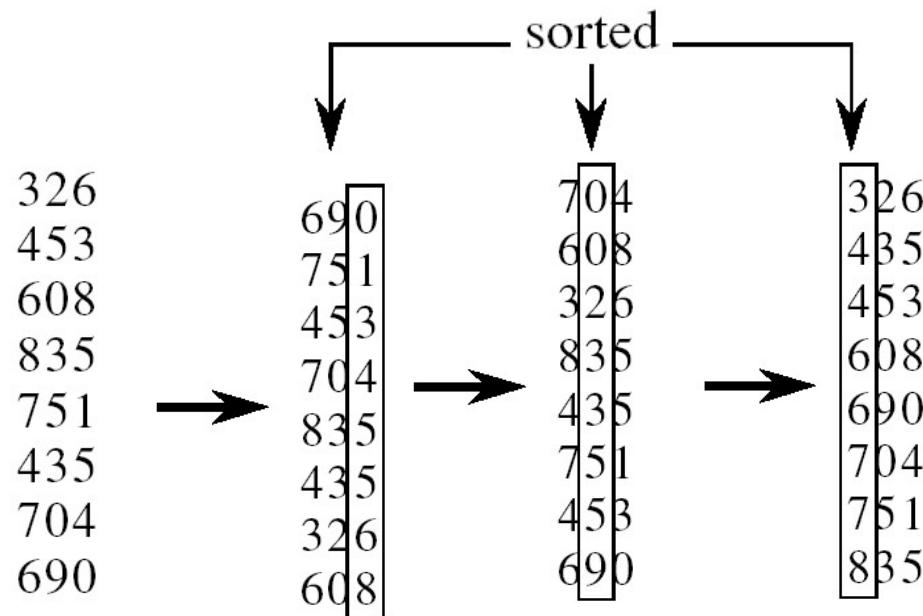
$d=O(1)$  and  $k=O(n)$

- **Sorting looks at one column at a time**
  - For a  $d$  digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit, until all digits have been sorted
  - Requires only  $d$  passes through the list

We would sort the integers by their least significant digits first using counting sort

# Example

---



---

Alg.: RADIX-SORT( $A$ ,  $d$ )

for  $i \leftarrow 1$  to  $d$

do use a **stable** sort to sort array  $A$  on digit  $i$

(stable sort: preserves order of identical elements)

# Complexity analysis

---

Given  $n$  numbers of  $d$  digits each, where each digit may take up to  $k$  (base) possible values,

RADIX-SORT correctly sorts the numbers in  $O(d(n+k))$ .

- One pass of sorting per digit takes  $O(n+k)$  assuming that we use **counting sort**
- There are  $d$  passes (for each digit)

Assuming  $d=O(1)$  and  $k=O(n)$ , running time is  $O(n)$

# Bucket Sort

---

Assumption:

- the input is generated by a random process that distributes elements uniformly over  $[0, 1]$

Idea:

- Divide  $[0, 1]$  into  $k$  equal-sized buckets ( $k=\Theta(n)$ )
- Distribute the  $n$  input values into the buckets
- Sort each bucket (e.g., using quicksort)
- Go through the buckets in order, listing elements in each one

**Input:**  $A[1 \dots n]$ , where  $0 \leq A[i] < 1$  for all  $i$

**Output:** elements  $A[i]$  sorted

# SORTING ALGORITHMS

---

Content Acknowledgments: [Dr. George Bebis](#) , CS477/677 Analysis of Algorithms

# Outline

---

- Sorting in linear time
  - Radix Sort
  - Bucket Sort
  - Choosing Suitable Sorting Algorithms

# Radix Sort

---

- Represents keys as  $d$ -digit(s) numbers in some base- $k$
- (e.g., decimal representation  $k=10$ )
- Every digit has  $k$  values ( $k = 10$  for decimal,  $k = 2$  for binary)

$\text{key} = x_1x_2\dots x_d \text{ where } 0 \leq x_i \leq k-1$

Example:  $\text{key}=15$

$\text{key}_{10} = 15, d=2, k=10 \text{ where } 0 \leq x_i \leq 9$

$\text{key}_2 = 1111, d=4, k=2 \text{ where } 0 \leq x_i \leq 1$

# Basic idea

---

- **Assumptions**

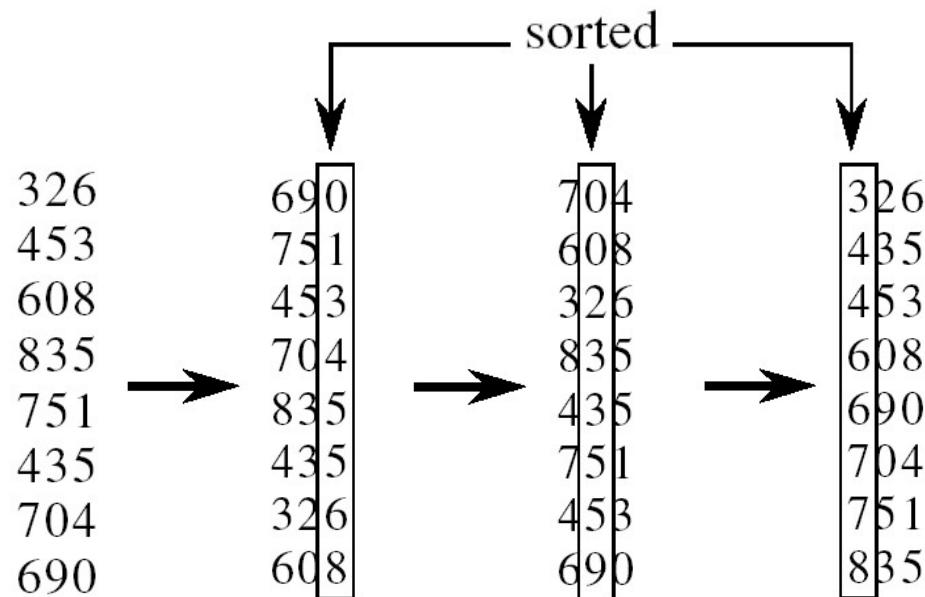
$d=O(1)$  and  $k=O(n)$

- **Sorting looks at one column at a time**
  - For a  $d$  digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit, until all digits have been sorted
  - Requires only  $d$  passes through the list

We would sort the integers by their least significant digits first using counting sort

# Example

---



---

Alg.: RADIX-SORT( $A$ ,  $d$ )

for  $i \leftarrow 1$  to  $d$

do use a **stable** sort to sort array  $A$  on digit  $i$

(stable sort: preserves order of identical elements)

# Complexity analysis

---

Given  $n$  numbers of  $d$  digits each, where each digit may take up to  $k$  possible values, RADIX-SORT correctly sorts the numbers in  $O(d(n+k))$ .

- One pass of sorting per digit takes  $O(n+k)$  assuming that we use **counting sort**
- There are  $d$  passes (for each digit)

Assuming  $d=O(1)$  and  $k=O(n)$ , running time is  $O(n)$

# Bucket Sort

---

Assumption:

- the input is generated by a random process that distributes elements uniformly over  $[0, 1]$

Idea:

- Divide  $[0, 1]$  into  $k$  equal-sized buckets ( $k=\Theta(n)$ )
- Distribute the  $n$  input values into the buckets
- Sort each bucket (e.g., using quicksort)
- Go through the buckets in order, listing elements in each one

**Input:**  $A[1 \dots n]$ , where  $0 \leq A[i] < 1$  for all  $i$

**Output:** elements  $A[i]$  sorted

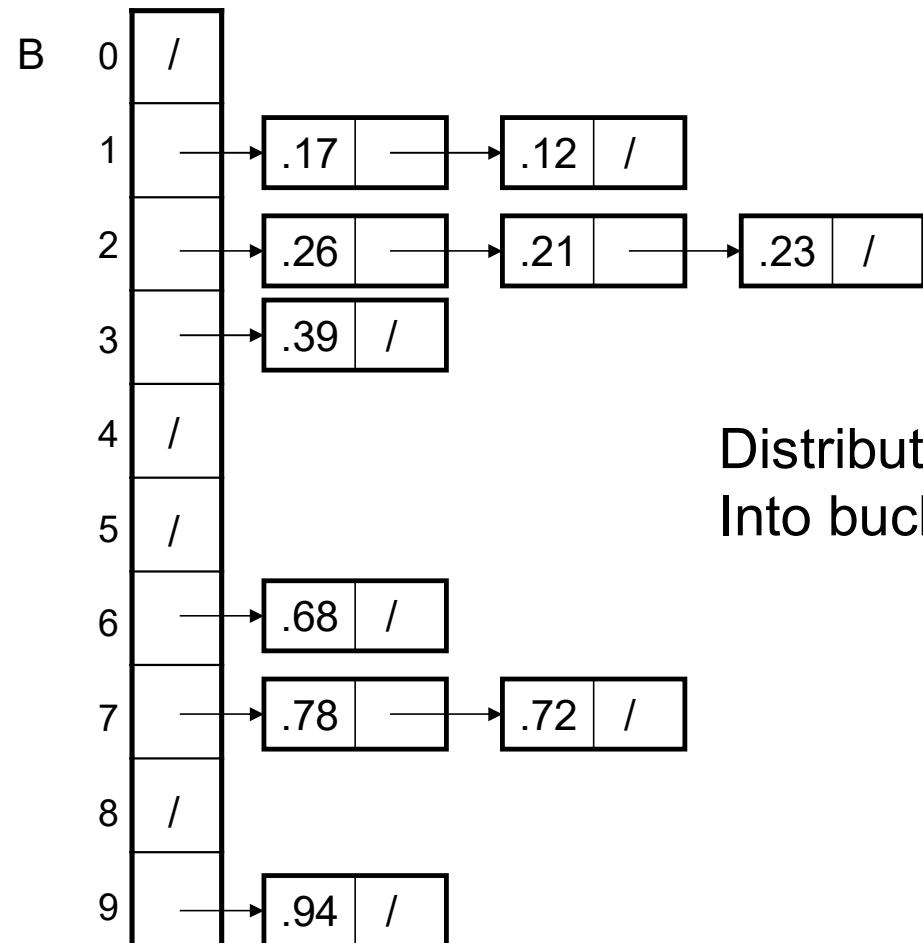
# Basic idea

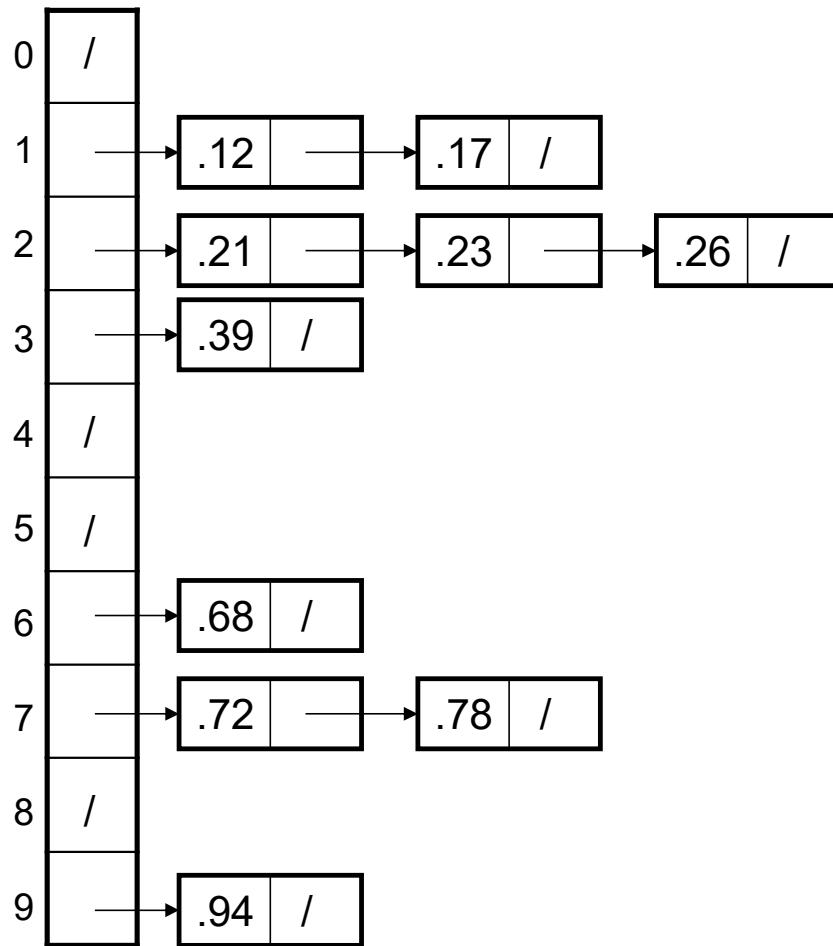
---

- Bucket sort can be used to sort such real numbers in average case linear complexity
- The idea is similar to direct sequence hashing and counting sort.
- We have an array of  $n$  pointers. Each position has a pointer pointing to a linked list.
- We have a function which generates an integer from a real number, and adds the real number in the linked list at the position.

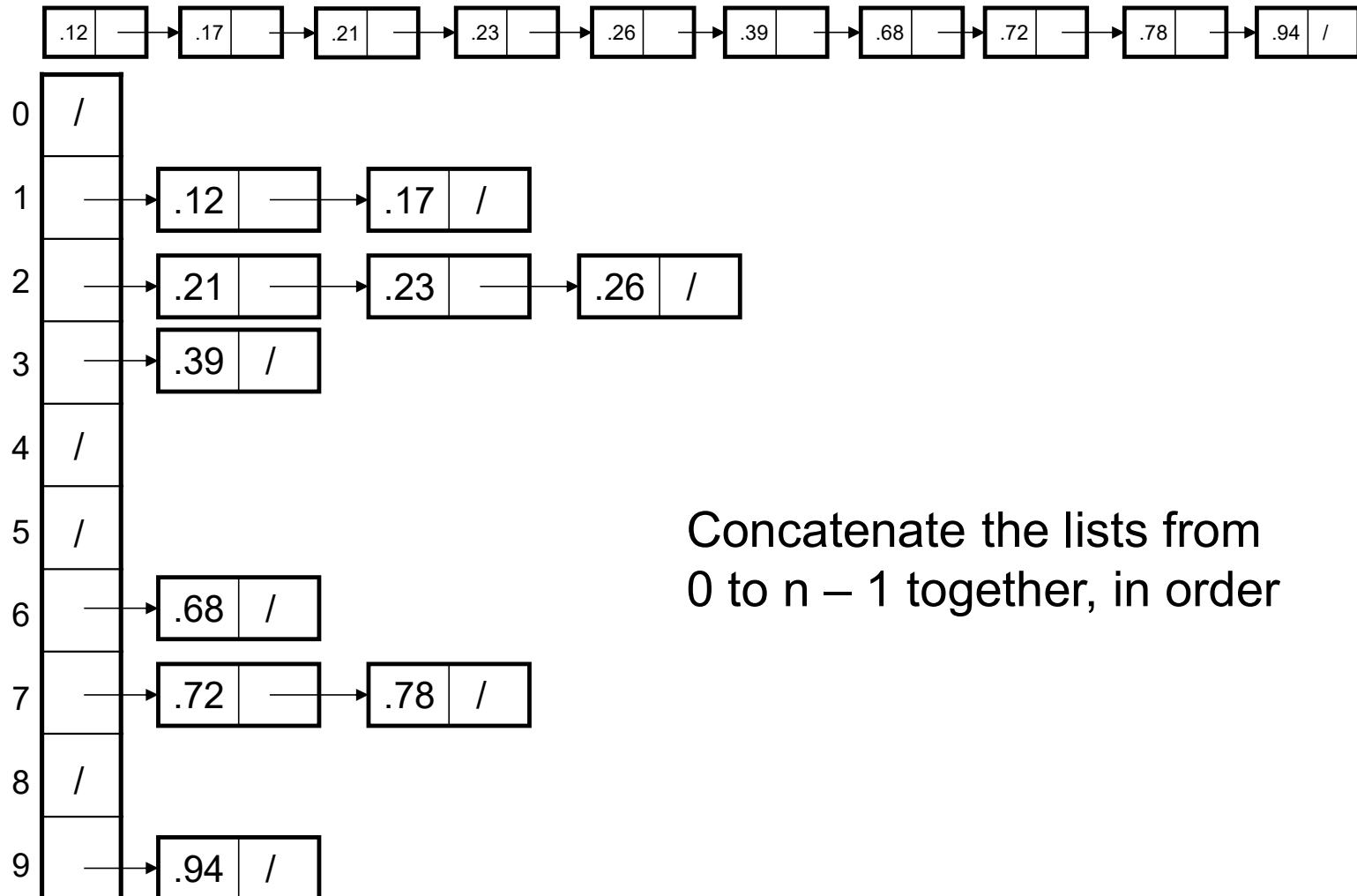
## Example - Bucket Sort

A	1	.78
	2	.17
	3	.39
	4	.26
	5	.72
	6	.94
	7	.21
	8	.12
	9	.23
	10	.68





Sort within each  
bucket



Concatenate the lists from  
0 to  $n - 1$  together, in order

*Alg.:* BUCKET-SORT(A, n)

**for**  $i \leftarrow 1$  **to**  $n$

**do** insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$

**for**  $i \leftarrow 0$  **to**  $k - 1$

**do** sort list  $B[i]$  with quicksort sort

concatenate lists  $B[0], B[1], \dots, B[n - 1]$

together in order

**return** the concatenated lists

$O(n)$

$k O(n/k \log(n/k))$   
 $=O(n \log(n/k))$

$O(k)$

---

$O(n)$  (if  $k = \Theta(n)$ )

## Contd...

---

There are  $n$  real numbers and  $n$  positions in the array, then each linked has roughly a constant number  $c$ .

- The sorting complexity for a linked list is a constant.
- Thus overall complexity is  $O(n)$

# In-place Sorting Algorithms

---

1. An in-place algorithm transforms the input without using any extra memory. As the algorithm executes, the input is usually overwritten by the output, and no additional space is needed for this operation.
2. An in-place algorithm may require a small amount of extra memory for its operation. However, the amount of memory required must not be dependent on the input size and should be constant.
3. Several sorting algorithms rearrange the input into sorted order in-place, such as **insertion sort**, **selection sort**, **quick sort**, **bubble sort**, **heap sort**, etc. All these algorithms require a constant amount of extra space for rearranging the elements in the input array.

# Not-in-place / Out-of-place

---

1. An algorithm that is not in-place is called a not-in-place or out-of-place algorithm. Unlike an in-place algorithm, the extra space used by an out-of-place algorithm depends on the input size.
2. The standard merge sort algorithm is an example of out-of-place algorithm as it requires  $O(n)$  extra space for merging. The merging can be done in-place, but it increases the time complexity of the sorting routine.

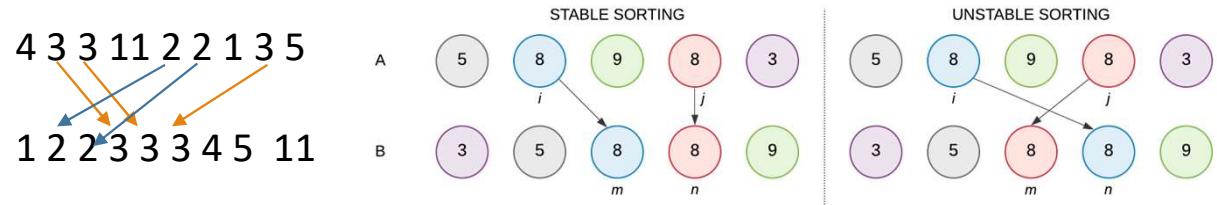
# SORTING ALGORITHMS

---

# In-place and Stable sorting Algorithms

## ■ Stable Sort :

- A sorting algorithm is said to be stable if the **ordering of identical keys in the input is preserved in the output.**



## ■ In-place Algorithms

- Some algorithms sort by swapping elements within the input array. Such algorithms are said to **sort in place**, and require only  $O(1)$  additional memory.

# In-place Sorting Algorithms

---

1. An in-place algorithm transforms the input **without using any extra memory**. As the algorithm executes, the input is usually overwritten by the output, and no additional space is needed for this operation.
2. An in-place algorithm may require a small amount of extra memory for its operation. However, the amount of memory required must not be dependent on the input size and should be constant.
3. Several sorting algorithms rearrange the input into sorted order in-place, such as **insertion sort**, **selection sort**, **quick sort**, **bubble sort**, **heap sort**, etc. All these algorithms require a constant amount of extra space for rearranging the elements in the input array.

# Not-in-place / Out-of-place

---

1. An algorithm that is not in-place is called a not-in-place or out-of-place algorithm. Unlike an in-place algorithm, the extra space used by an out-of-place algorithm depends on the input size.
2. The standard merge sort algorithm is an example of out-of-place algorithm as it requires  $O(n)$  extra space for merging. The merging can be done in-place, but it increases the time complexity of the sorting routine.

Sorting Algorithms	In - Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Quick Sort	Yes	No
Merge Sort	No (because it requires an extra array to merge the sorted subarrays)	Yes
Heap Sort	Yes	No

<sup>1</sup><https://afteracademy.com/blog/comparison-of-sorting-algorithms>

## In-place and Stable sorting Algorithms <sup>1</sup>

### Selection Sort

Note: Subscripts are only used for understanding the concept.

Input :  $4_A \ 5 \ 3 \ 2 \ 4_B \ 1$

Output :  $1 \ 2 \ 3 \ 4_B \ 4_A \ 5$

# Choosing a Sorting Algorithm

---

To choose a sorting algorithm for a particular problem, consider the running time, space complexity, and the expected format of the input list.

*Criteria for choosing a sorting algorithm<sup>1</sup>*

Only a few items	Insertion Sort
Items are mostly sorted already	Insertion Sort
Concerned about worst-case scenarios	Heap Sort
Interested in a good average-case result	Quicksort
Items are drawn from a dense universe	Bucket Sort
Desire to write as little code as possible	Insertion Sort

<sup>1</sup><https://www.oreilly.com/library/view/algorithms-in-/9780596516246/ch04s09.html>

# Choosing a Sorting Algorithm

---

➤ While choosing a sorting algorithm we need consider the running time, space complexity, and the expected format of the input list.

➤ Example :

- Quicksort => a very fast algorithm but implementation is bit hard ;
- bubble sort => slow algorithm but is very easy to implement.
- To sort small sets of data => bubble sort may be a better option (quickly implementable)
- Larger datasets, => the speedup from quicksort might be worth the trouble implementing the algorithm.

Algorithm	Best-case	Worst-case	Average-case	Space Complexity	Stable?
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$\log n$ best, $n$ avg	Usually not*
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Counting Sort	$O(k + n)$	$O(k + n)$	$O(k + n)$	$O(k + n)$	Yes

<https://brilliant.org/wiki/sorting-algorithms/>

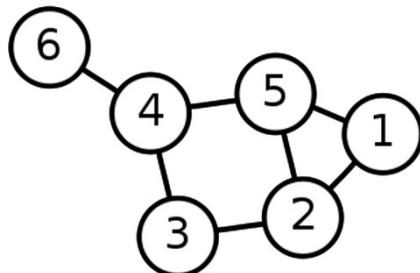
# GRAPHS

---

# Graph

- A Graph G, consists of a set of vertices, V a set of edges, E where each edge is associated with a pair of vertices

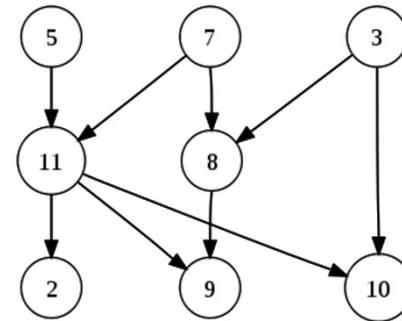
$G(V,E)$



Simple undirected graph

$G(V,E)$  :

Vertex set  $V = \{1, 2, 3, 4, 5, 6\}$   
Edge set  $E = \{\{1,2\}, \{1,5\}, \{2,3\}, \{2,5\}, \{3,4\}, \{4,5\}, \{4,6\}\}$ .



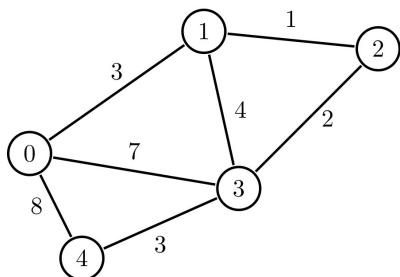
Simple directed graph:  
(If the pair is ordered, then the graph is **directed**.)

Vertex set  $V = \{2,3,5,7,8,9,10,11\}$   
Edge set  $E = \{\{3,8\}, \{3,10\}, \{5,11\}, \{7,8\}, \{7,11\}, \{8,9\}, \{11,2\}, \{11,9\}, \{11,10\}\}$ .

# Contd...

## Weighted Graphs

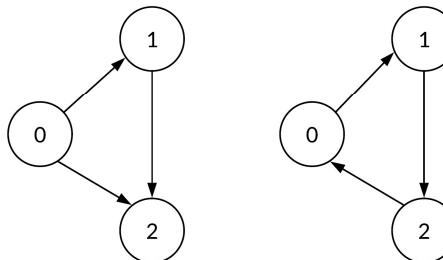
- A **weighted graph** is a **graph** in which each edge is given a numerical **weight**.



Acylic Graph

Cyclic Graph

- Cycles — Acyclic vs Cyclic Graphs



# Graph examples

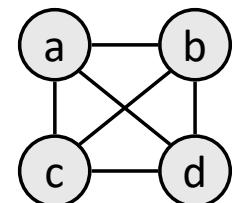
---

- For each, what are the vertices and what are the edges?
  - Airline routes
  - Family trees
  - Road maps (e.g., Google maps)
  - Methods in a program that call each other

# Graph Terminology

---

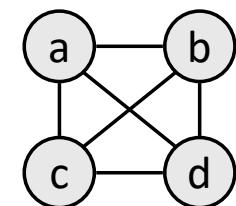
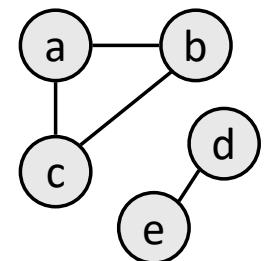
- Each edge is a pair  $(v, w)$ , where  $v, w \in V$
- A **path** in a graph is a sequence of vertices  $w_1, w_2, w_3, \dots, w_N$  such that  $(w_i, w_{i+1}) \in E$  for  $1 \leq i < N$ .
  - The number of edges on the path is **length** ( $N - 1$ ).
- If the graph contains an edge  $(v, v)$  from a vertex to itself, then the path  $v, v$  is sometimes referred to as a **loop**.



# Graph Terminology

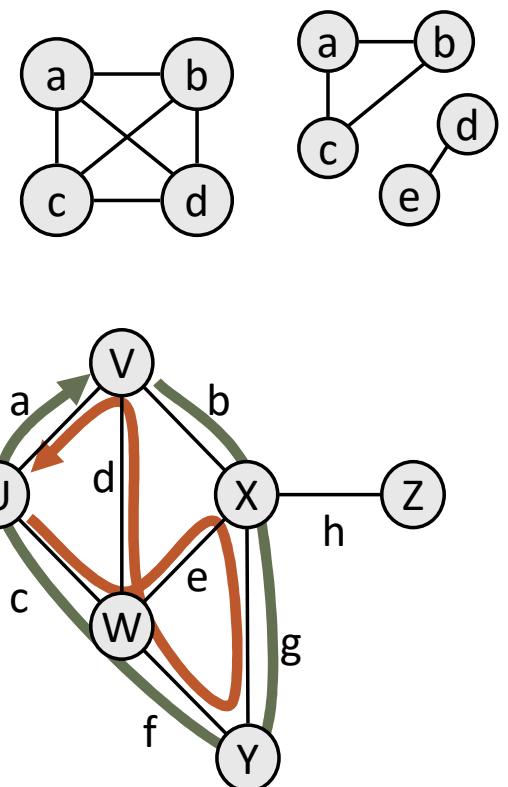
---

- **Adjacent nodes:** In a graph, if two nodes are connected by an edge then they are called adjacent nodes or neighbors.
- **Degree of the node:** The **number of edges that are connected to a particular node** is called the degree of the node.
  - In-degree : the number of edges coming to the vertex.
  - Out-degree: the number edges which are coming out from the vertex.
- **Reachable:** Vertex  $a$  is *reachable* from  $b$  if a path exists from  $a$  to



# Graph Terminology

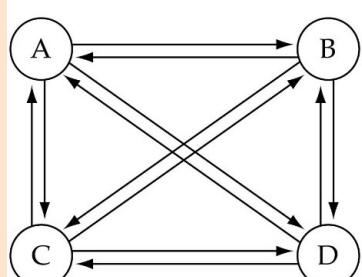
- **connected:** A graph is connected if every vertex is reachable from any other.
- **strongly connected:** When every vertex has an edge to every other vertex.
- **Distance between two Vertices:**
  - Number of edges that are available in the shortest path between vertex A and vertex B
- A **complete graph** is a graph in which there is an edge between every pair of vertices.
- **cycle :** A path of non-zero length from and to the same vertex with no repeated edges. Graph with no cycles: **acyclic**



# Graph terminology

- What is the number of edges in a complete directed graph with N vertices?

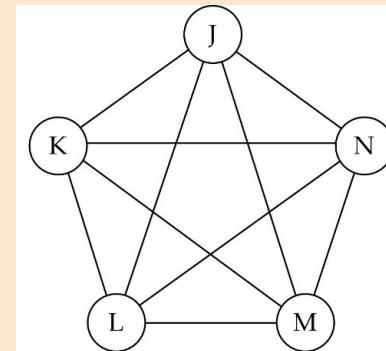
$$N * (N-1)$$



(a) Complete directed graph.

- What is the number of edges in a complete undirected graph with N vertices?

$$N * (N-1) / 2$$



(b) Complete undirected graph.

# Representing graphs

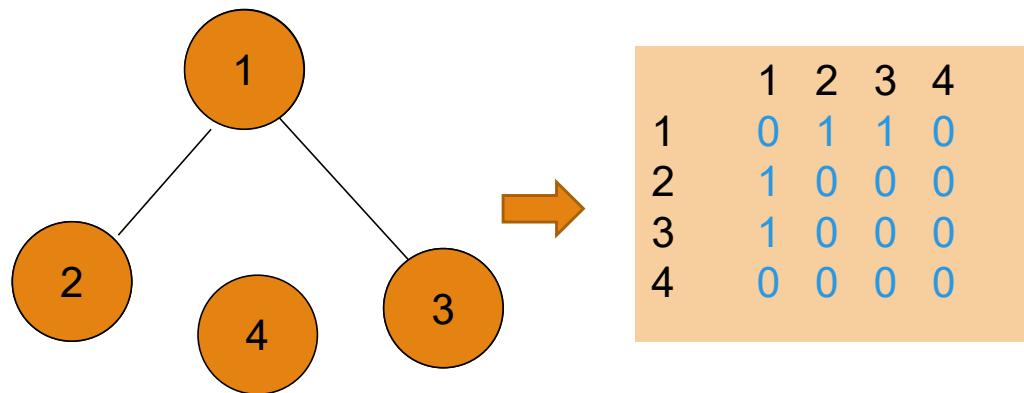
---

Using Adjacency matrix :

- When graph is dense use **Adjacency matrix**.
- Graphs are represent using a two-dimensional array (**Adjacency matrix**)
- Matrix of size  $|V| \times |V| \Rightarrow$ 
  - For each edge  $(u, v)$ , we set  $A[u][v]$  to true; otherwise the entry in the array is false.
  - If the edge has a weight associated with it, then we can set  $A[u][v]$  equal to the weight
    - use  $\infty$  (or perhaps 0) to represent nonexistent edges.
- Space requirement is  $(|V|^2)$ ,

# Examples

---



# C program to create a graph using adjacency matrix.

```
#include<stdio.h>
#define N 5
void edge(int arr[][N],int src, int dest)
{
    arr[src][dest] = 1;
}

int main()
{
    int adjMat[N][N];
    // init. adjMat to zero
    int i,j;
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            adjMat[i][j] = 0;
    // add edges to adjmat
```

Adjacency matrix Program  
www.shorturl.at/isKQV

```
        ,
// add edges to adjmat
edge(adjMat,0,4);
edge(adjMat,0,2);
edge(adjMat,0,3);
edge(adjMat,1,2);
edge(adjMat,1,4);
edge(adjMat,2,3);
edge(adjMat,3,1);
edge(adjMat,4,2);
// display adjmat
for(i = 0; i < N; i++)
{
    for(j = 0; j < N; j++)
    {
        printf("%d ", adjMat[i][j]);
    }
    printf("\n");
}
```

# Representing graphs

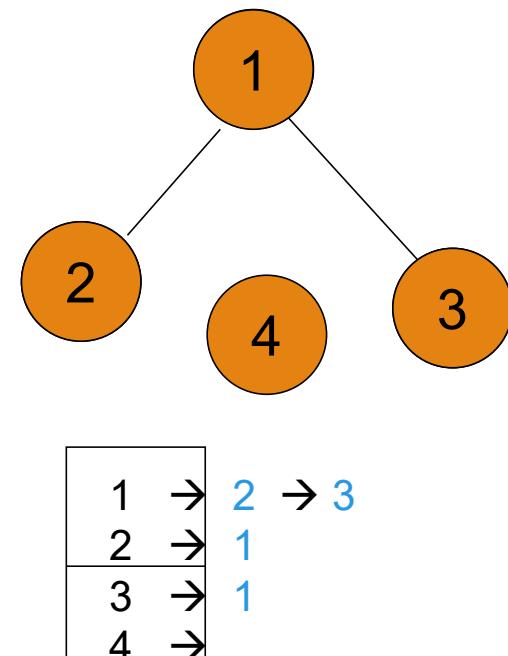
---

Using Adjacency List:

- An array  $\text{Adj}[ ]$  of size  $|V|$
- Each cell holds a list for associated vertex
- $\text{Adj}[u]$  is list of all vertices **adjacent** to  $u$ 
  - List does not have to be sorted
  - Undirected graphs: Each edge is represented twice

Storage Complexity:

- $O(|V| + |E|)$
- In undirected graph:  $O(|V|+2*|E|) = O(|V|+|E|)$



## Representing graphs using Adjacency List:

---

### Structure :

1) To represent a graph (graph is an array of adjacency lists.

```
struct Graph
{
    int V;
    struct AdjList* arr;
};
```

2) To represent an adjacency list

```
struct AdjList
{
    struct AdjListNode
    *head;
};
```

3) A structure to represent an adjacency list node

```
struct AdjListNode
{
    int dest;
    struct AdjListNode* next;
};
```

# Graph Traversals

---

- There are two standard graph traversal(search) techniques:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- In both DFS and BFS
  - the nodes of the undirected graph are visited in a systematic manner so that every node is visited exactly one.
  - When a node  $x$  is visited, it is labeled as visited, and it is added to the tree
  - If the traversal got to node  $x$  from node  $y$ ,  $y$  is viewed as the parent of  $x$ , and  $x$  a child of  $y$

# Depth-First Search

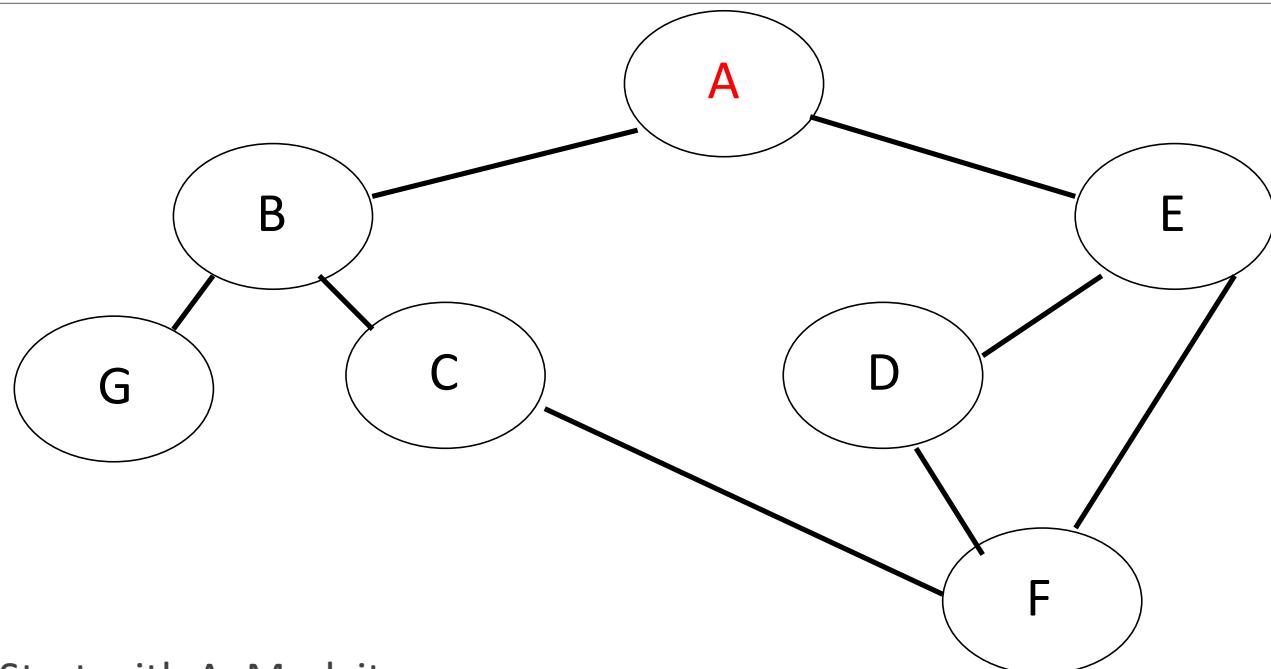
---

DFS follows the following rules:

1. Select an unvisited node  $x$ , visit it, and treat as the **current node**
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, **backtrack** to the its parent, and make that parent the new current node;
4. Repeat steps 3 and 4 until no more nodes can be visited.
5. If there are still unvisited nodes, repeat from step 1.

## Illustration of DFS

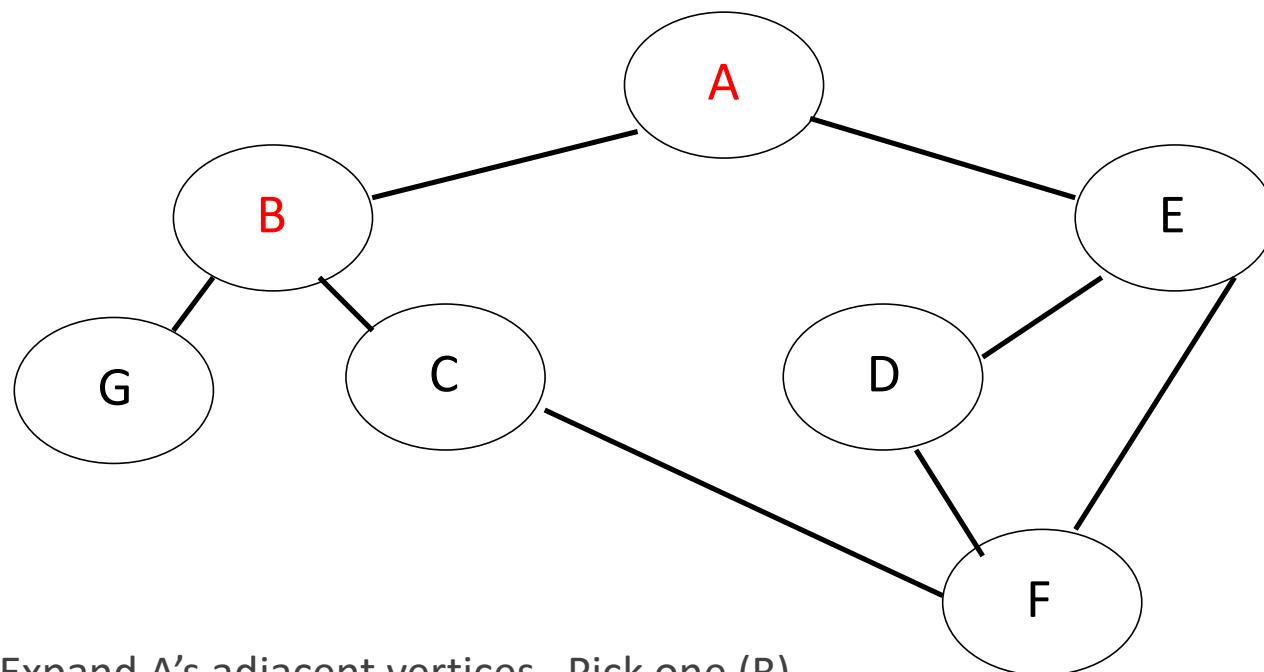
Current vertex: A



Start with A. Mark it.

Current: B

---

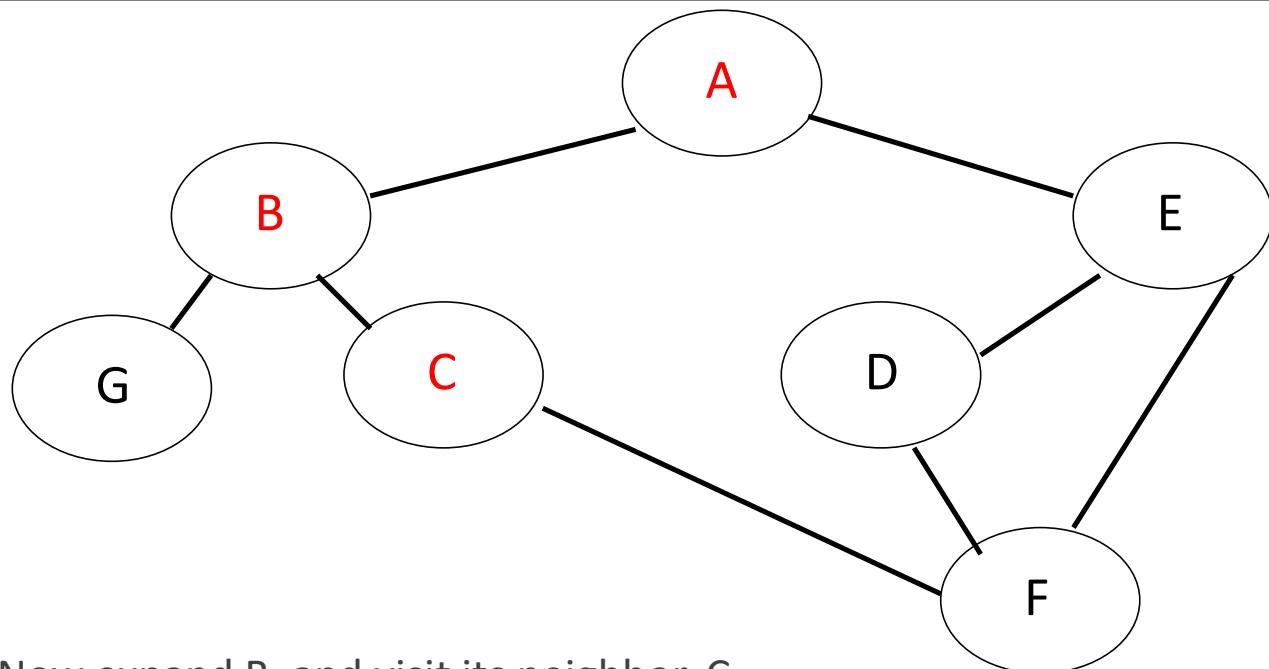


Expand A's adjacent vertices. Pick one (B).

Mark it and re-visit.

Current: C

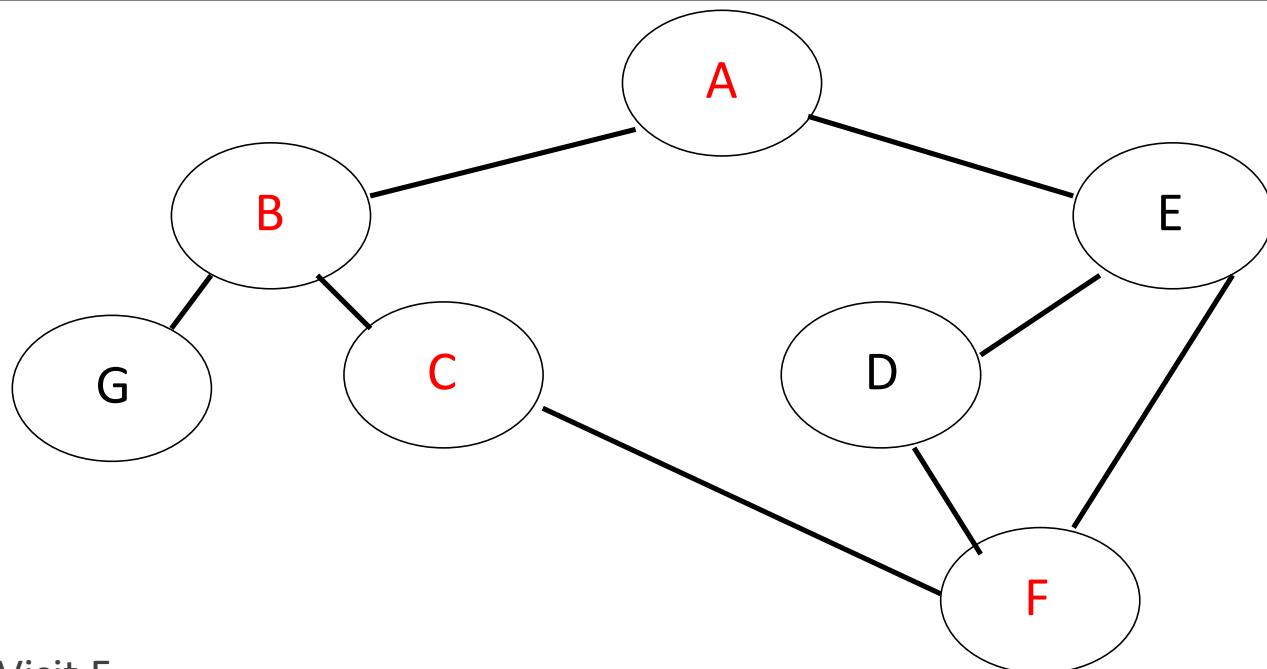
---



Now expand B, and visit its neighbor, C.

Current: F

---

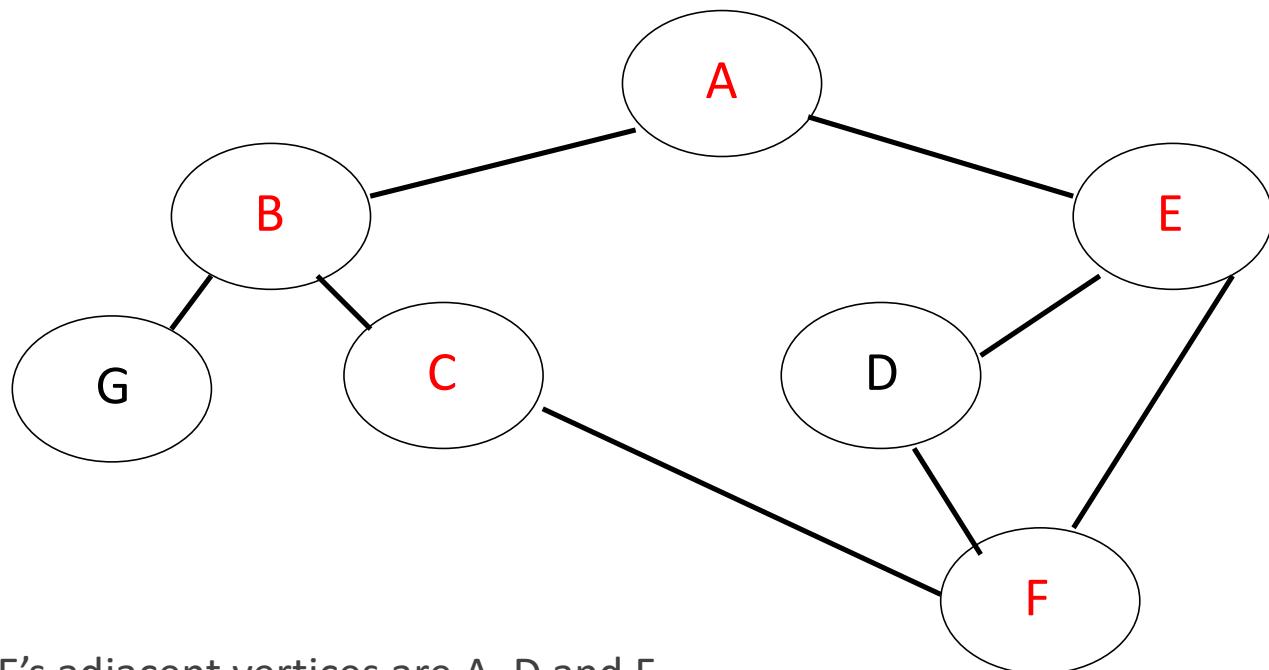


Visit F.

Pick one of its neighbors, E.

Current: E

---

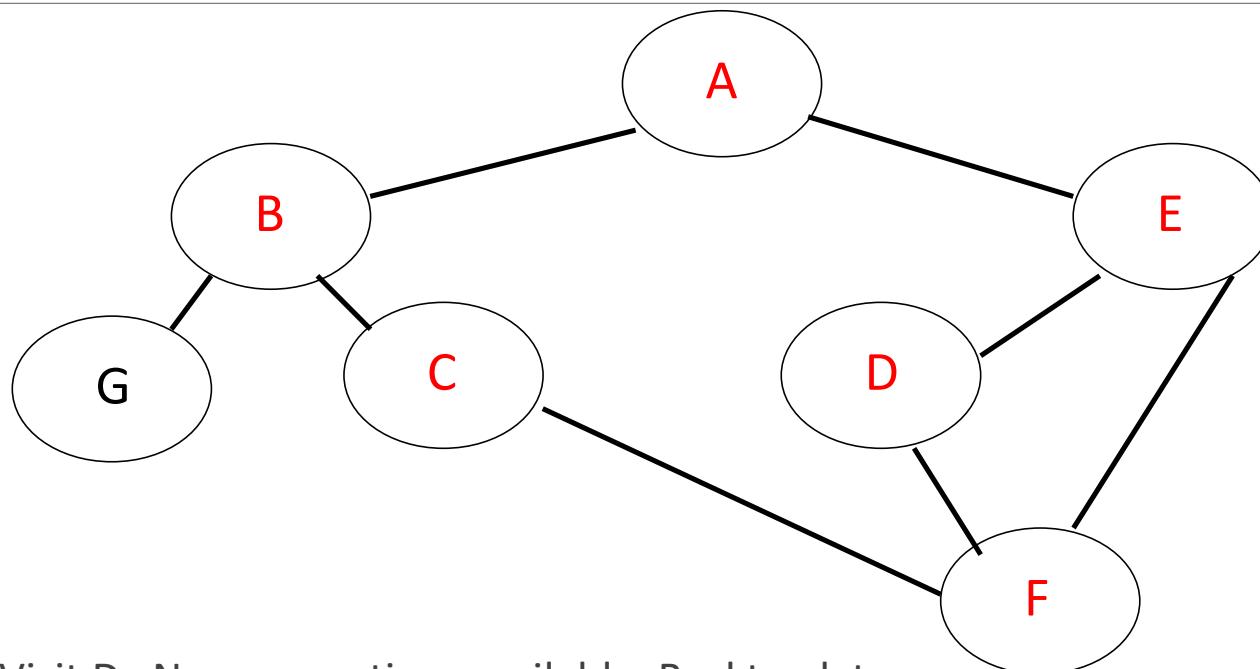


E's adjacent vertices are A, D and F.

A and F are marked, so pick D.

Current: D

---

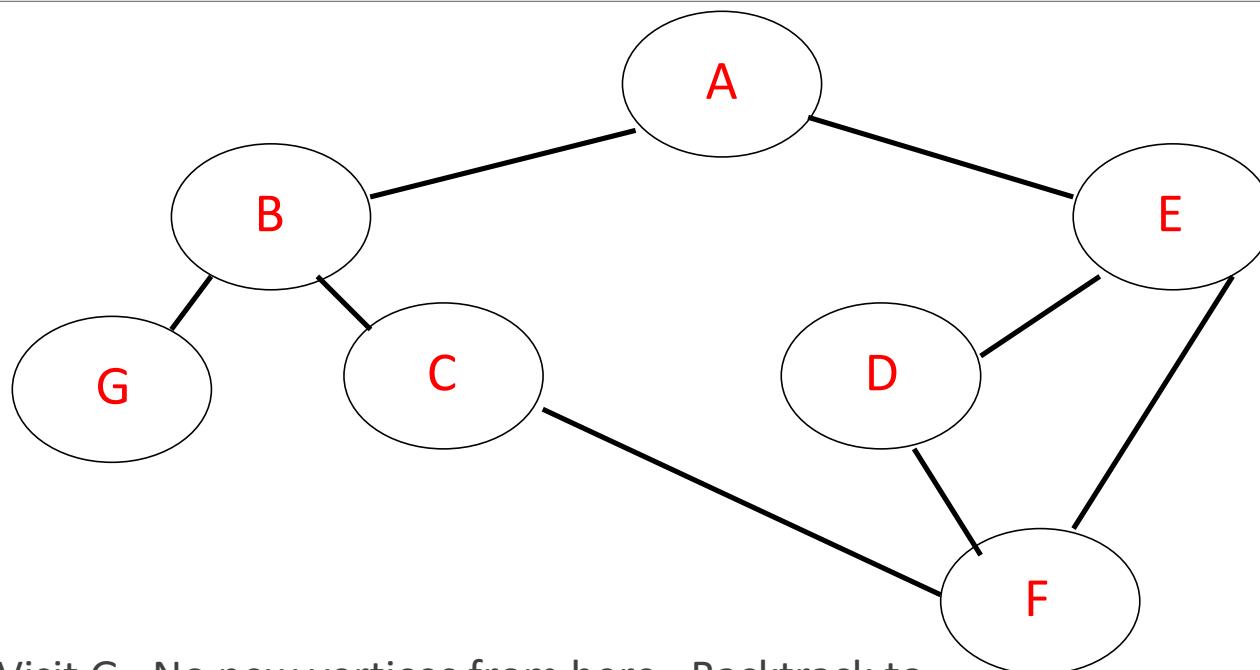


Visit D. No new vertices available. Backtrack to

E. Backtrack to F. Backtrack to C. Backtrack to B

Current: G

---

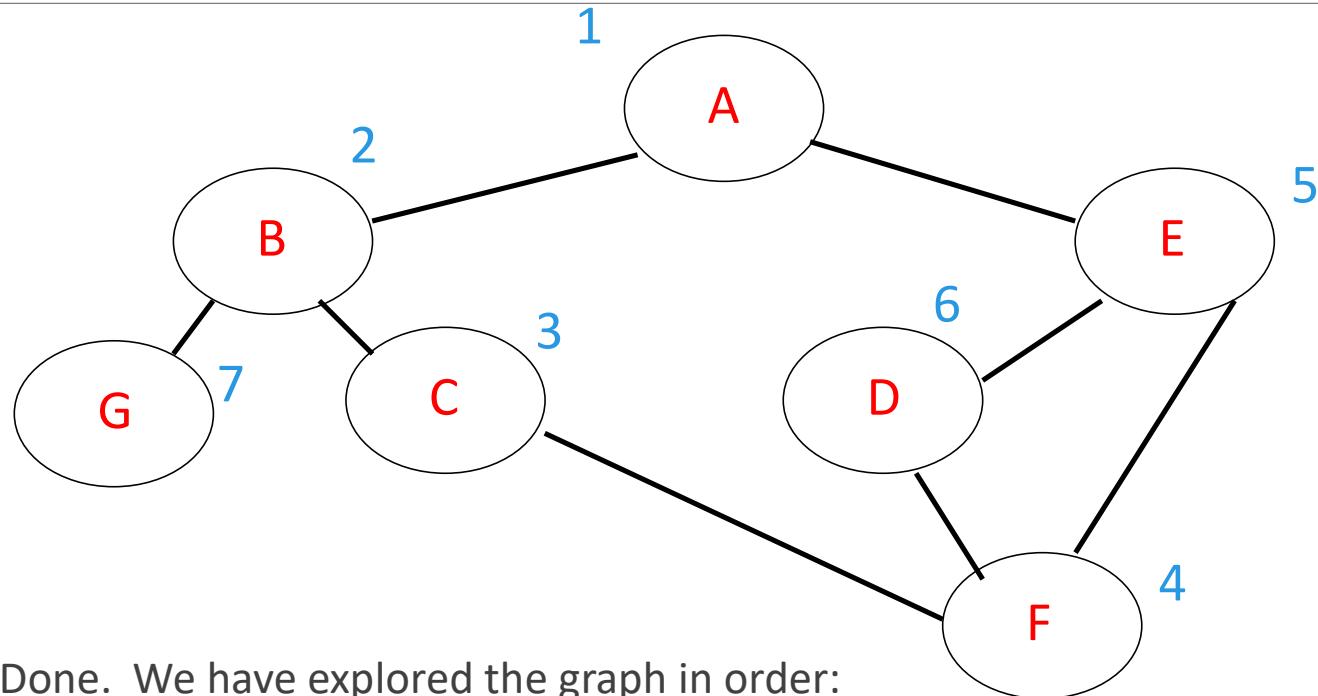


Visit G. No new vertices from here. Backtrack to

B. Backtrack to A. E already marked so no new.

**Current:**

---

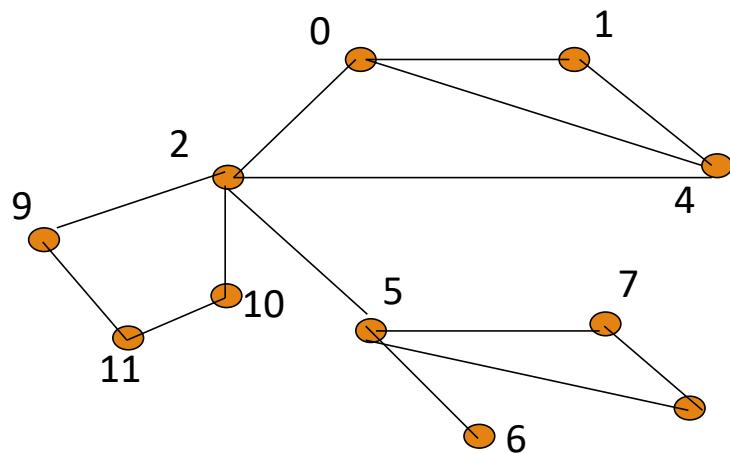


Done. We have explored the graph in order:

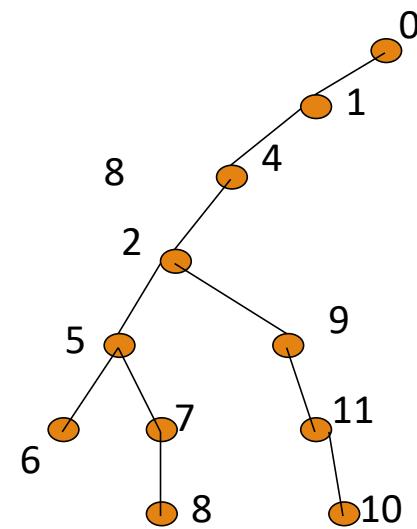
A B C F E D G

# Illustration of DFS

---



Graph G



DFS Tree

# Observations

- the last node visited is the first node from which to proceed.
- Also, the backtracking proceeds on the basis of "last visited, first to backtrack too".

This suggests that a stack is the proper data structure to remember the current node and how to backtrack.

```
DFS-A(G,s)
    for all v in V[G] do
        visited[v] := false
    end for
    S := EmptyStack
    Push(S,s)
    while not Empty(S) do
        u := Pop(S)
        if not visited[u] then
            visited[u] := true
            for all w in Adj[u] do
                if not visited[w] then
                    Push(S,w)
                end if
            end for
        end if
    end while
```

# Recursive DFS

**Data:** G: The graph stored in an adjacency list

root: The starting node

**Result:** Prints all nodes inside the graph in the *DFS* order  
 $visited \leftarrow \{false\};$   
 $DFS(root);$

**Function**  $DFS(u)$ :

```
if  $visited[u] = true$  then
    | return;
  end
  print( $u$ );
   $visited[u] \leftarrow true$ ;
  for  $v \in G[u].neighbors()$  do
    |  $DFS(v)$ ;
  end
end
```

---

DFS Complexity:  $O(|V| + |E|)$

- All vertices visited once, then marked
  - For each vertex on queue, we examine all edges
- In other words, we traverse all edges once

# Breadth-First Search

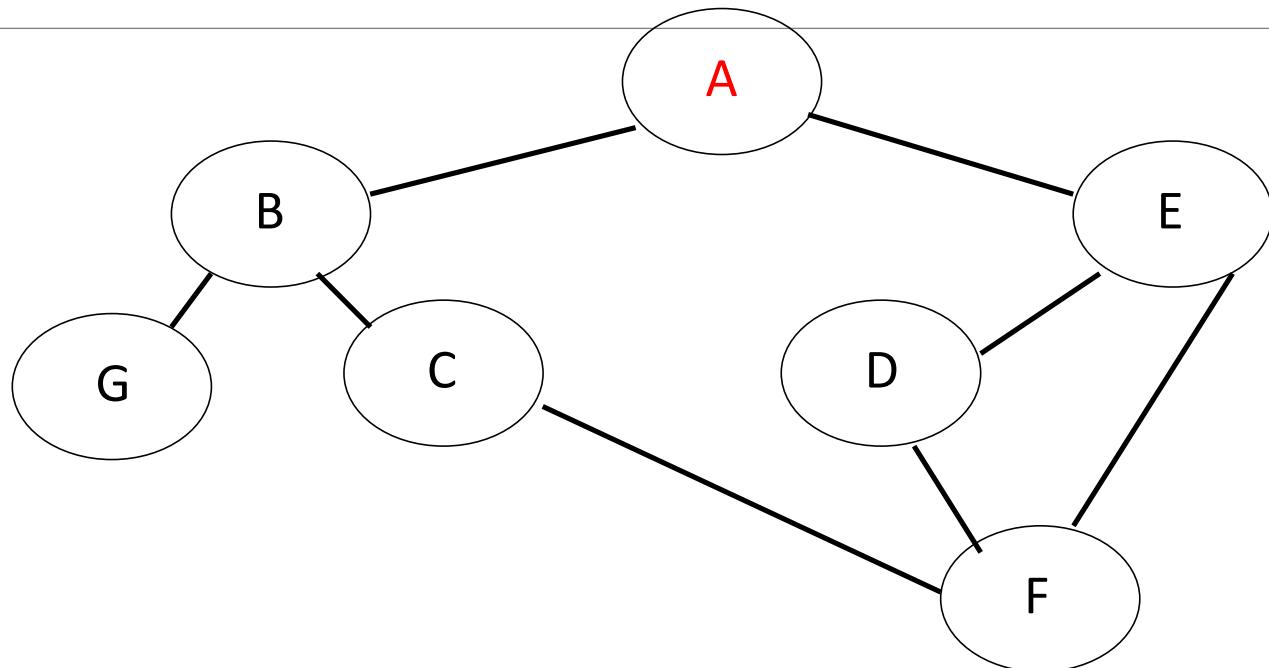
---

BFS follows the following rules:

1. Select an unvisited node  $x$ , visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
2. From each node  $z$  in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of  $z$ . The newly visited nodes from this level form a new level that becomes the next current level.
3. Repeat step 2 until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from Step 1.

## Queue: A

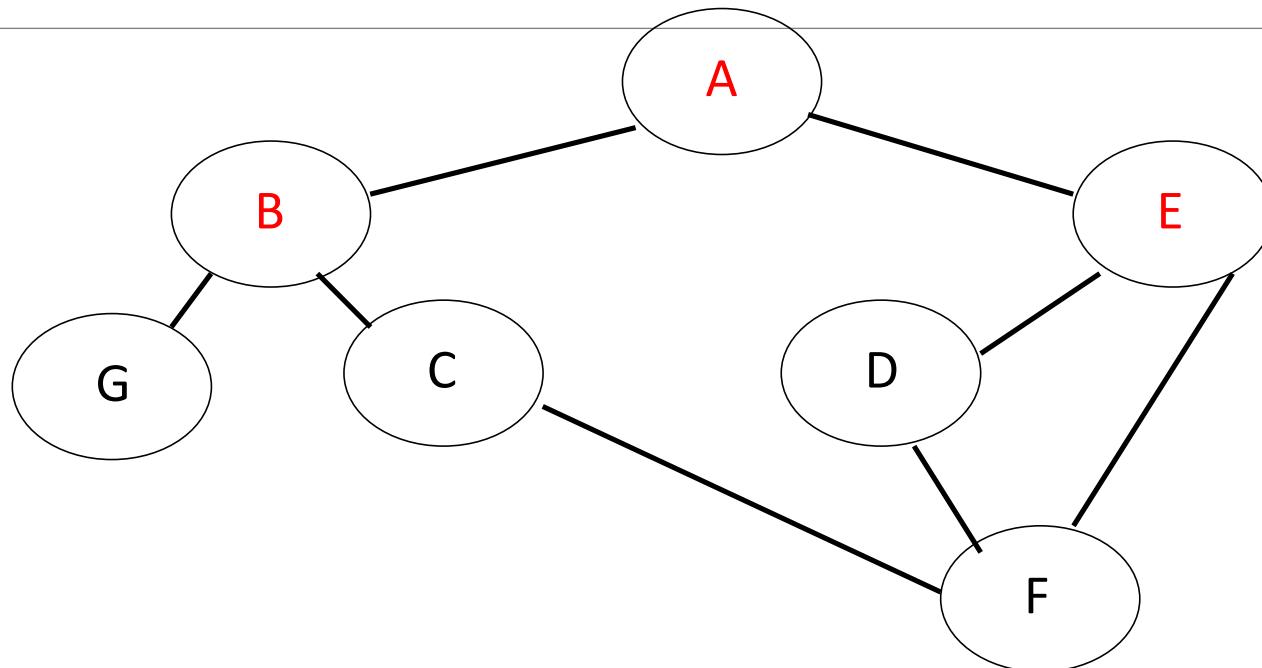
---



Start with A. Mark it.

Queue: A B E

---

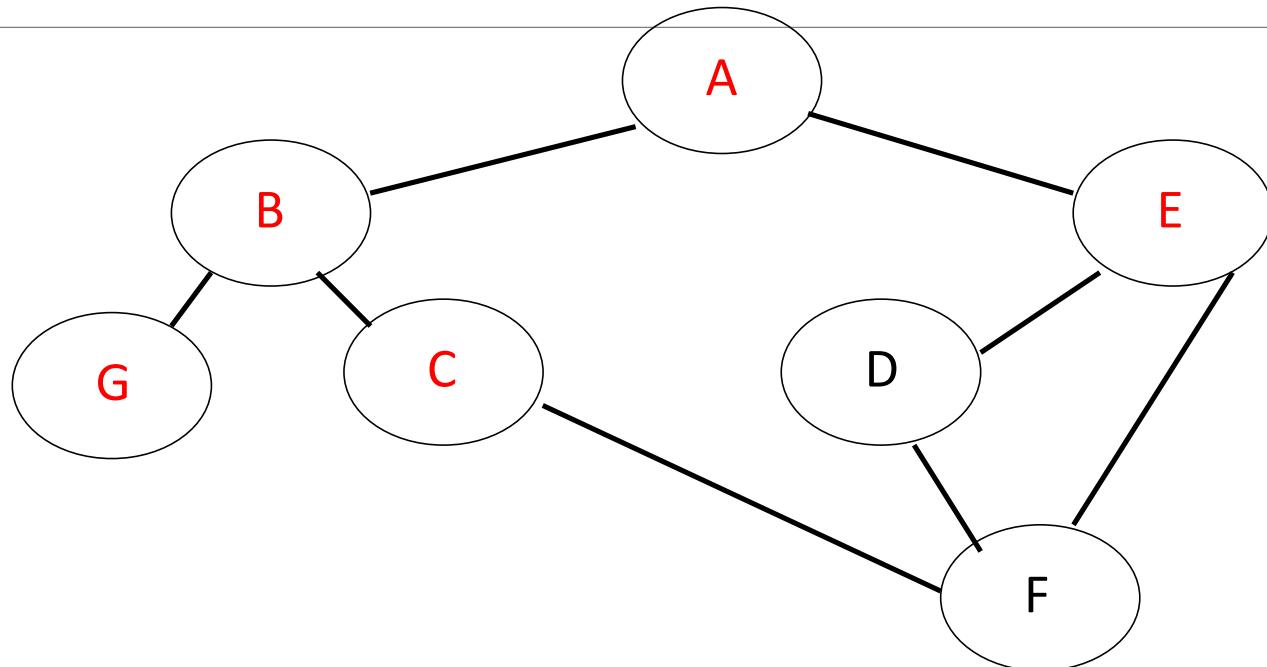


Expand A's adjacent vertices.

Mark them and put them in queue.

Queue: A B E C G

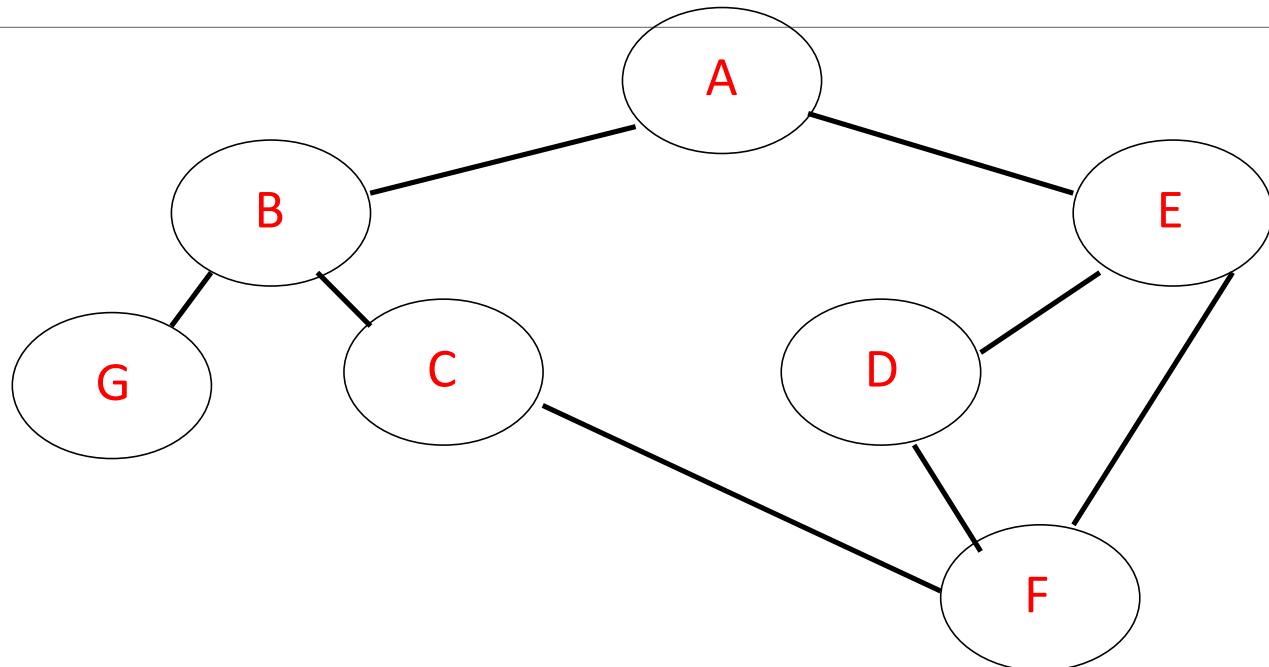
---



Now take B off queue, and queue its neighbors.

Queue: A B E C G D F

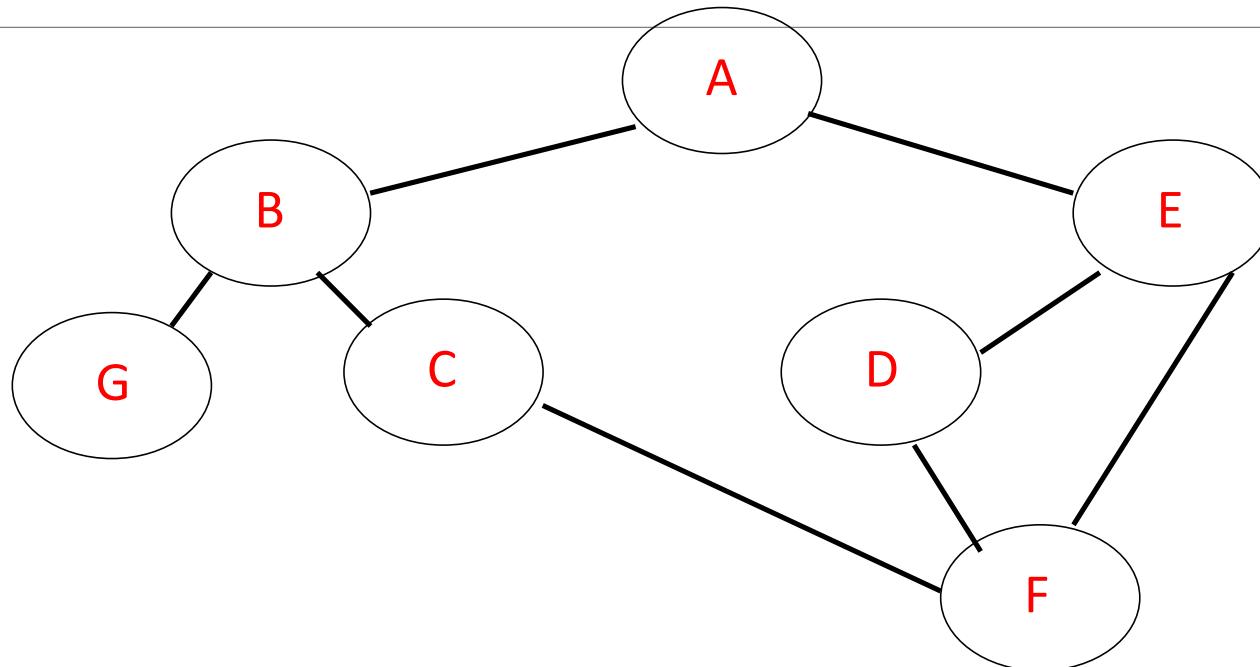
---



Do same with E.

Queue: A B E C G D F

---

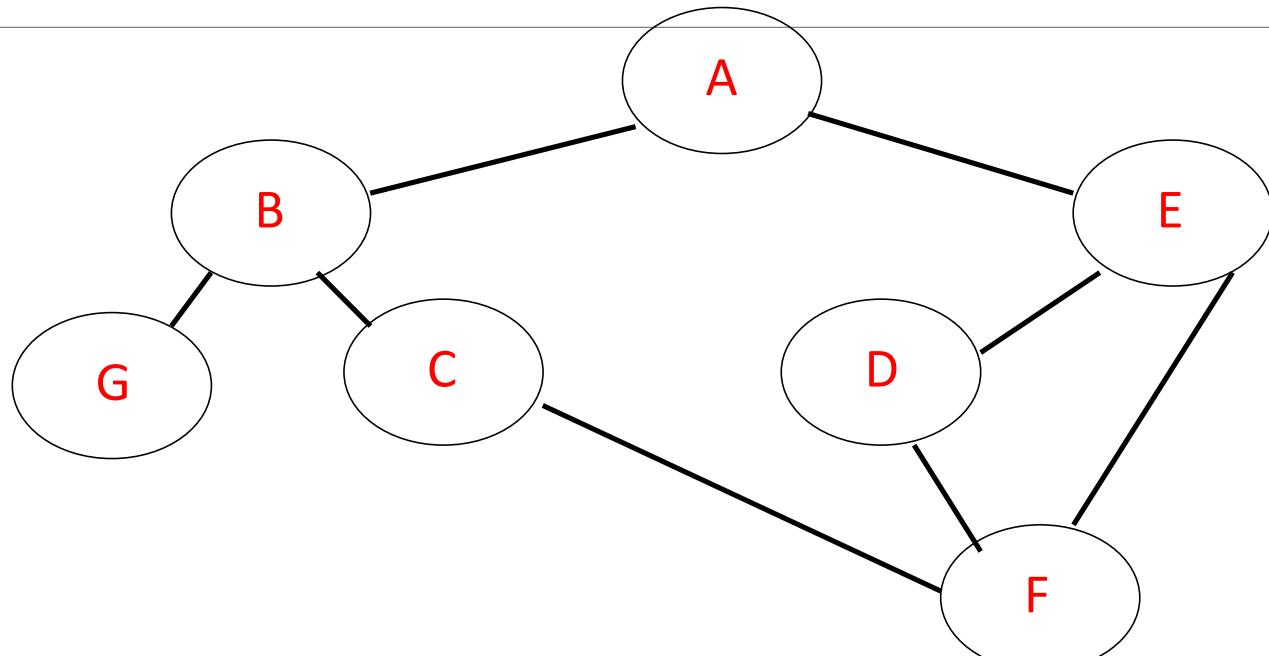


Visit C.

Its neighbor F is already marked, so not queued.

Queue: A B E C G D F

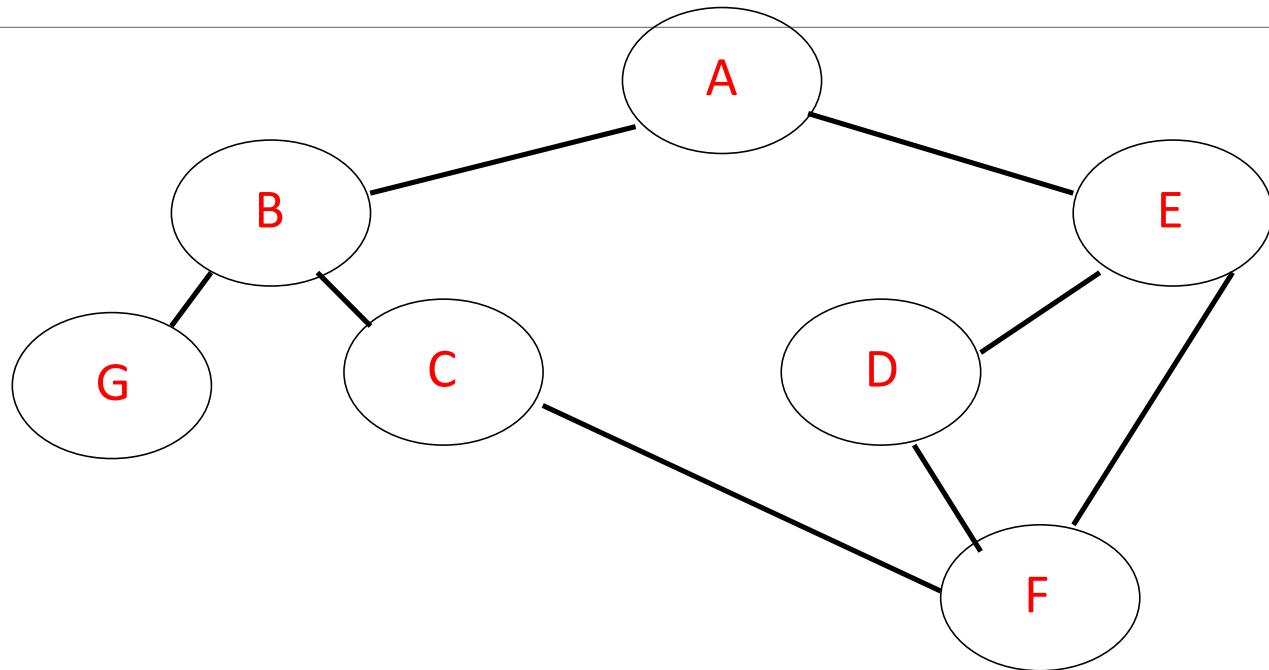
---



Visit G.

Queue: A B E C G D F

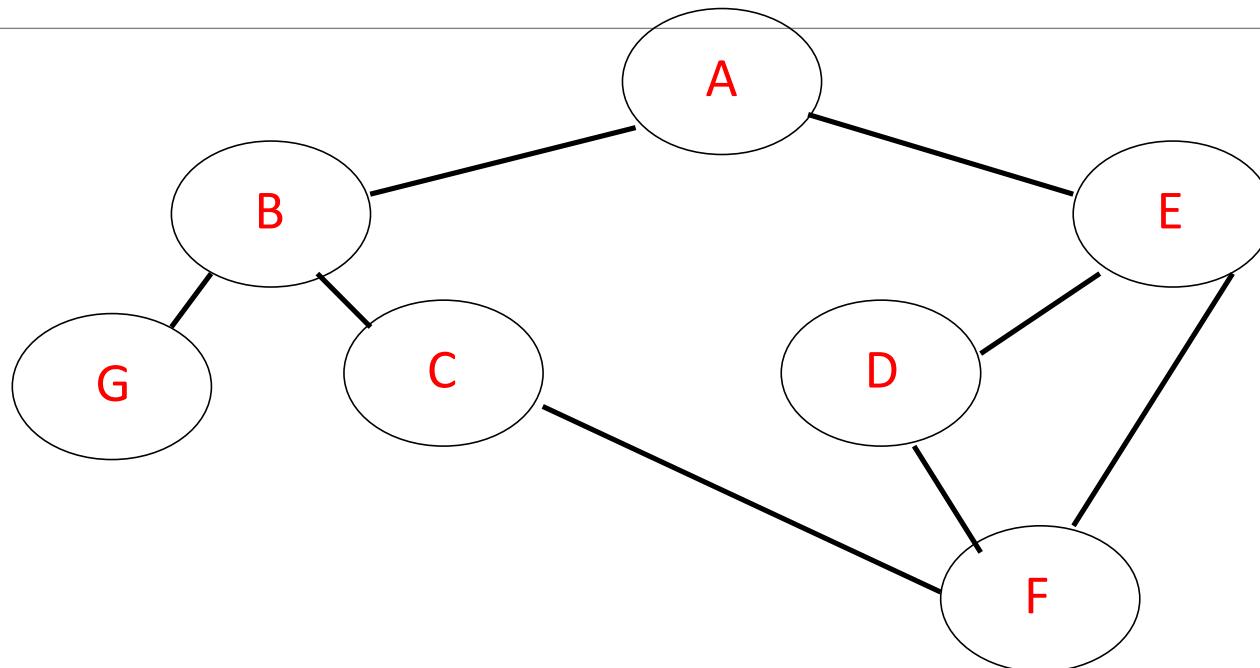
---



Visit D. F, E marked so not queued.

Queue: A B E C G D F

---

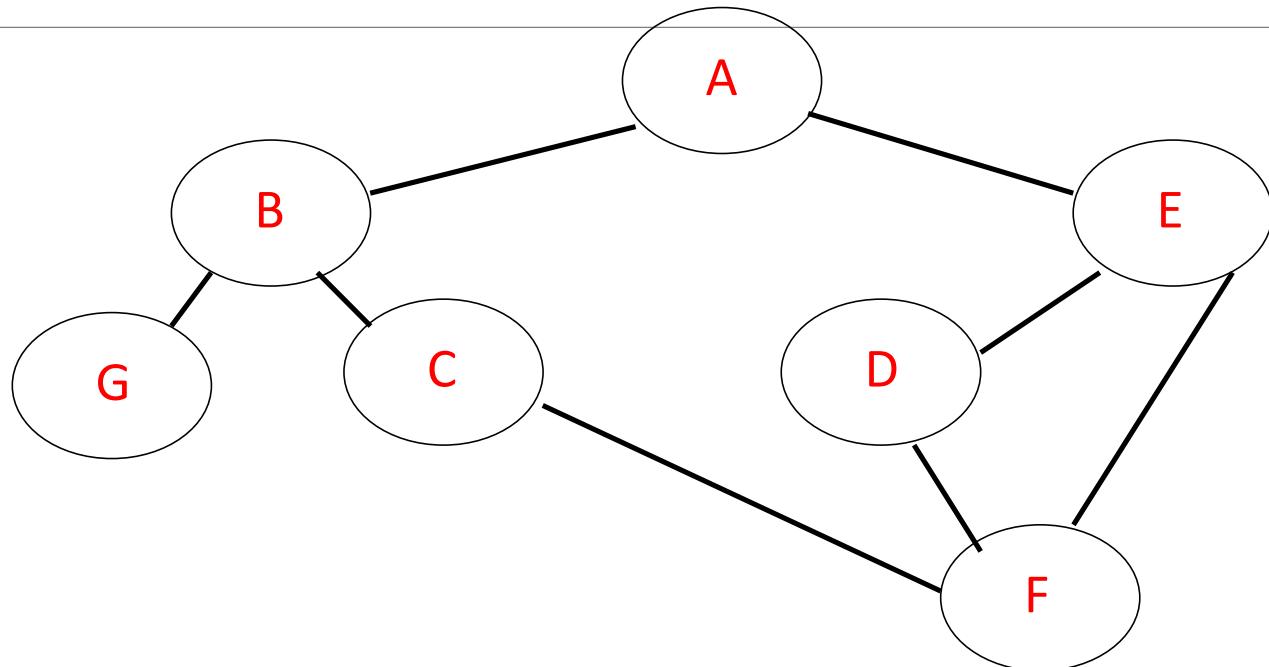


Visit F.

E, D, C marked, so not queued again.

**Queue:** A B E C G D F

---

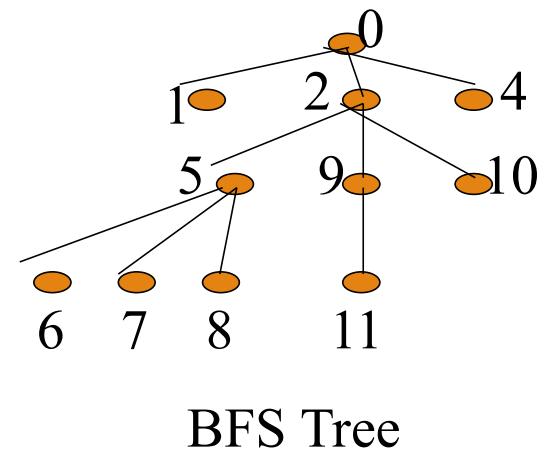
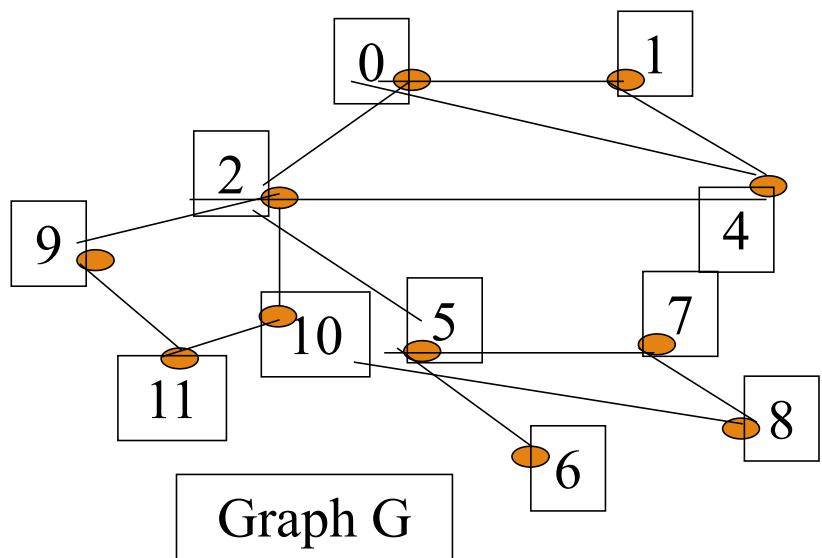


Done. We have explored the graph in order:

A B E C G D F.

# Illustration of BFS

---



---

## Observations:

- The first node visited in each level is the first node from which to proceed to visit new nodes.
- This suggests that a queue is the proper data structure to remember the order of the steps.

---

## BFS(graph g, vertex s)

```
1.  unmark all vertices in G
2.  q ← new queue
3.  mark s
4.  enqueue(q, s)
5.  while (not empty(q))
6.      curr ← dequeue(q)
7.      visit curr // e.g., print its
       data
8.      for each edge <curr, v>
9.          if v is unmarked
10.             mark v
11.             enqueue(q, v)
```

---

Complexity:  $O(|V| + |E|)$

- All vertices put on queue exactly once
- For each vertex on queue, we expand its edges
- In other words, we traverse all edges once

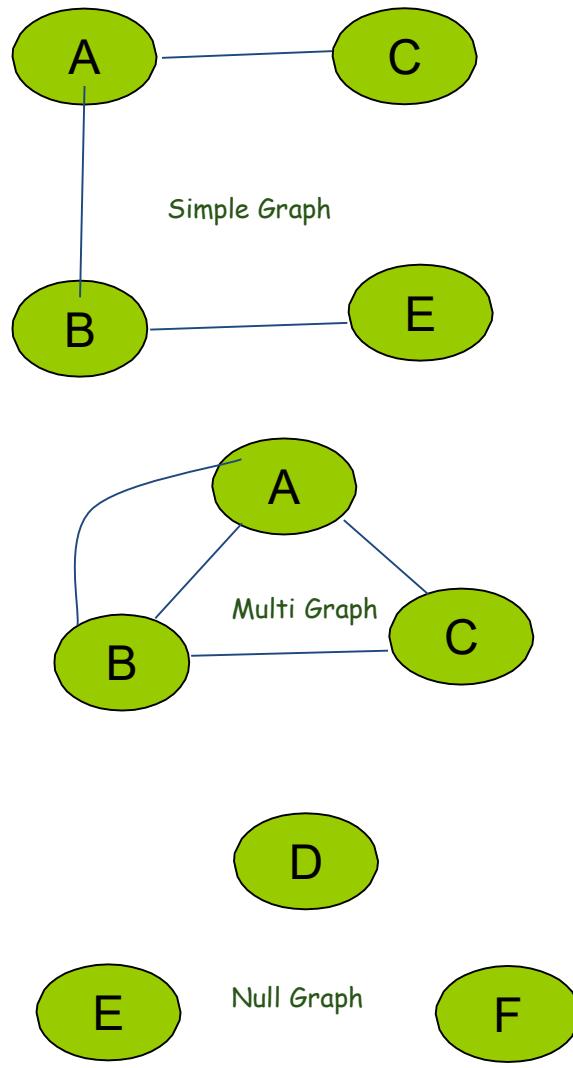


# Minimum Spanning Tree

Presented by:  
Dr. Rakesh Kumar Sanodiya



## GRAPH



Graph can be classified into three types:

- Simple Graph
- Multi Graph
- Null Graph

**Simple Graph:** Between any two vertices **at most one edge**

**Multi Graph:** Between any two nodes more than **one edge** or **loop**

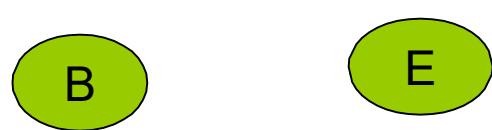
**Null Graph:** No edge



## GRAPH

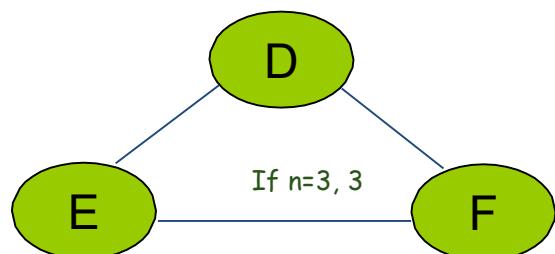


Zero Edge



Simple Graph can contain minimum number of edges **zero**

Simple Graph with **n nodes** contains less than or equal to  $n(n-1)/2$  edges.



If number of nodes is **3**, then maximum number of edges in simple graph is **3**

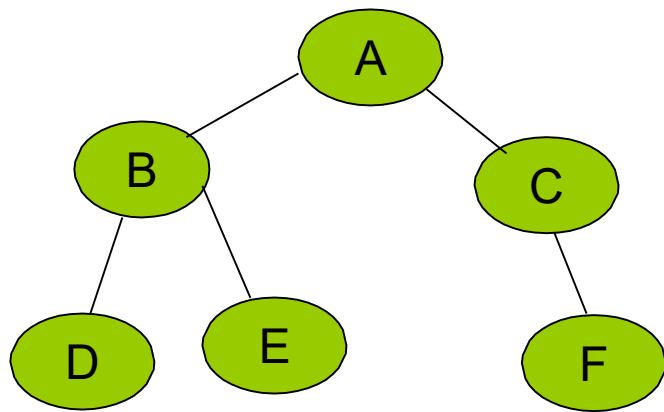
Worst Case number of edges  
 $E=O(n^2)$ , if  $n=V$

$E=O(V^2)$

Number of vertices  $V=O(\log(E))$



# TREE

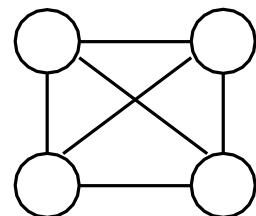


Connected acyclic graph

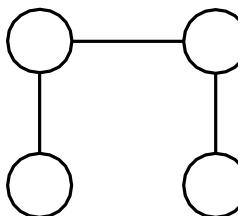
Tree with **n** nodes contains  
exactly **n-1** edges.

# SPANNING TREE...

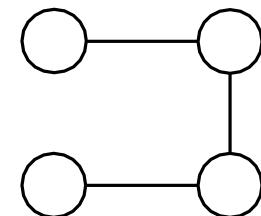
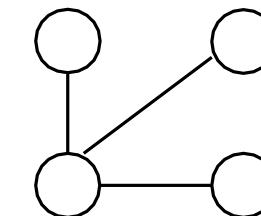
- Minimum number of edges required to connect all the nodes is called Spanning Tree
- Minimum number of edges required to connect  $n$  nodes is  $n-1$
- Suppose you have a connected undirected graph
  - Connected: every node is reachable from every other node
  - Undirected: edges do not have an associated direction
- a spanning tree of the graph is a connected subgraph in which there are no cycles
- For a complete graph ( $K_n$ ), all possible the number of spanning tree=  $n^{n-2}$



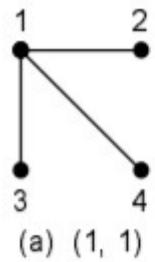
A connected,  
undirected graph



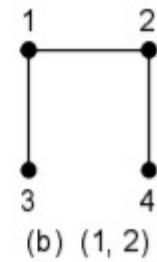
Four of the spanning trees of the graph



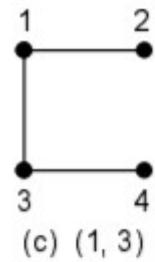
## Contd...



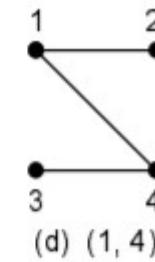
(a) (1, 1)



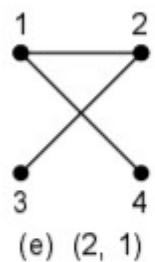
(b) (1, 2)



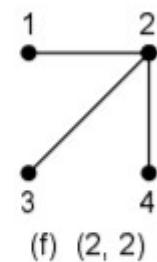
(c) (1, 3)



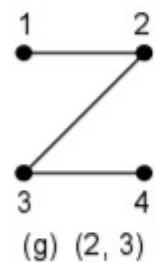
(d) (1, 4)



(e) (2, 1)



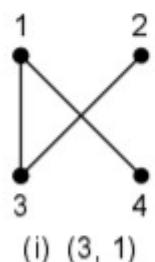
(f) (2, 2)



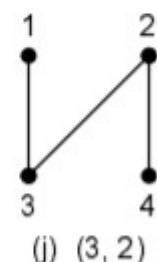
(g) (2, 3)



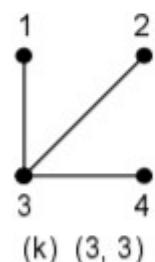
(h) (2, 4)



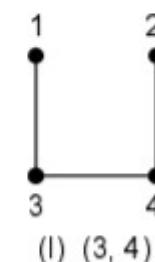
(i) (3, 1)



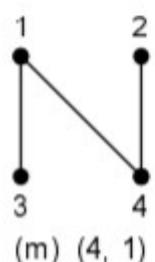
(j) (3, 2)



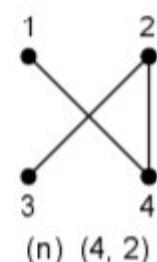
(k) (3, 3)



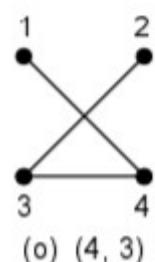
(I) (3, 4)



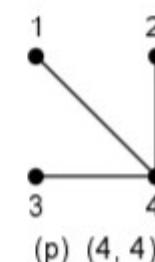
(m) (4, 1)



(n) (4, 2)



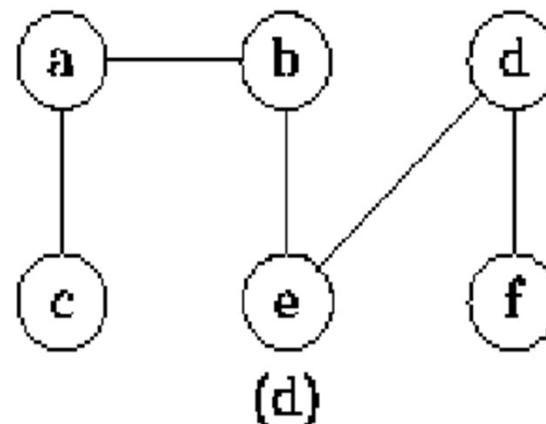
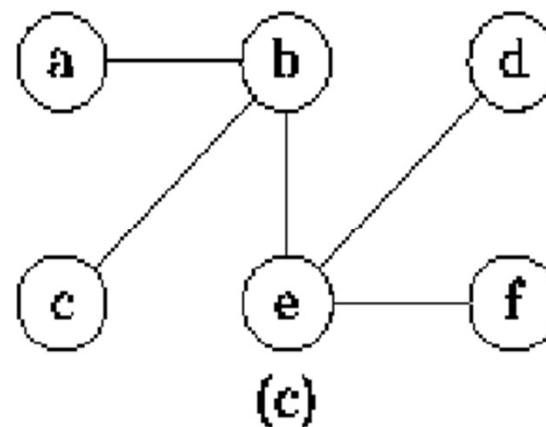
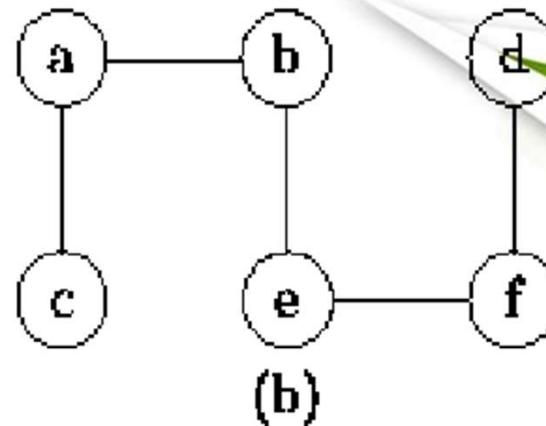
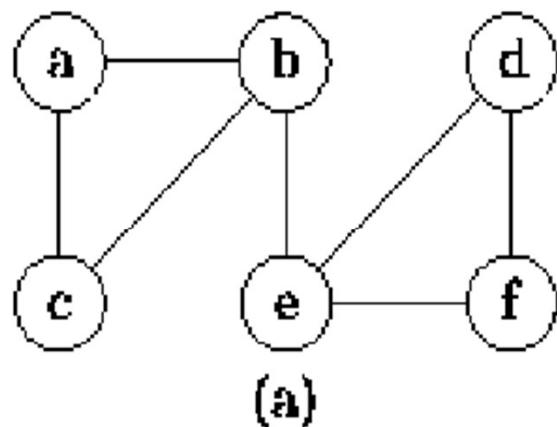
(o) (4, 3)



(p) (4, 4)

# EXAMPLE..

$G_{13}$



# Minimizing costs

Suppose you want to **supply** a set of houses (say, in a new subdivision) with:

- electric power
- water
- sewage lines
- telephone lines

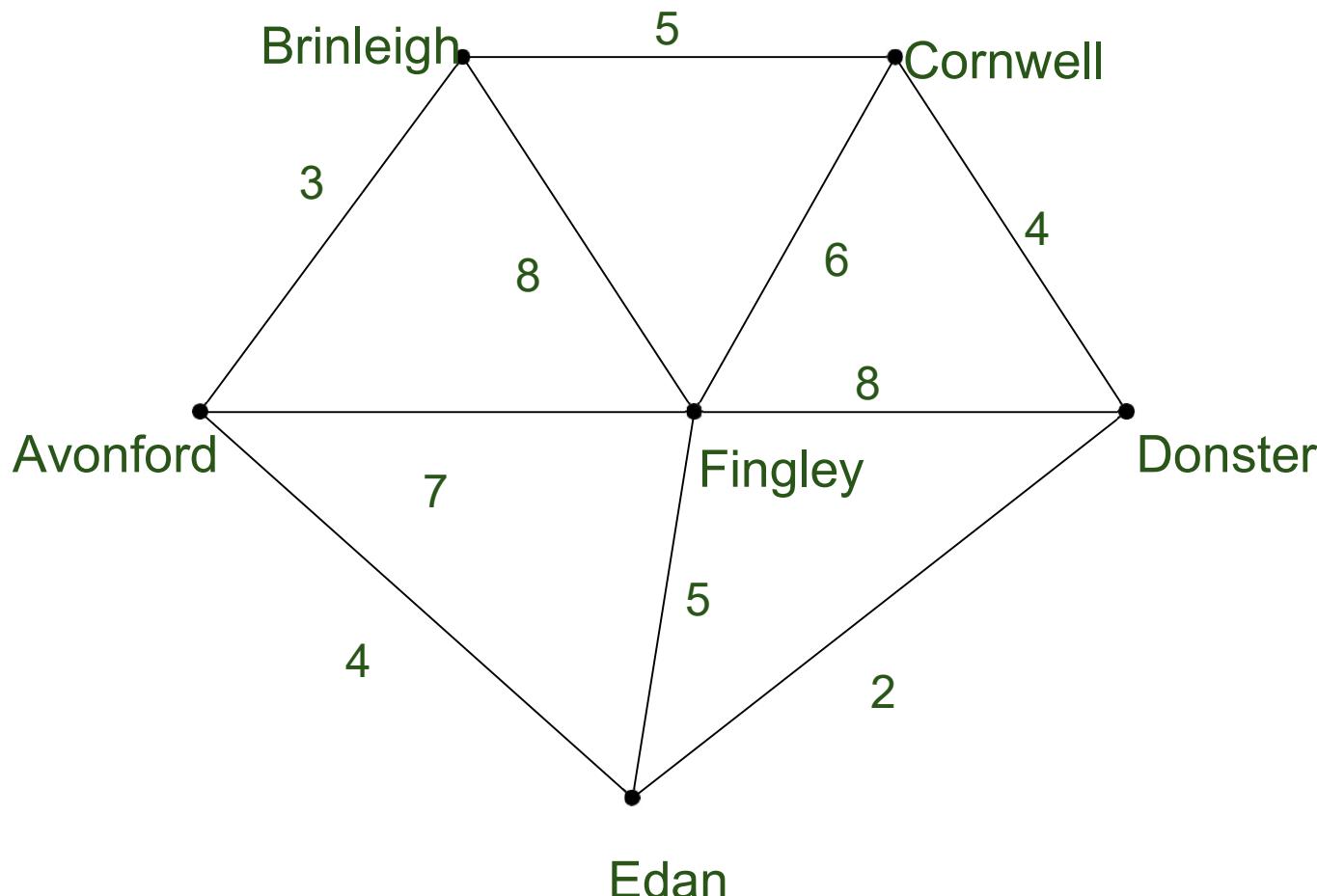
To **keep costs down**, you could connect these houses with a **spanning tree** (of, for example, power lines)

However, the houses are **not all equal distances apart**

To **reduce costs** even further, you could connect the houses with a **minimum-cost spanning tree**

# Example

A cable company want to connect **five villages** to their network which currently extends to the market town of Avonford. What is the minimum length of cable needed?



# MINIMUM SPANNING TREE

Let  $G = (N, A)$  be a connected, undirected graph where  $N$  is the set of nodes and  $A$  is the set of edges.

Each edge has a given nonnegative length.

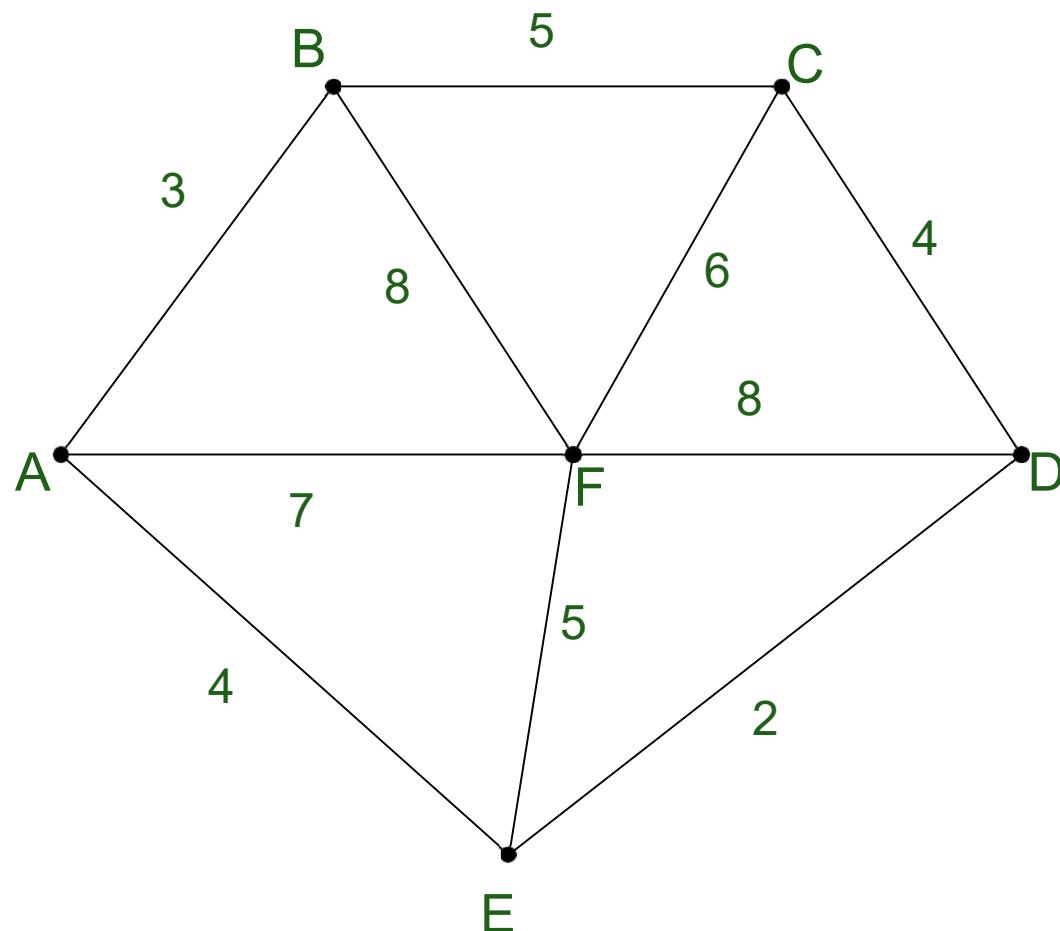
The problem is to find a subset  $T$  of the edges of  $G$  such that all the nodes remain connected when only the edges in  $T$  are used, and the sum of the lengths of the edges in  $T$  is as small as possible.

Since  $G$  is connected, at least one solution must exist.

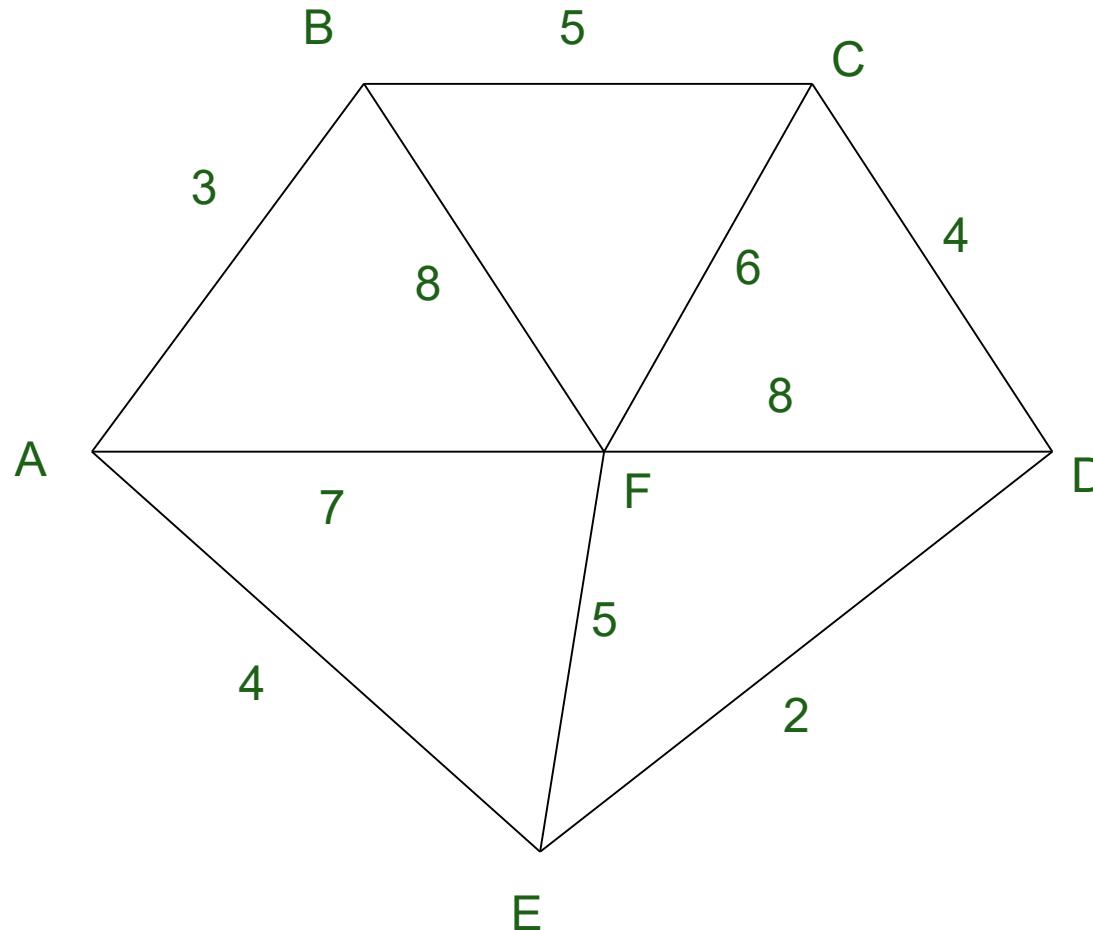
# Finding Spanning Trees

- There are two basic algorithms for finding minimum-cost spanning trees, and both are **greedy algorithms**
- **Kruskal's algorithm:**
  - Developed in 1957 by Joseph Kruskal
- **Prim's algorithm**
  - Developed by Robert C. Prim

We model the situation as a network, then the problem is to find the **minimum connector** for the network



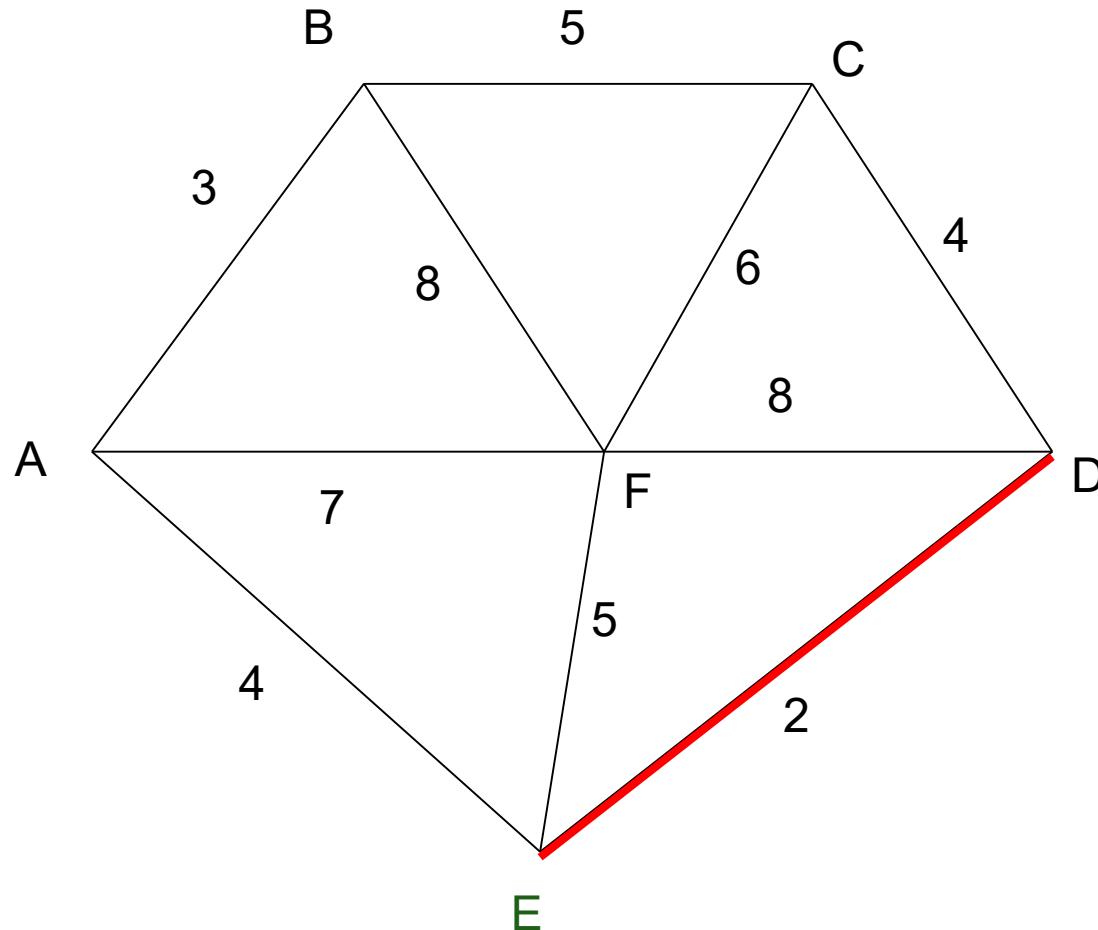
# Kruskal's Algorithm



List the edges in order of size:

ED	2
AB	3
AE	4
CD	4
BC	5
EF	5
CF	6
AF	7
BF	8
CF	8

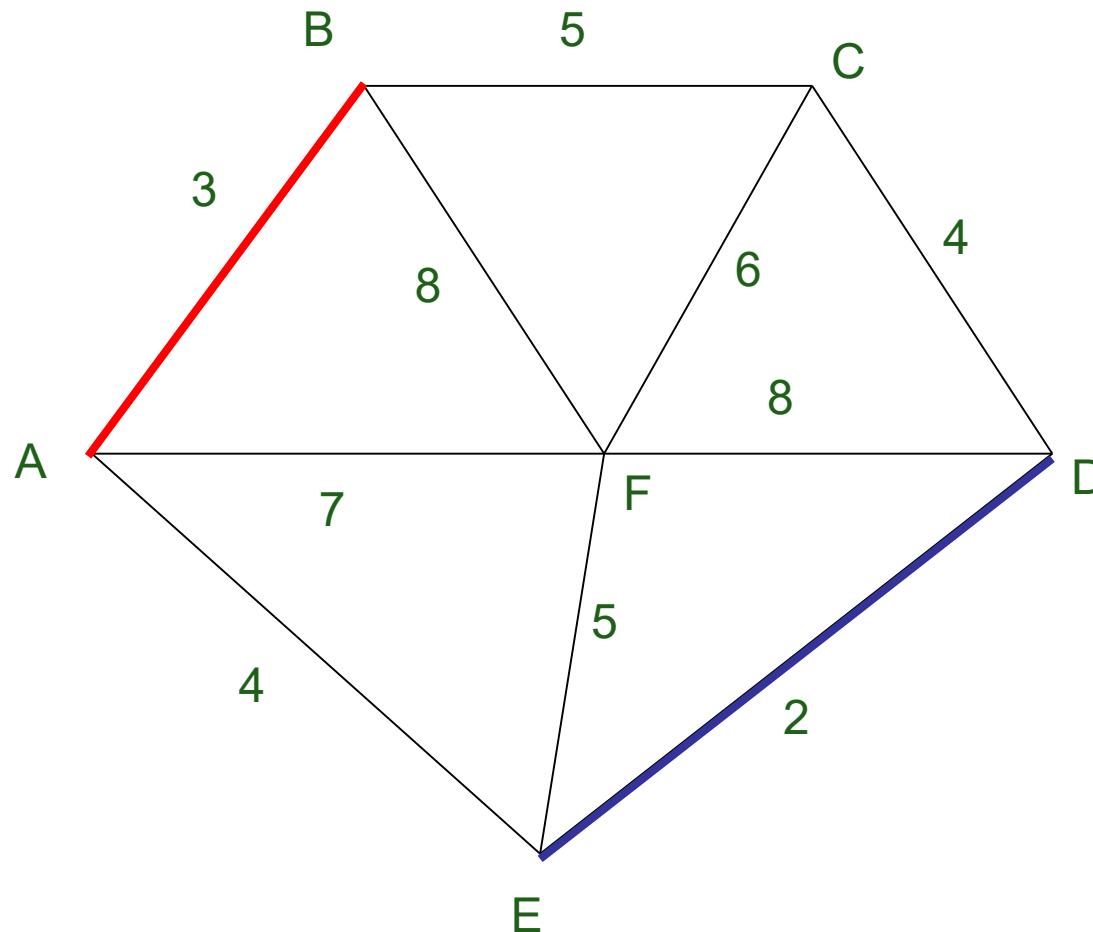
# Kruskal's Algorithm



Select the shortest edge in the network

**ED    2**

# Kruskal's Algorithm



Select the next  
shortest  
edge which does not  
create a cycle

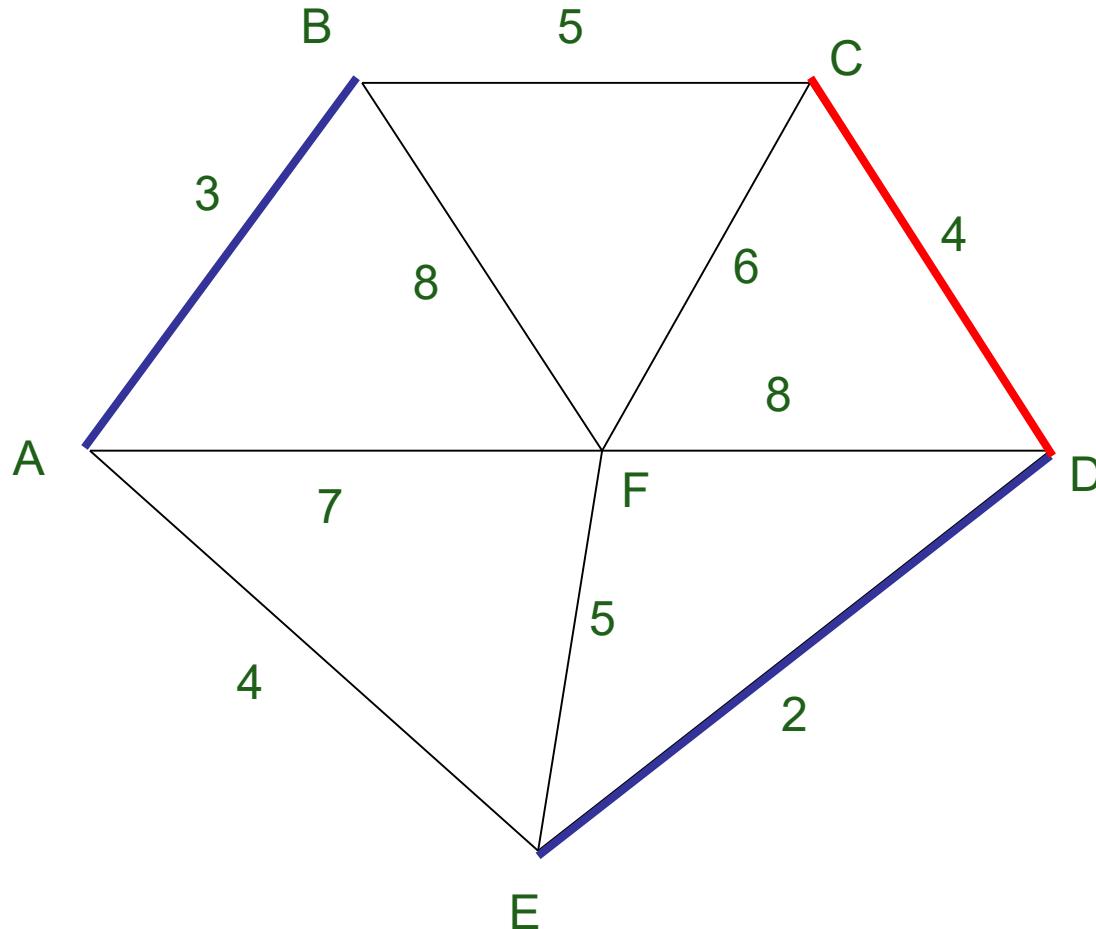
ED

2

AB

3

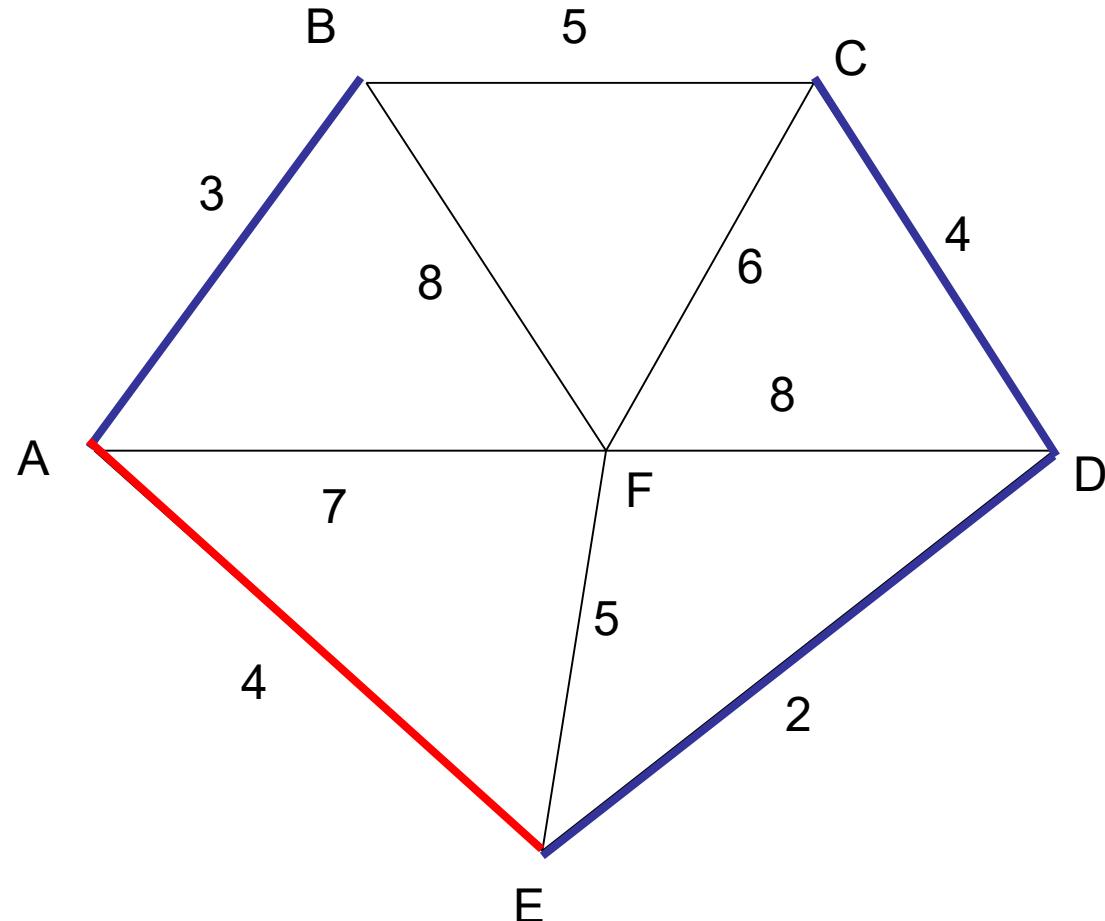
# Kruskal's Algorithm



Select the next  
shortest  
edge which does not  
create a cycle

ED	2
AB	3
CD	4 (or AE)
	4)

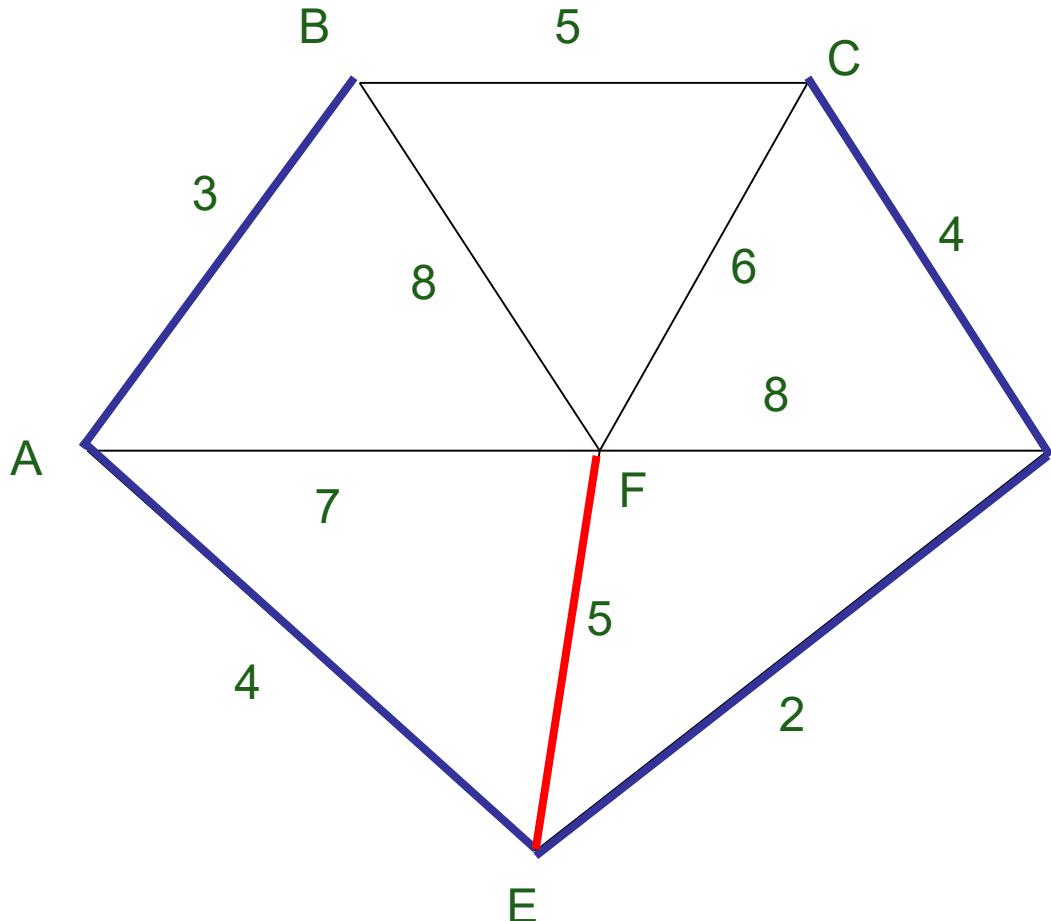
# Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED	2
AB	3
CD	4
AE	4

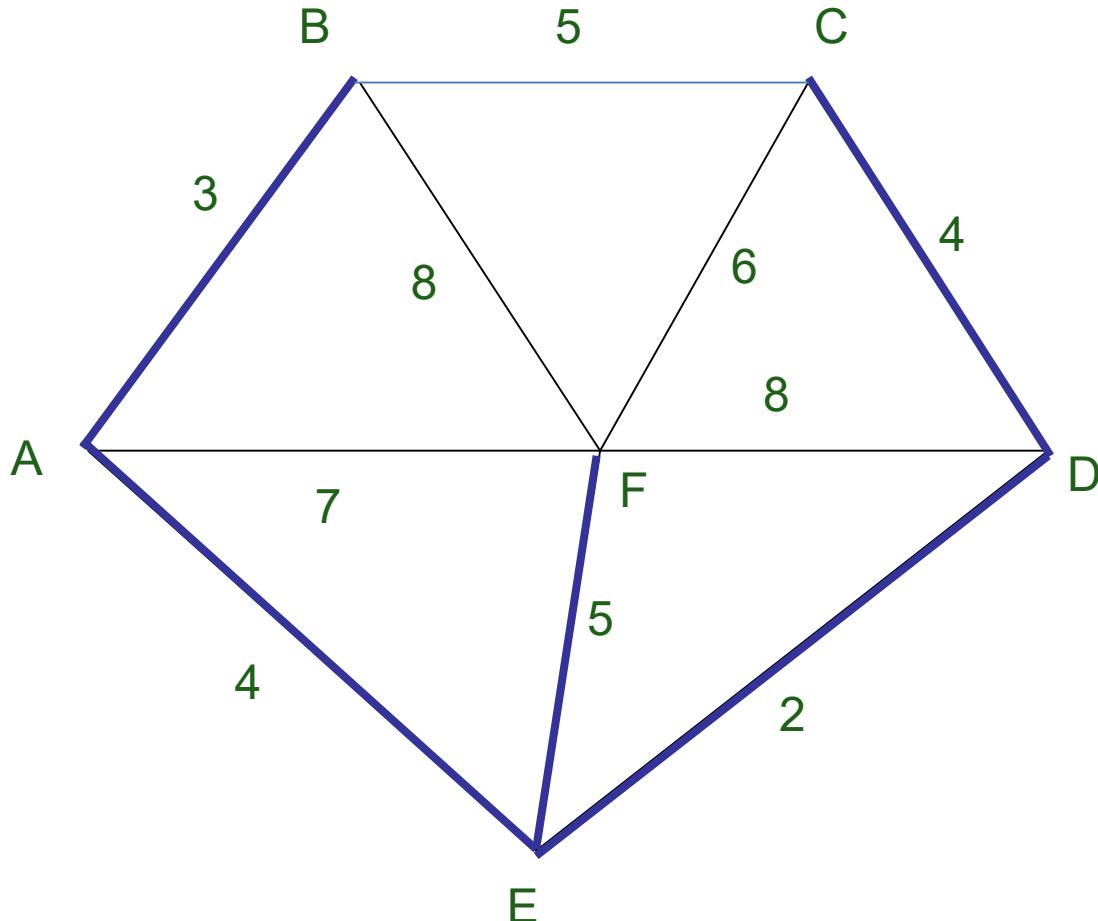
# Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED	2
AB	3
CD	4
AE	4
BC	5 - forms a cycle
EF	5

# Kruskal's Algorithm



All vertices have been connected.

The solution is

ED	2
AB	3
CD	4
AE	4
EF	5

Total weight of tree:  
18

# Algorithm

```
function Kruskal ( $G=(N,A)$ : graph ; length :  $A \rightarrow R^+$ ):set of edges
{initialisation}
    sort A by increasing length
     $N \leftarrow$  the number of nodes in N
     $T \leftarrow \emptyset$  {will contain the edges of the minimum spanning tree}
    initialise n sets, each containing the different element of N
{greedy loop}
repeat
     $e \leftarrow \{u, v\} \leftarrow$  shortest edge not yet considered
     $ucomp \leftarrow \text{find}(u)$ 
     $vcomp \leftarrow \text{find}(v)$ 
    if  $ucomp \neq vcomp$  then
        merge( $ucomp, vcomp$ )
         $T \leftarrow T \cup \{e\}$ 
until  $T$  contains  $n-1$  edges
return  $T$ 
```

## Kruskal's Algorithm: complexity

$O(E \log V)$

$O(V)$

$O(E \log V)$

Sorting loop:

Initialization of components:

Finding and merging:

Worst case complexity -

$O(E \log V)$

## Contd...

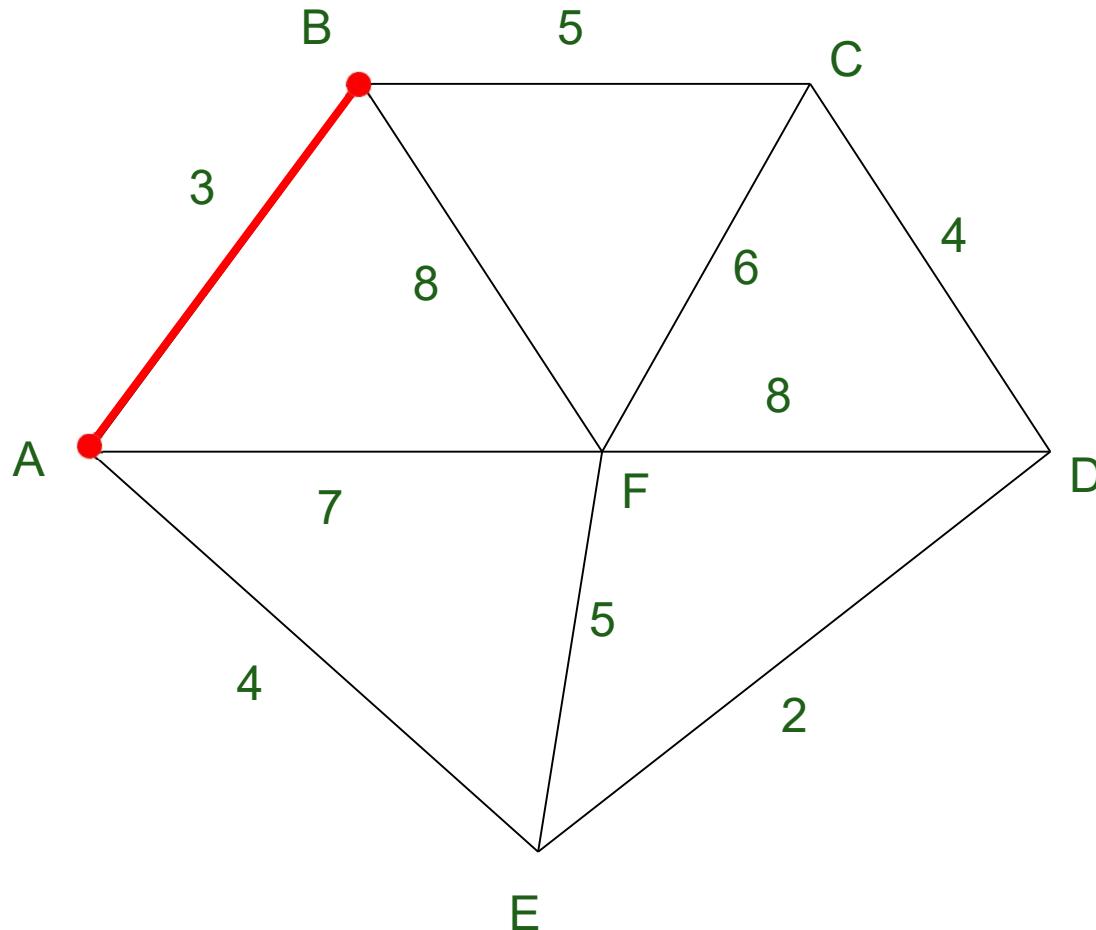


- The edges are maintained as min heap.
- The next edge can be obtained in  $O(\log E)$  time if graph has  $E$  edges.
- So, Kruskal's Algorithm takes  $O(E \log E)$  time.
- The value of  $E$  can be at most  $O(V^2)$ .
- So,  $O(\log V)$  and  $O(\log E)$  are same.

### Special Case-

- If the edges are already sorted, then there is no need to construct min heap.
- So, deletion from min heap time is saved.
- In this case, time complexity of Kruskal's Algorithm =  $O(E + V)$

# Prim's Algorithm



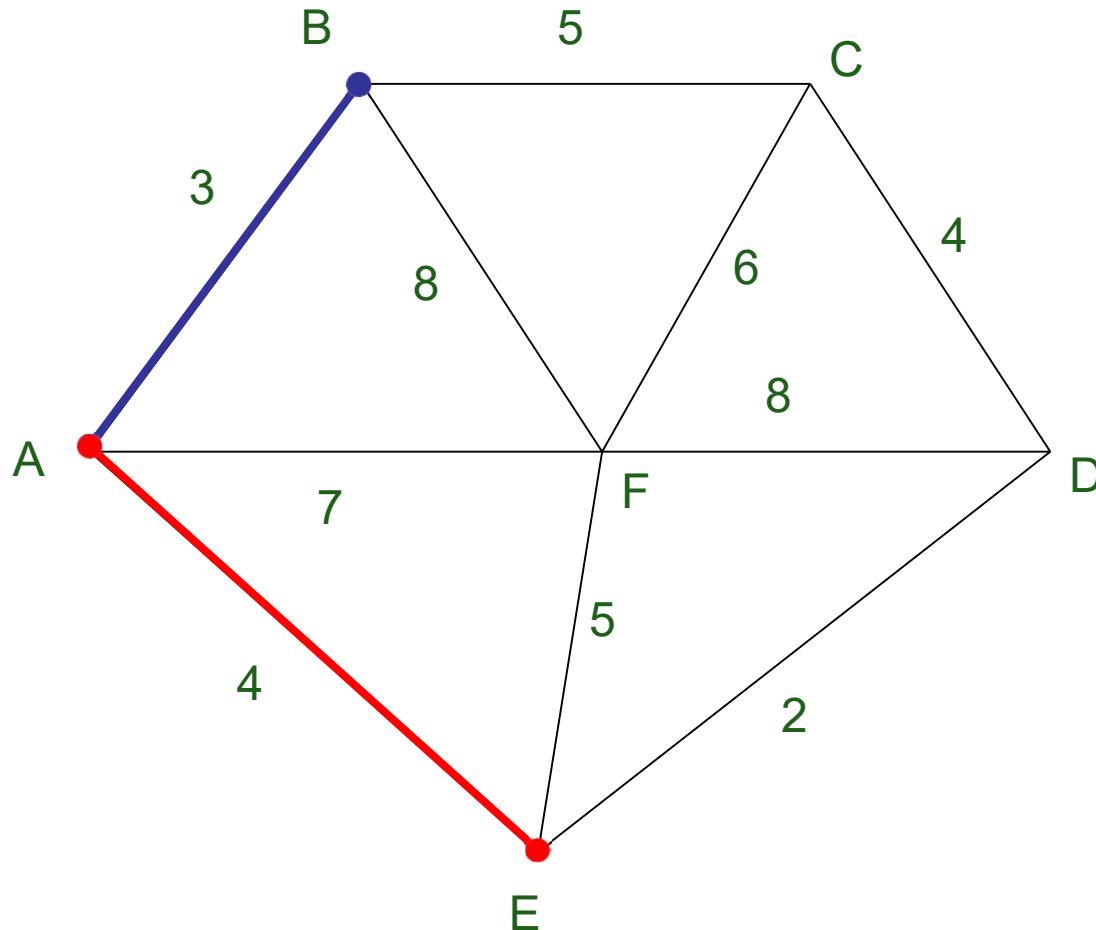
Select any vertex

A

Select the shortest  
edge connected to  
that vertex

AB 3

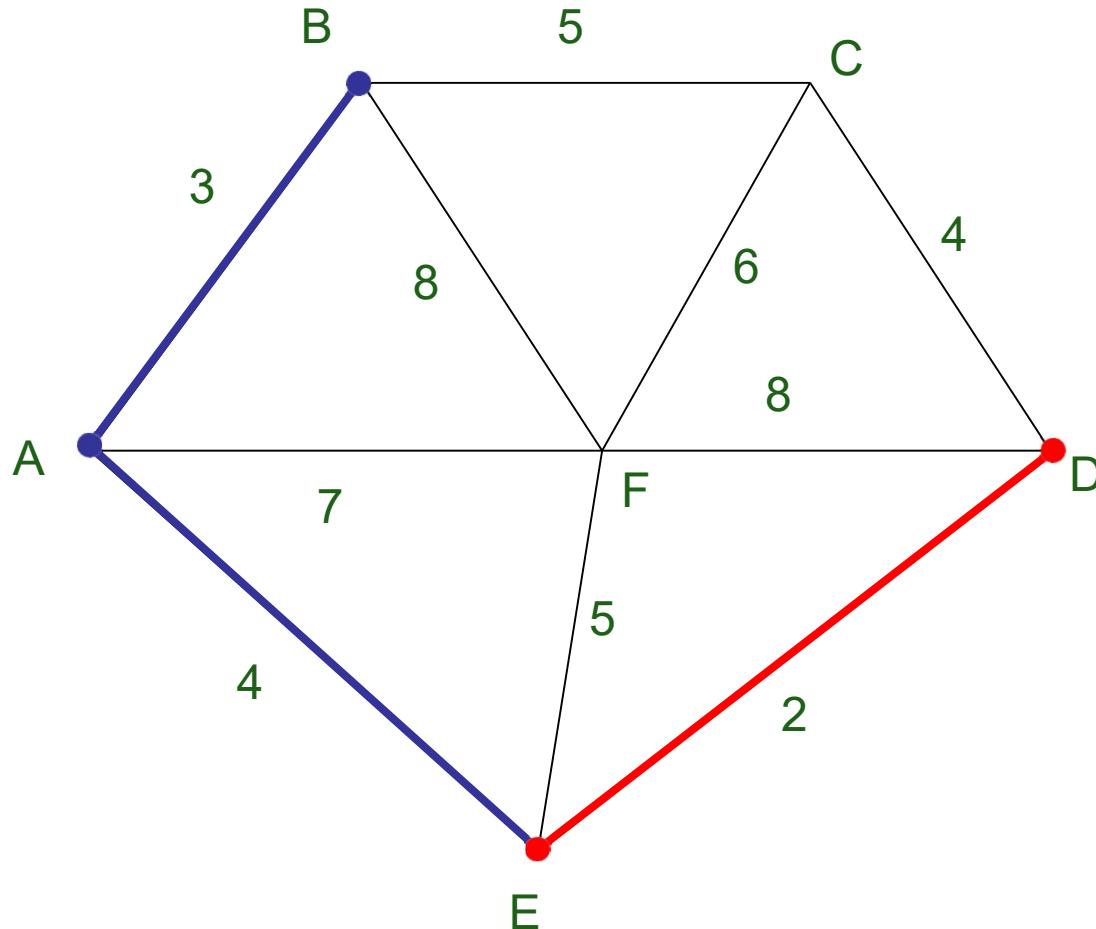
# Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

AE 4

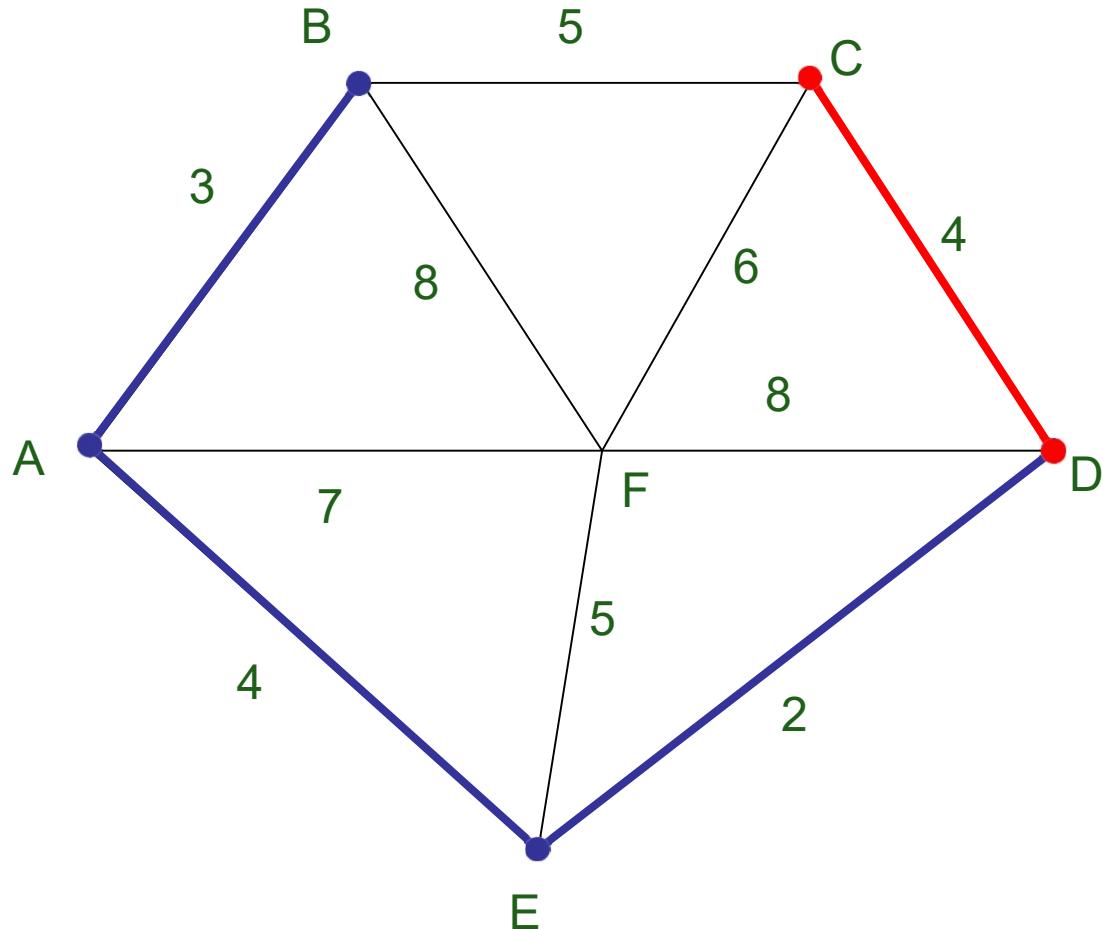
# Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

ED 2

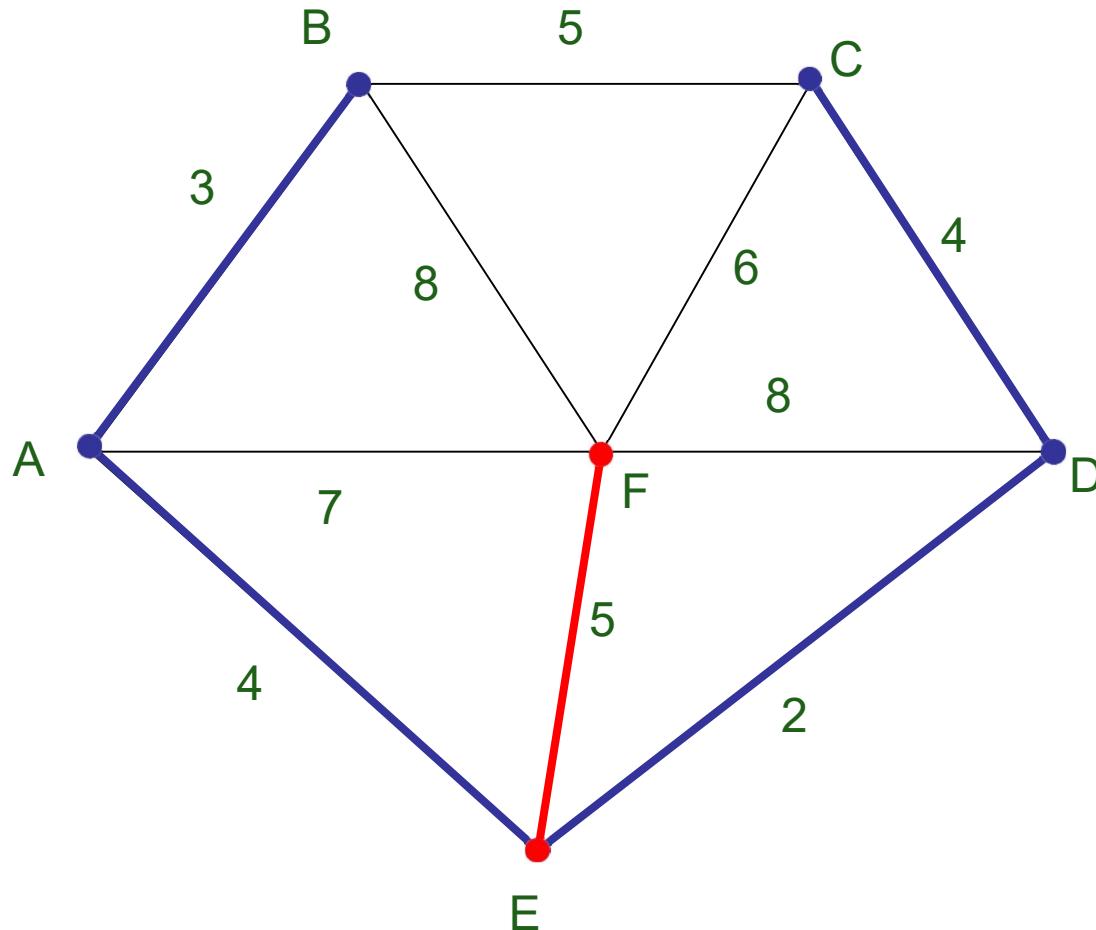
# Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

DC 4

# Prim's Algorithm

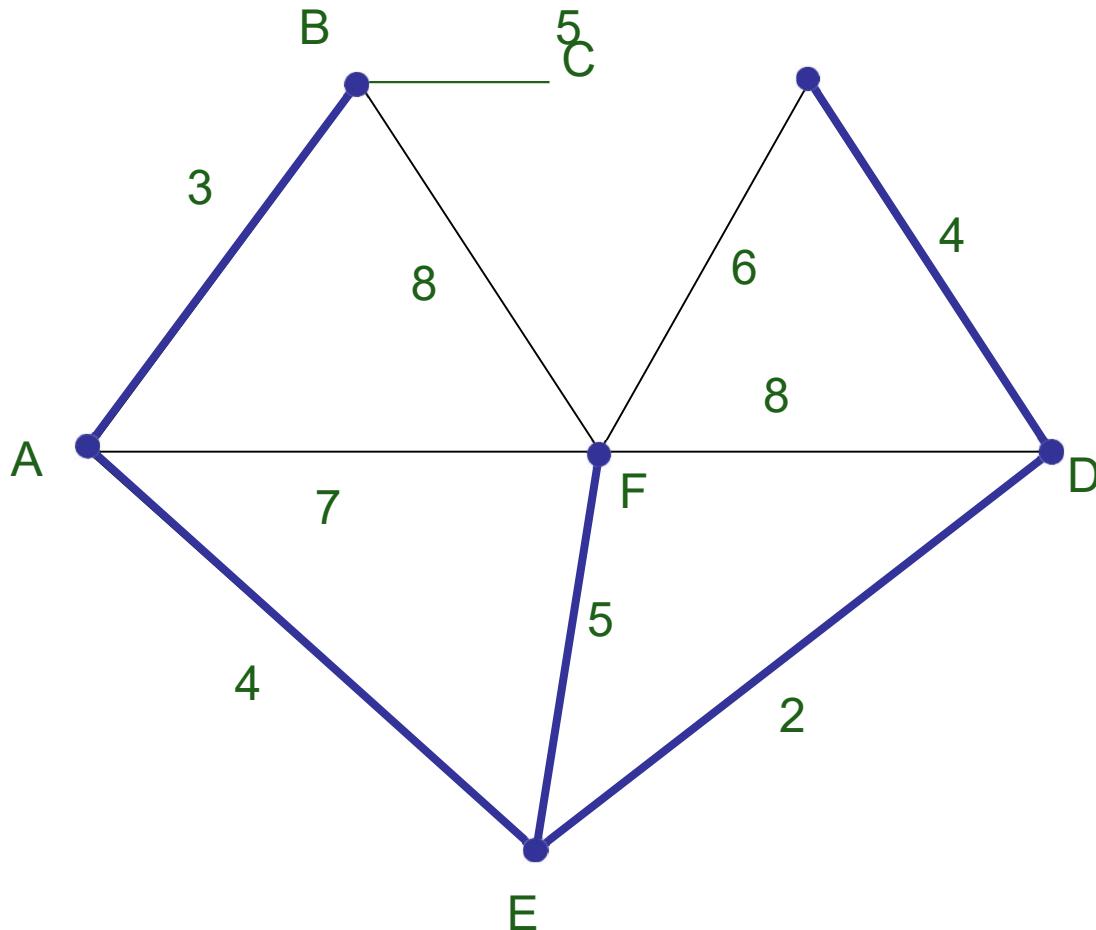


Select the shortest edge connected to any vertex already connected.

EF

5

# Prim's Algorithm



All vertices have been connected.

The solution is

**AB 3  
AE 4  
ED 2  
DC 4  
EF 5**

Total weight of tree:  
**18**

# Prim's Algorithm

function Prim( $G = \langle N, A \rangle$ : graph ; length :  $A \rightarrow \mathbb{R}^+$ ) : set of edges

{initialisation}

$T \leftarrow \emptyset$

$B \leftarrow \{\text{an arbitrary member of } N\}$

While  $B \neq N$  do

    find  $e = \{u, v\}$  of minimum length such  
         $u \in B$  and  $v \in N \setminus B$

$T \leftarrow T \cup \{e\}$

$B \leftarrow B \cup \{v\}$

Return  $T$

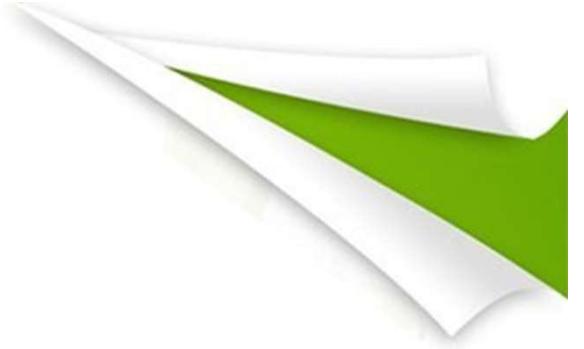
## Complexity:

Outer loop:  $n-1$  times

Inner loop:  $n$  times

$O(n^2)$  if uses

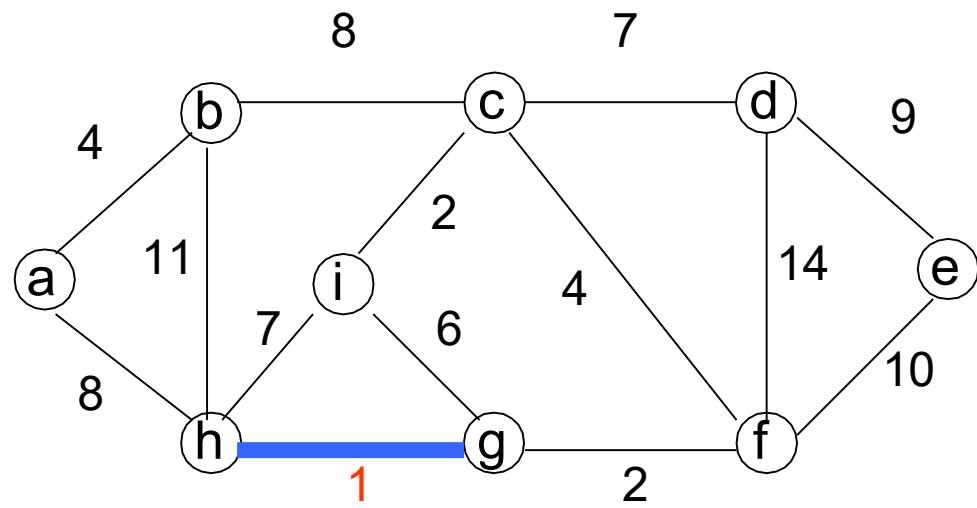
adjacency matrix



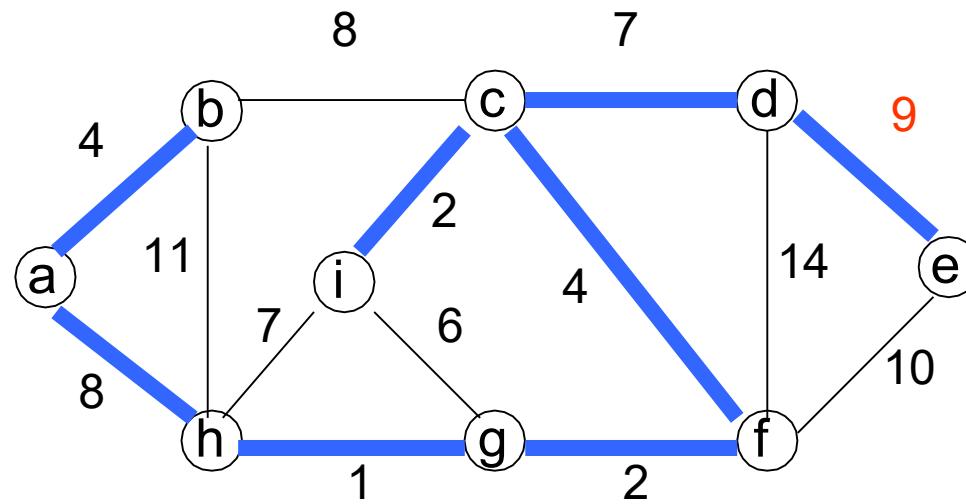
**Contd...**

**The time complexity is  $O(V \log V + E \log V) = O(E \log V)$ , making it the same as Kruskal's algorithm. However, Prim's algorithm can be improved using Fibonacci Heaps (cf Cormen) to  $O(E + \log V)$ .**

# Example



# Solution



# Minimum Connector Algorithms

## Kruskal's algorithm

- Select the shortest edge in a network
- Select the next shortest edge which does not create a cycle
- Repeat step 2 until all vertices have been connected

## Prim's algorithm

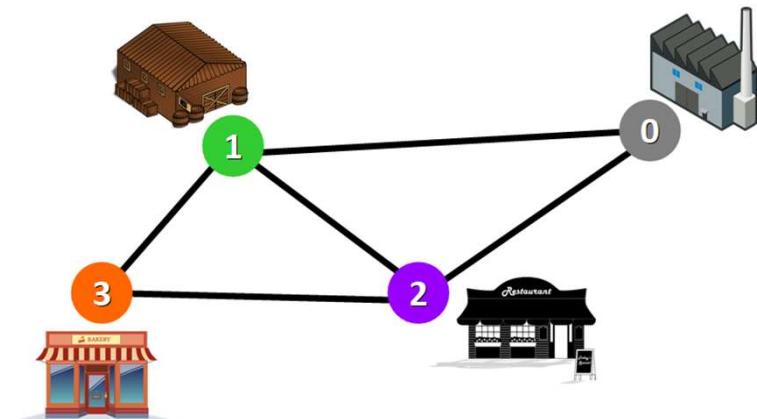
- Select any vertex
- Select the shortest edge connected to that vertex
- Select the shortest edge connected to any vertex already connected
- Repeat step 3 until all vertices have been connected



# Dijkstra's Algorithm

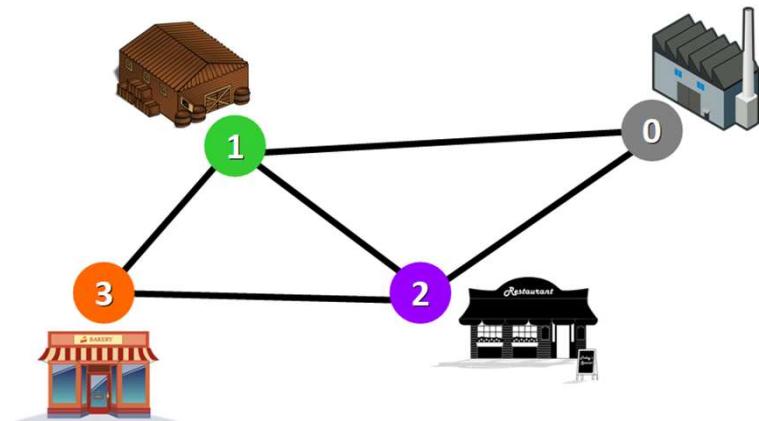
Presented by: Dr.  
Rakesh Kumar Sanodiya

## Dijkstra's Algorithm

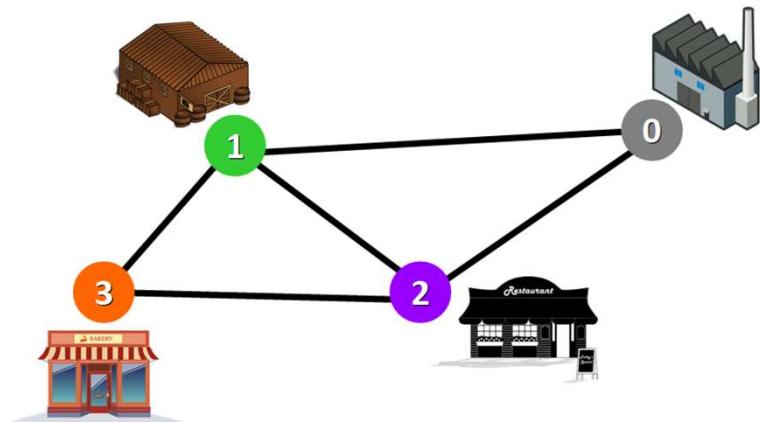


- derived by a Dutch computer scientist '**Edsger Wybe Dijkstra**' in 1956 and published in 1959
- a solution to the **single-source shortest** path problem in graph theory.
- Works on both **directed** and **undirected** graphs. However, all edges must have **nonnegative weights**.
- Approach: **Greedy**
- Input: Weighted graph  $G=\{E,V\}$  and source vertex, such that all edge weights are nonnegative
- Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex to all other vertices

## How it works?



- This algorithm finds the **path with lowest cost** (i.e. the shortest path) between that **vertex** and **every other vertex**
- For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, **Dijkstra's algorithm** can be used to find the **shortest route** between **one city** and **all other cities**.



## Numerical Algorithm?

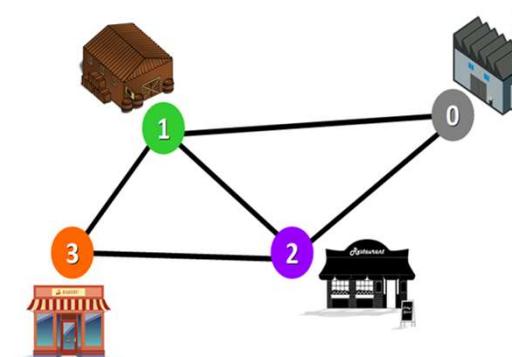
- **Formula** :  $O(|V|^2 + |E|) = O(|V|^2)$
- Where,  $E$ = Edges,  $V$ = Vertices  $|E|$  = Function of Edges  $|V|$  = Function of Vertices and  $O$  = Constant

## Algorithm:

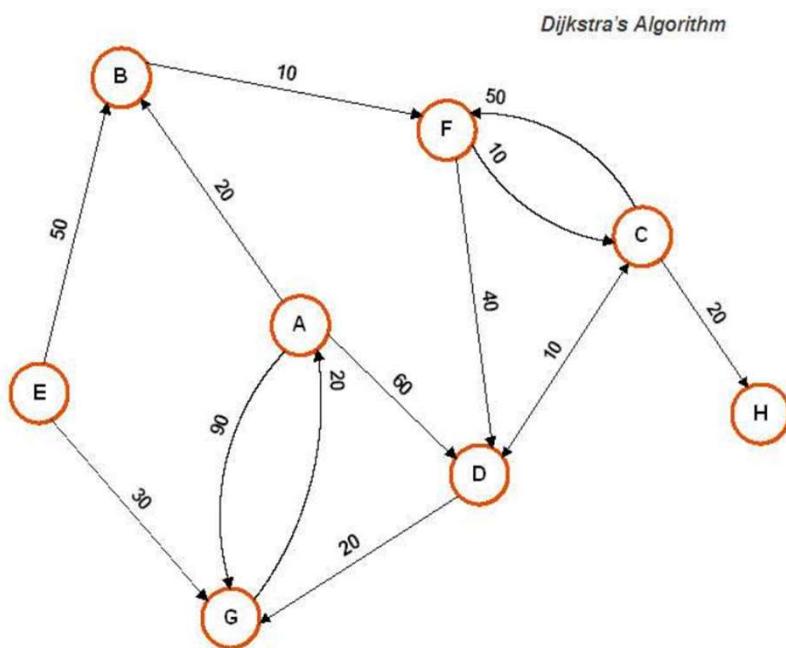
- 1) Create a set shortest path tree set '**S**' that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as **0** for the **source vertex** so that it is picked first.
- 3) While **set S** doesn't include all vertices
  - a) Pick a vertex u which is not there in **set S** and has minimum distance value.
  - b) Include u to **set S**.
  - c) Update distance value of all adjacent vertices of u.

*To update the distance values, iterate through all adjacent vertices.*

*For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge {u,v}, is less than the current distance value of v, then update the distance value of v.*



## Graph Algorithm

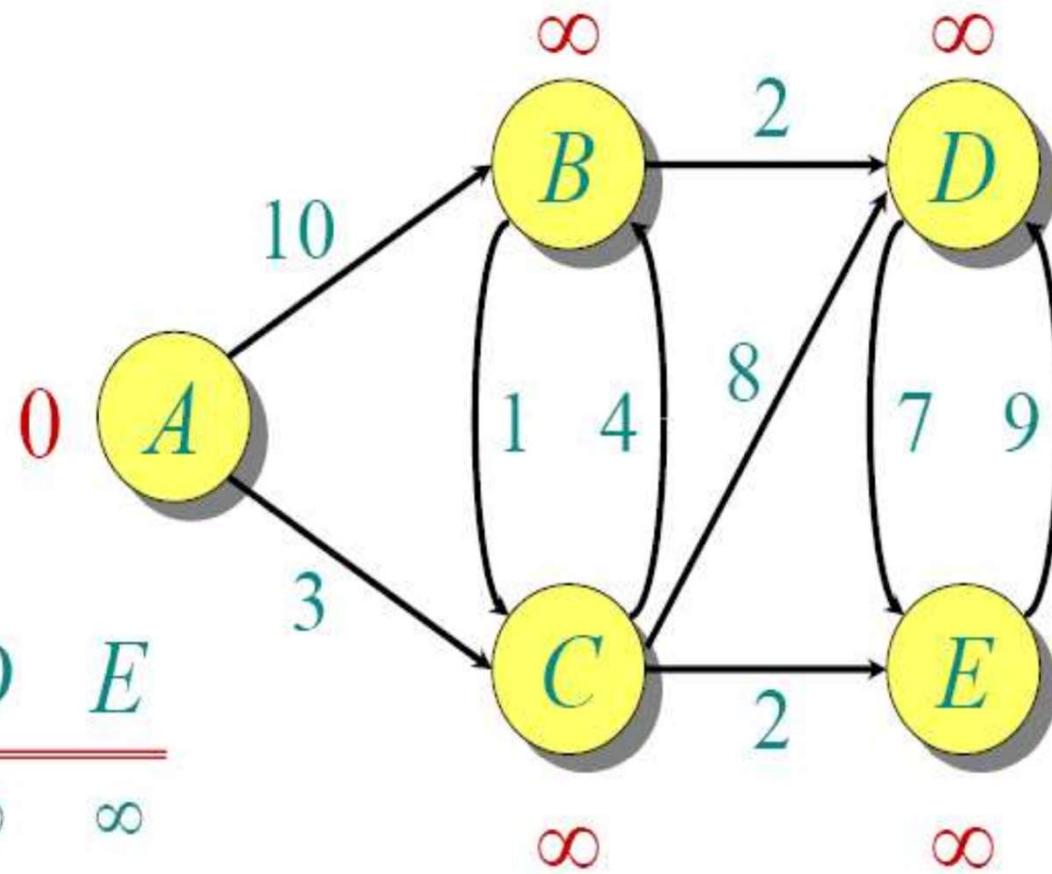


- In this interconnected 'Vertex' we'll use 'Dijkstra's Algorithm'.
- To use this algorithm in this network we have to start from a decided vertex and then continue to others

## Example

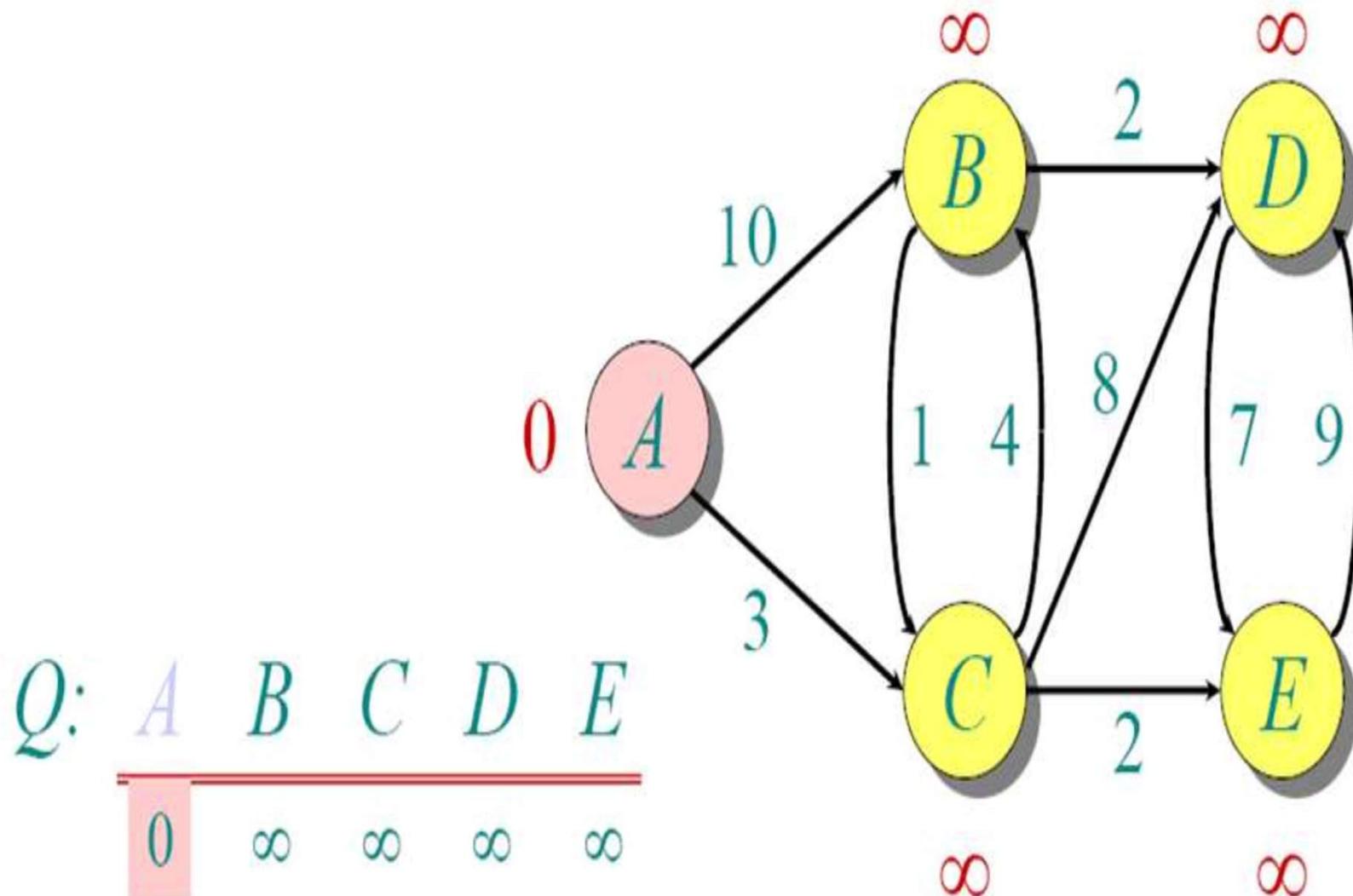
Initialize:

$$Q: \begin{array}{ccccc} A & B & C & D & E \\ \hline 0 & \infty & \infty & \infty & \infty \end{array}$$

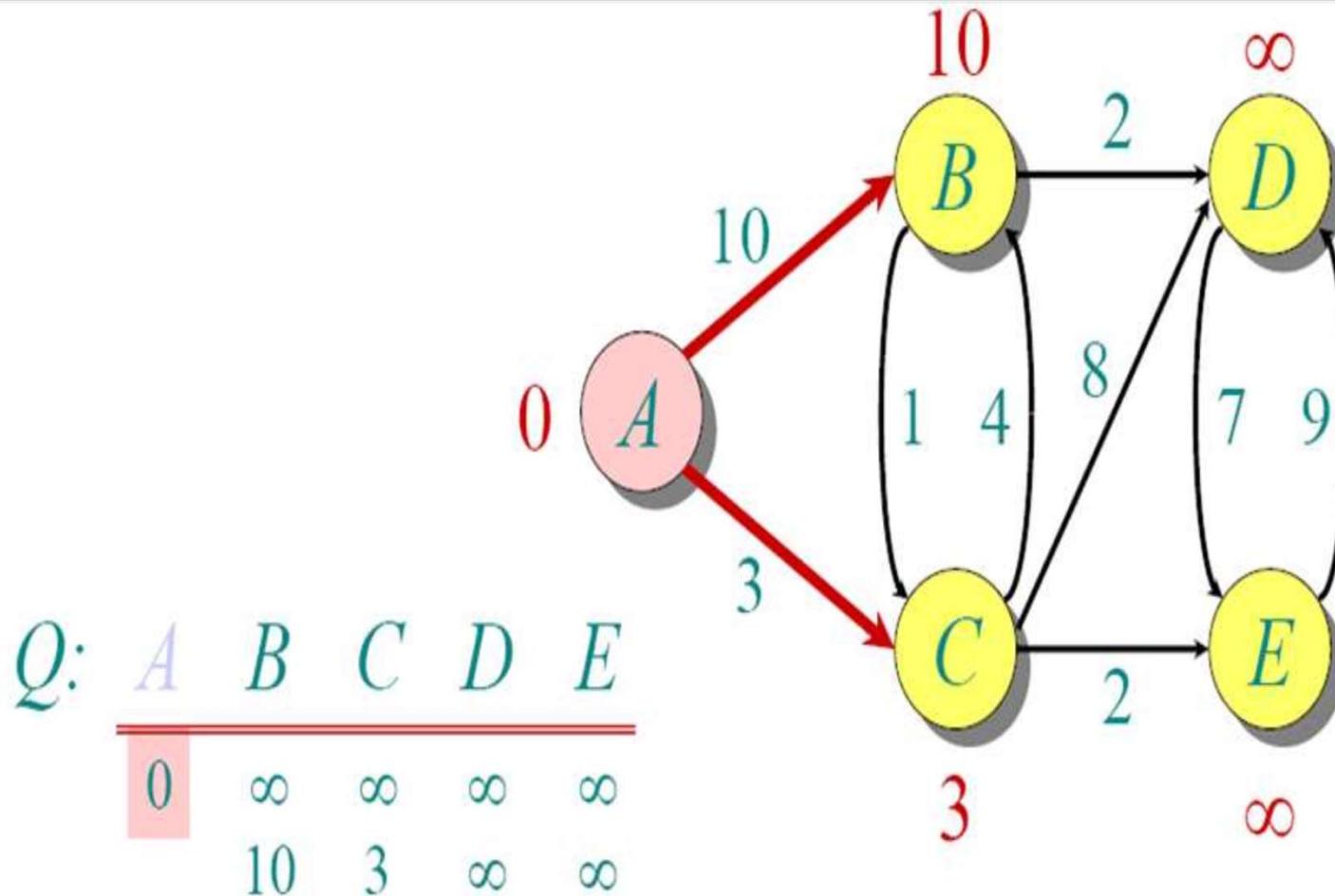


$$S: \{\}$$

## Example

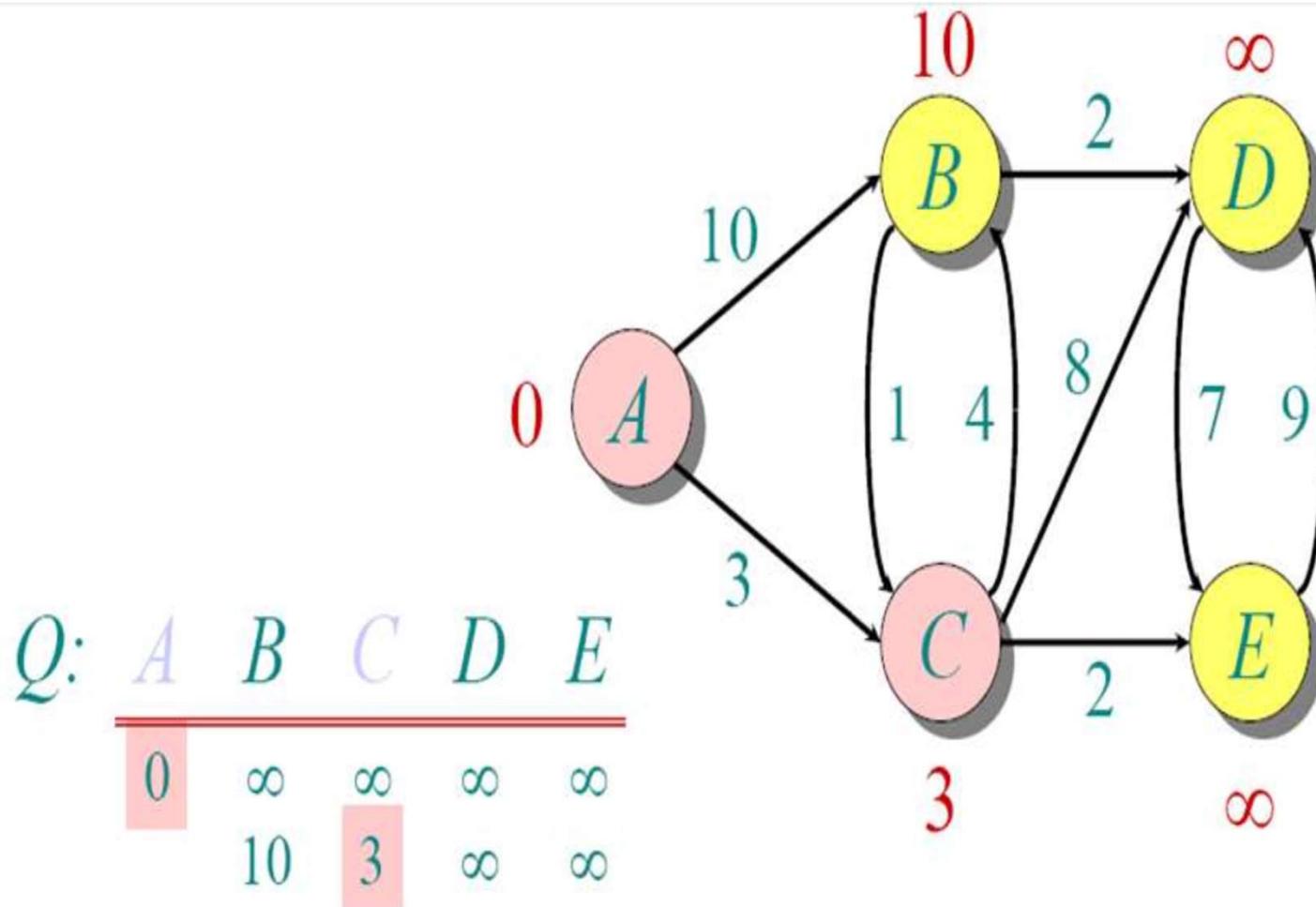


## Example



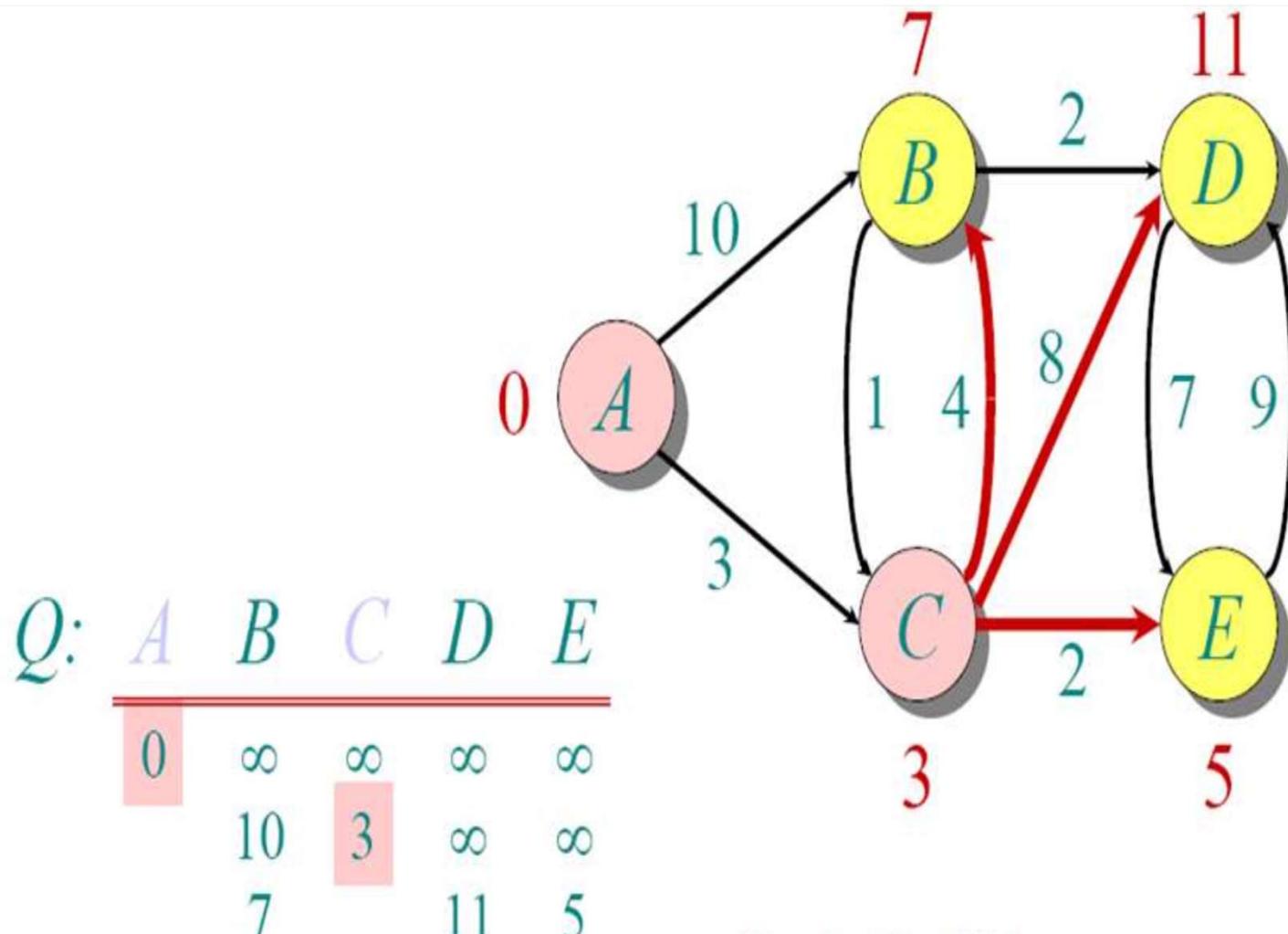
S: { A }

## Example



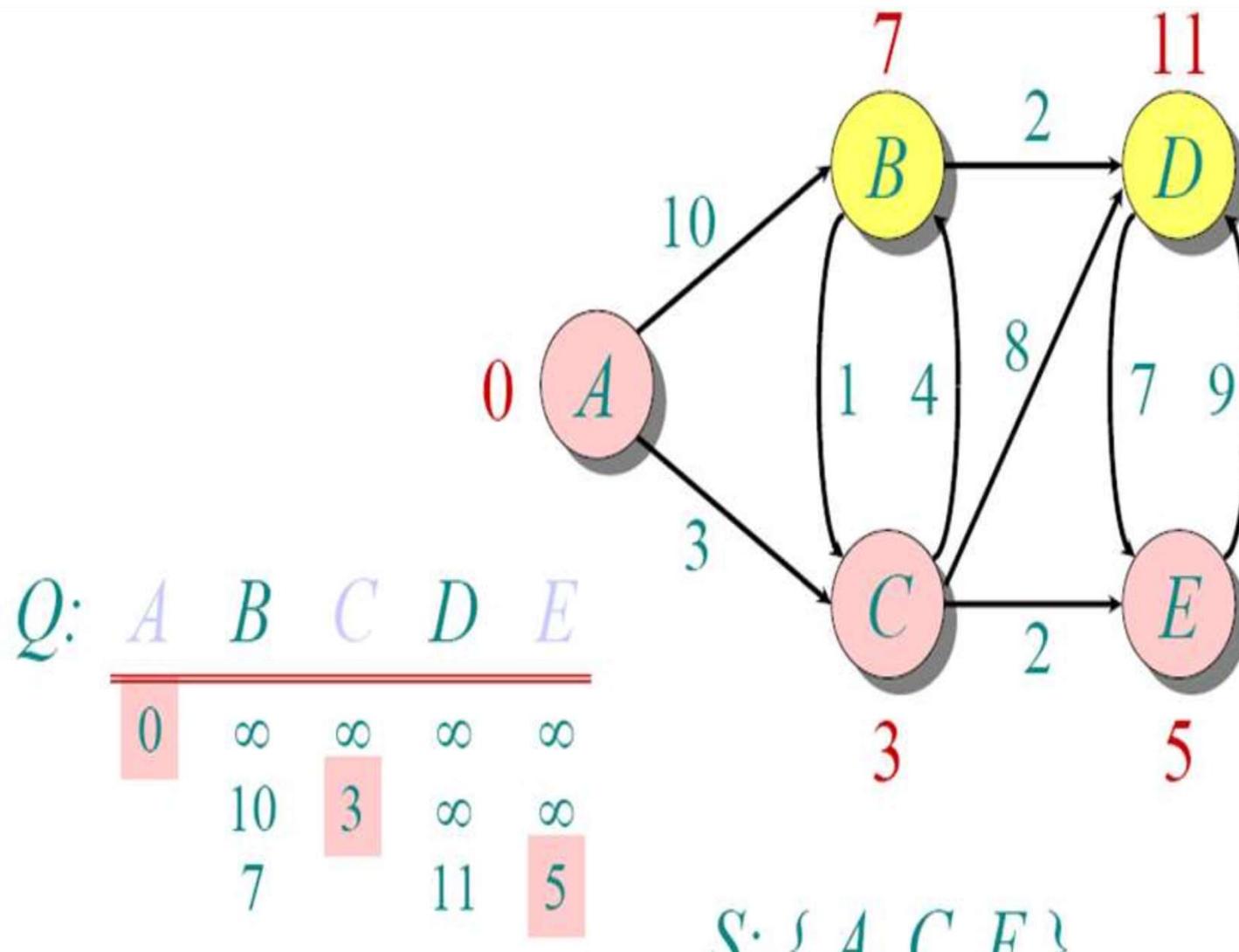
S: {A, C}

## Example

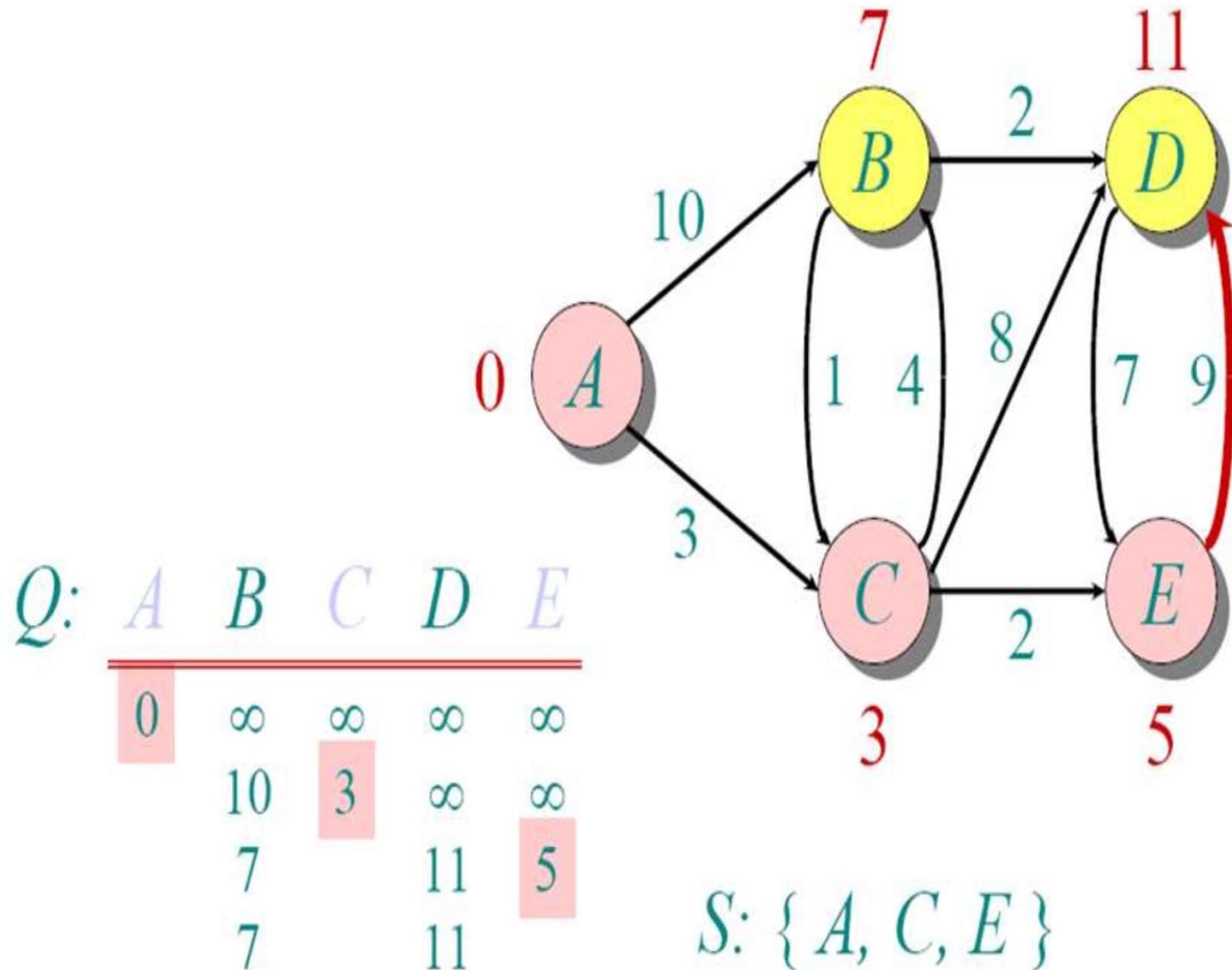


$S: \{ A, C \}$

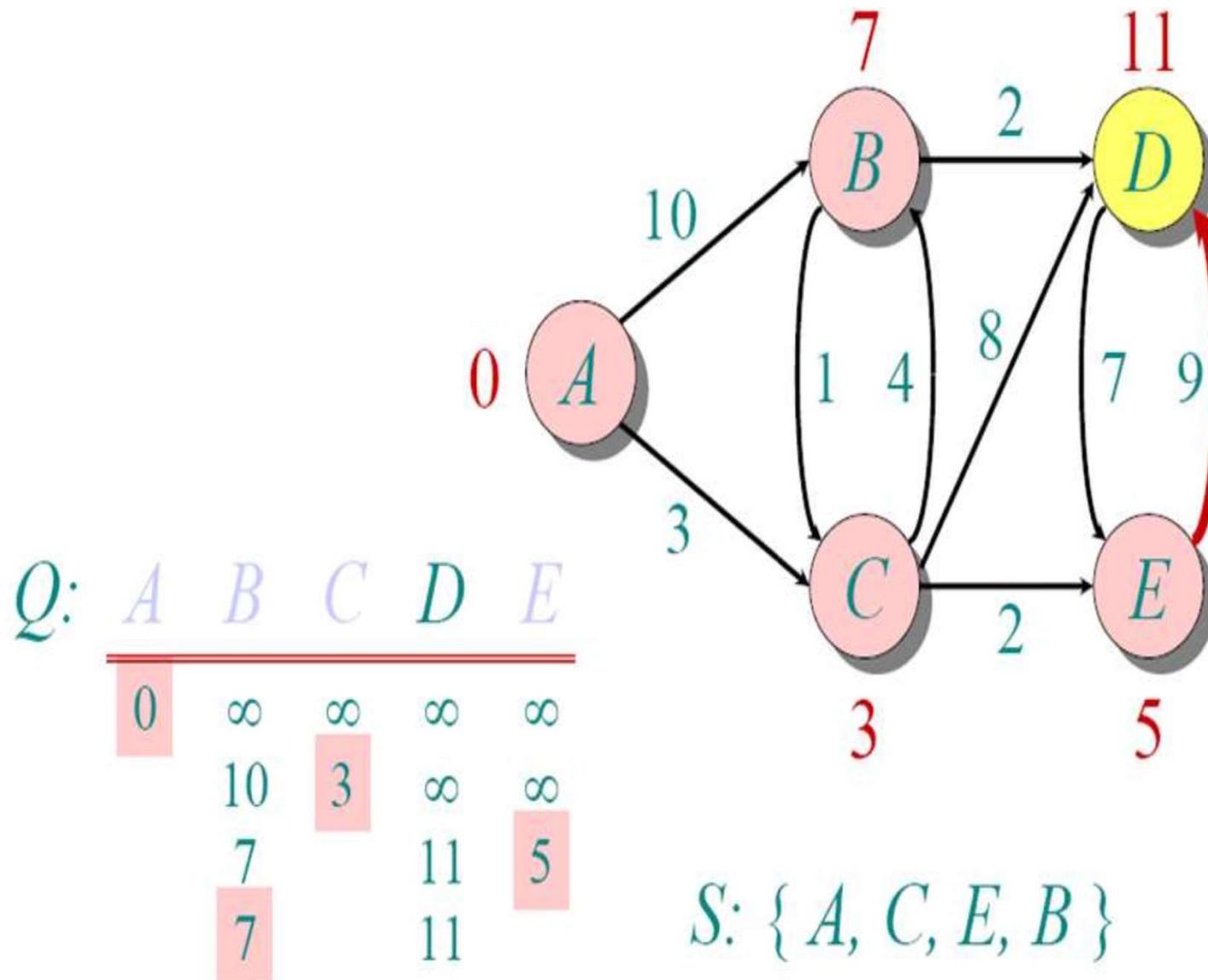
## Example



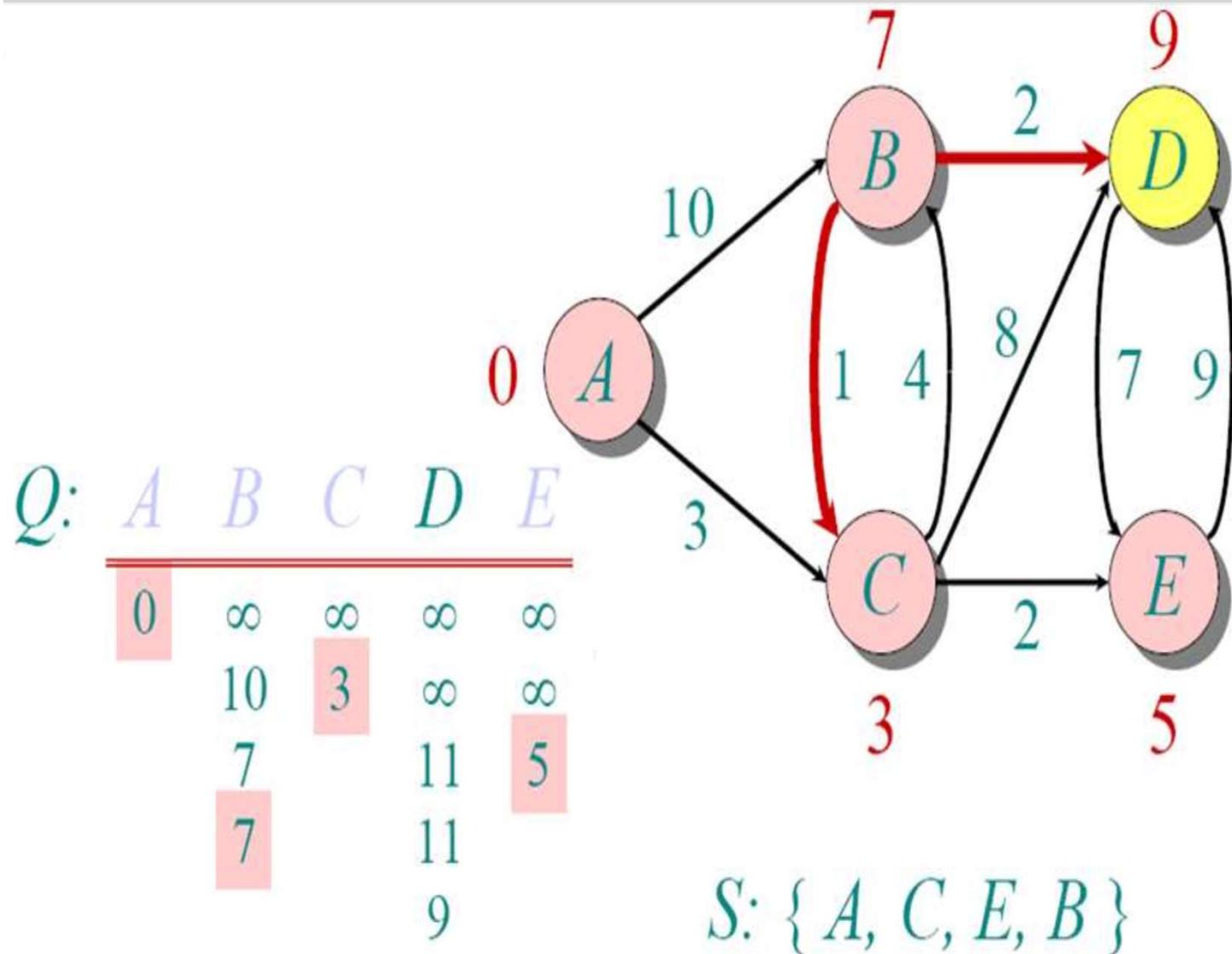
## Example



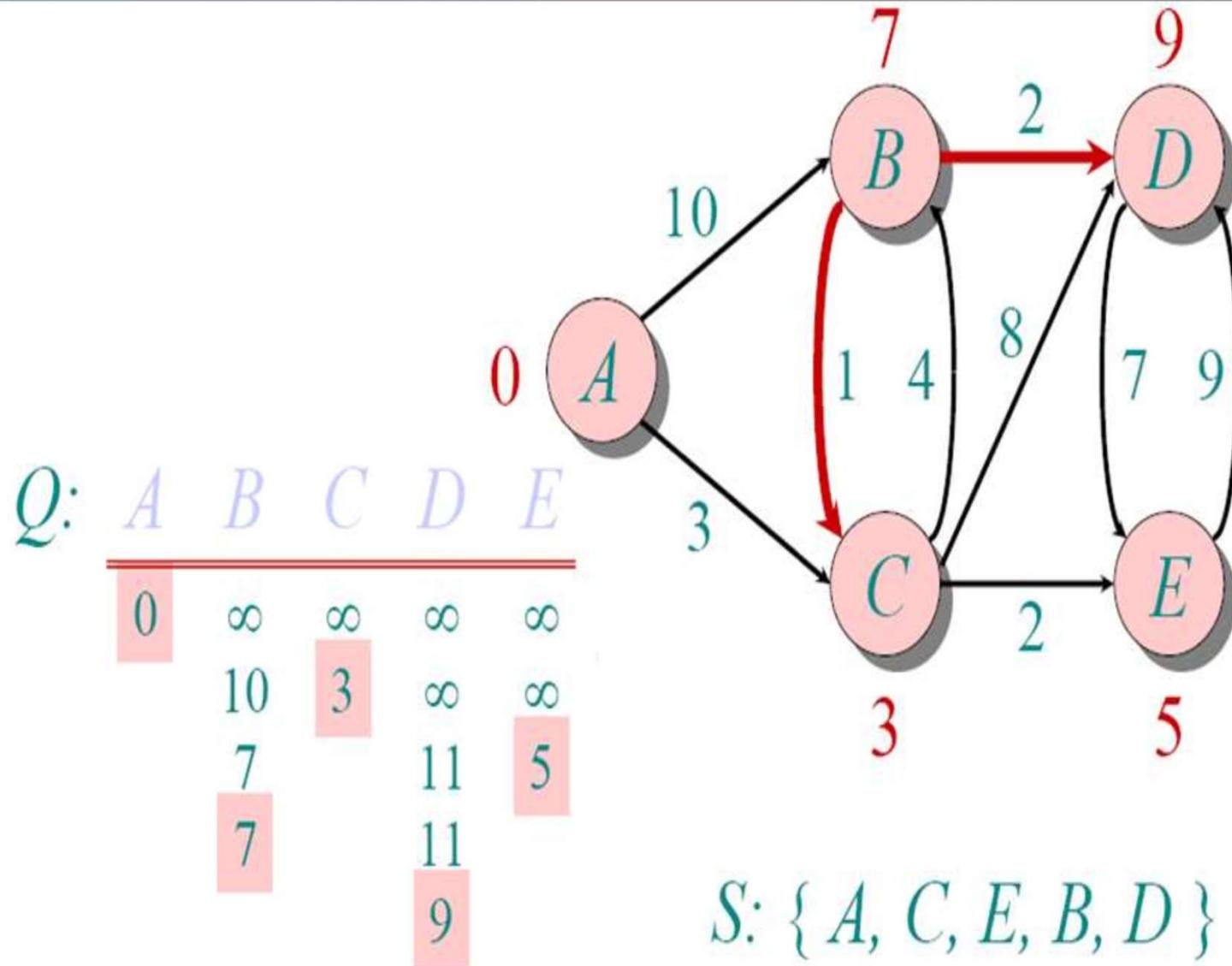
## Example



## Example



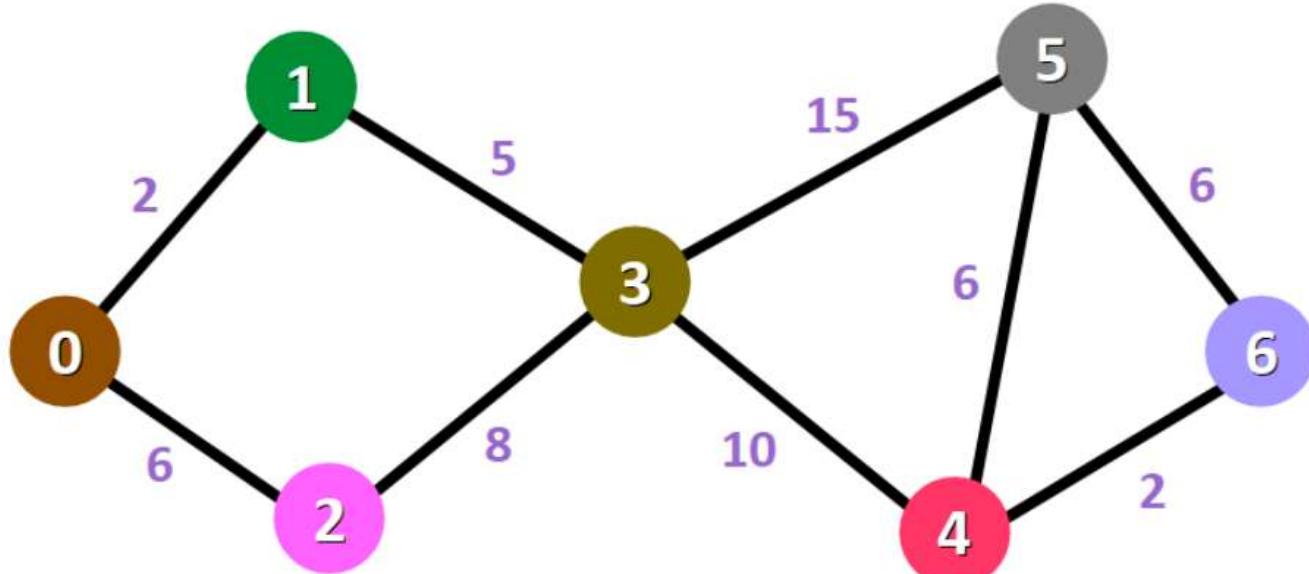
## Example



## Example-2

### Distance:

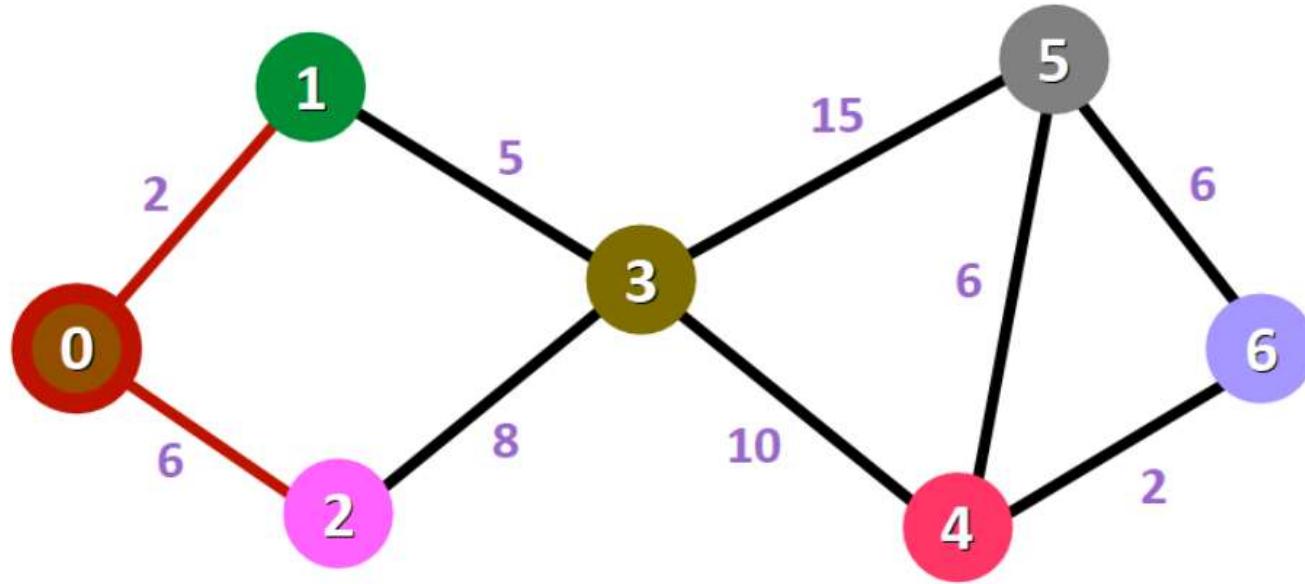
0:	0
1:	$\infty$
2:	$\infty$
3:	$\infty$
4:	$\infty$
5:	$\infty$
6:	$\infty$

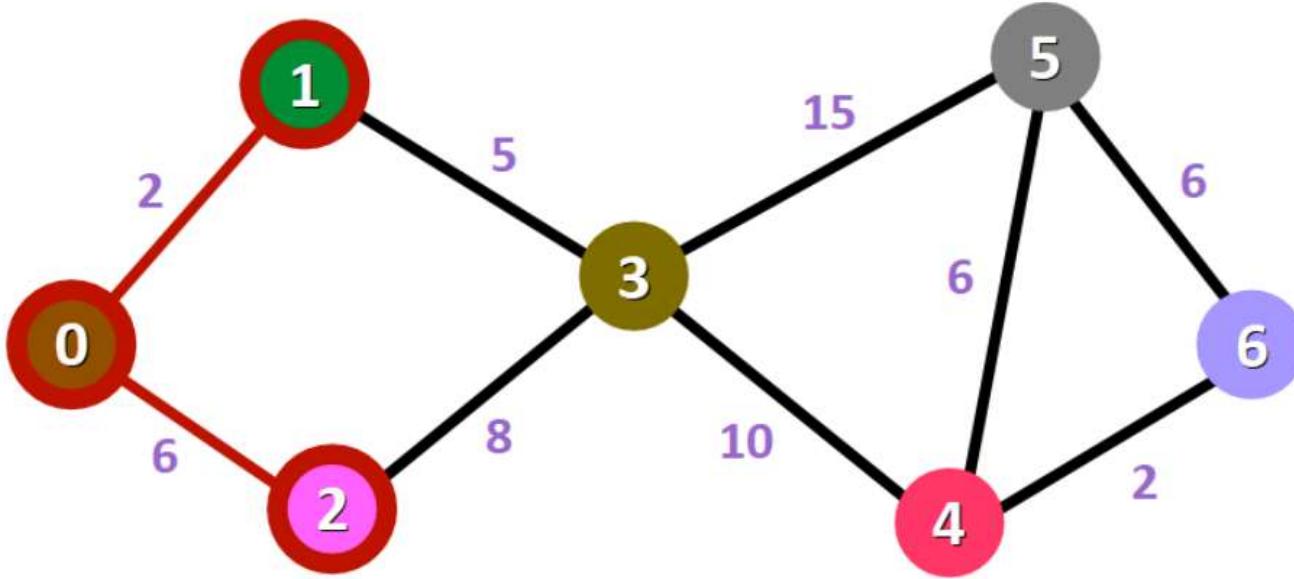


*Contd...*

**Distance:**

0:	0
1:	<del>∞</del> 2
2:	<del>∞</del> 6
3:	∞
4:	∞
5:	∞
6:	∞





### Distance:

0: 0

1: ~~0~~ 2 ■

2: ~~0~~ 6 ■

3: ~~0~~ 7 from (5 + 2) vs. 14 from (6 + 8)

4:  $\infty$

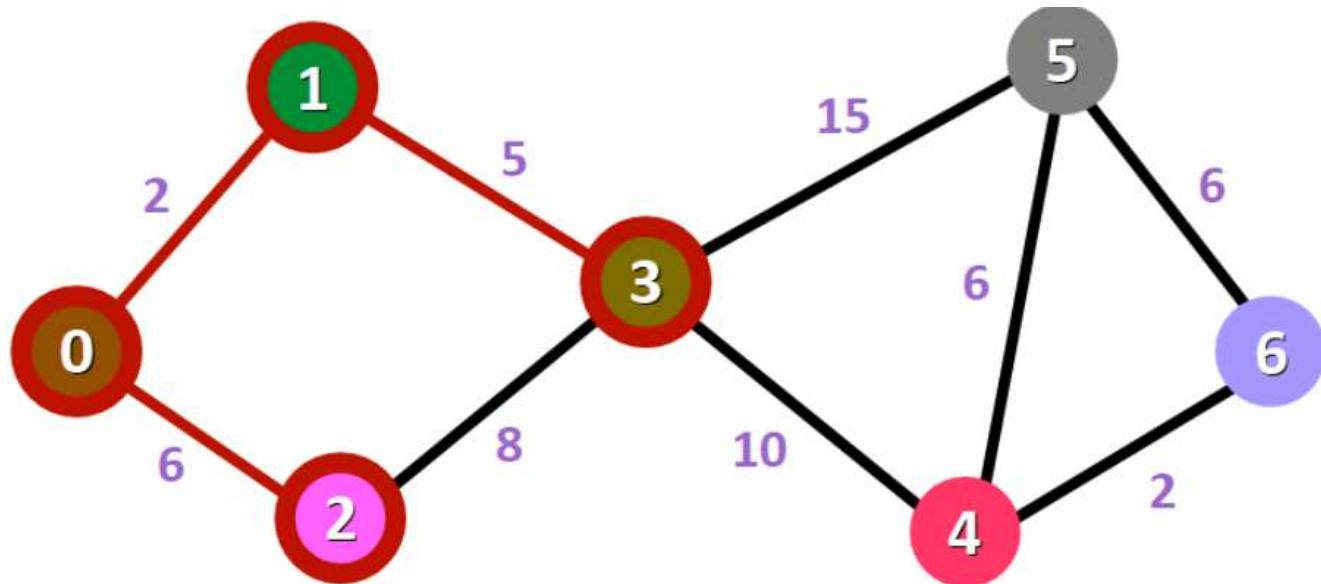
5:  $\infty$

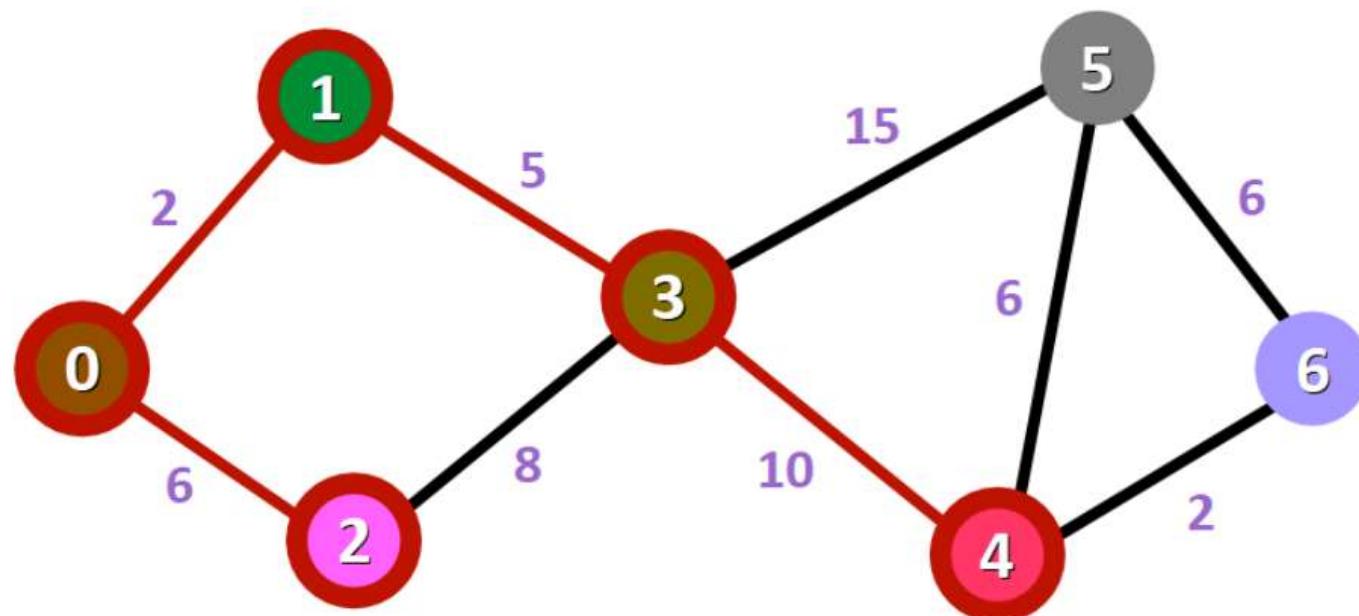
6:  $\infty$

*Contd...*

**Distance:**

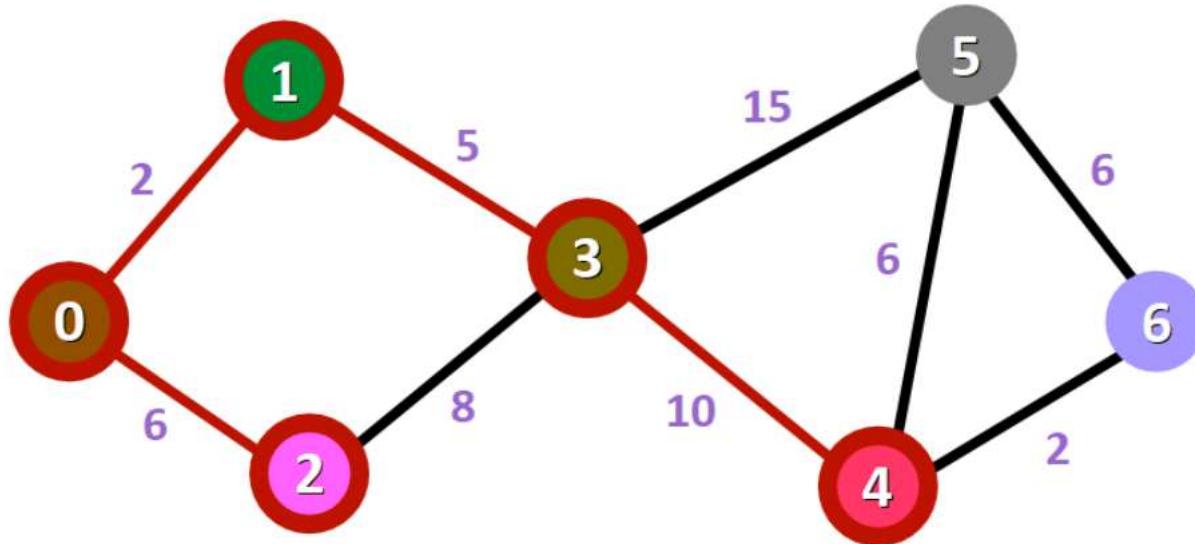
0:	0
1:	2
2:	6
3:	7
4:	$\infty$
5:	$\infty$
6:	$\infty$





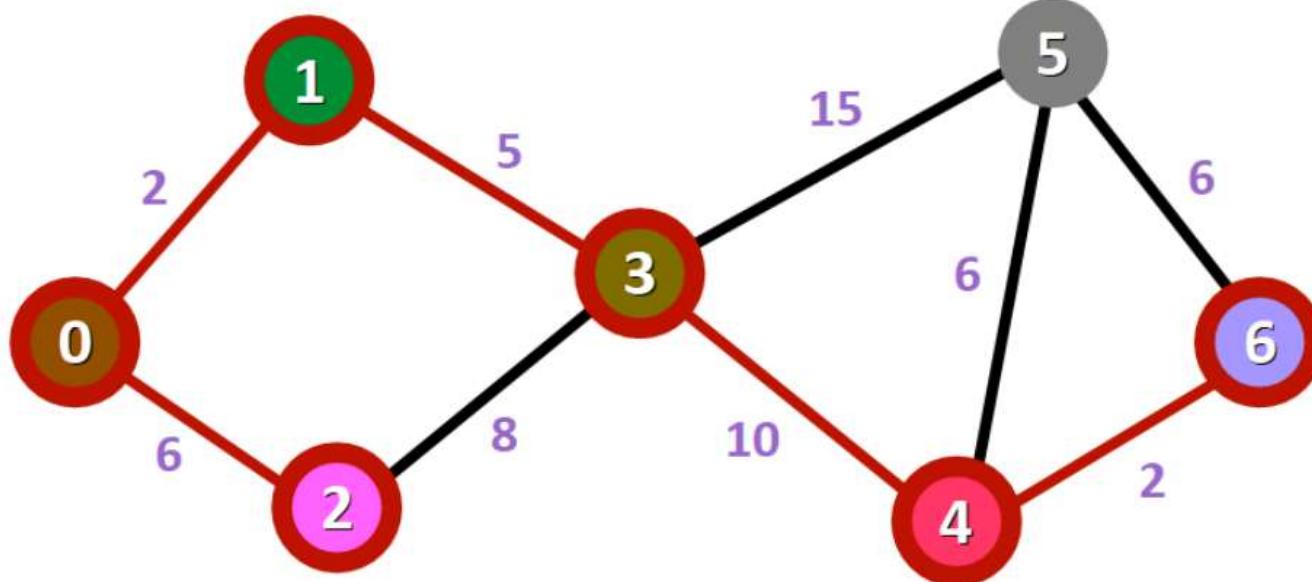
## Distance:

0:	0
1:	<del>∞</del> 2 ■
2:	<del>∞</del> 6 ■
3:	<del>∞</del> 7 ■
4:	<del>∞</del> 17 from $(2 + 5 + 10)$
5:	<del>∞</del> 22 from $(2 + 5 + 15)$
6:	$\infty$



## Distance:

0:	0
1:	<del>∞</del> 2 ■
2:	<del>∞</del> 6 ■
3:	<del>∞</del> 7 ■
4:	<del>∞</del> 17 ■
5:	<del>∞</del> 22 vs. 23 (2 + 5 + 10 + 6)
6:	<del>∞</del> 19 from (2 + 5 + 10 + 2)

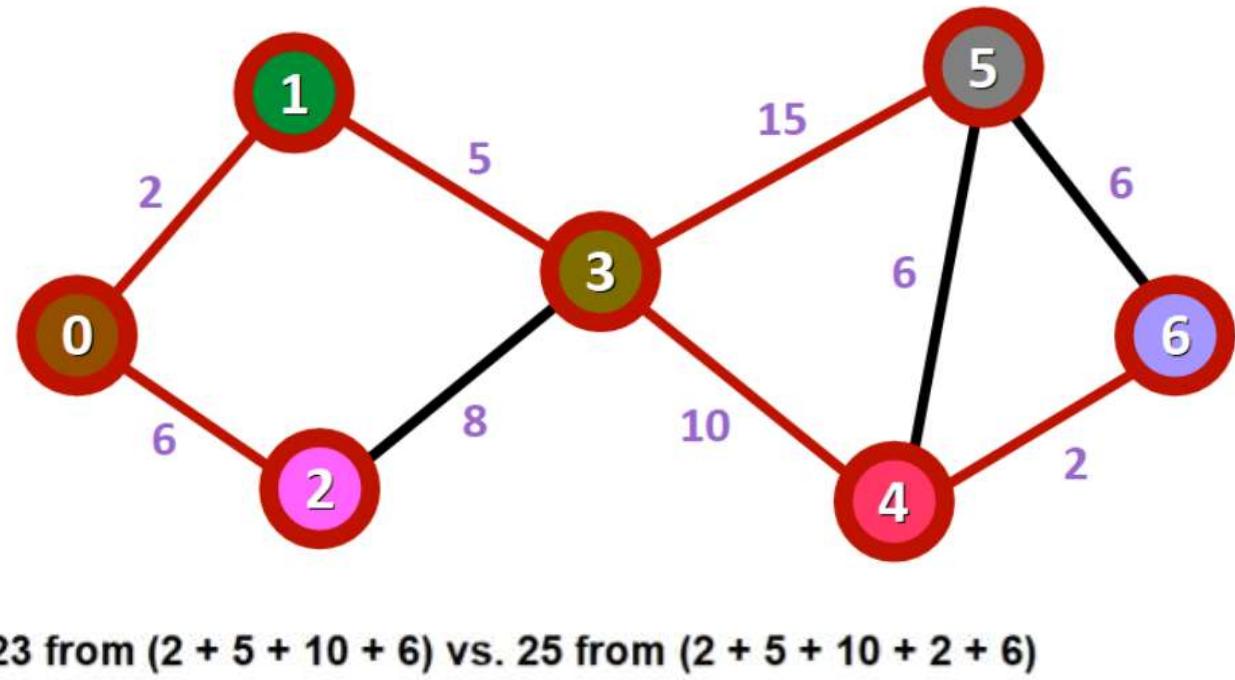


## Distance:

0:	0
1:	∞ 2 ■
2:	∞ 6 ■
3:	∞ 7 ■
4:	∞ 17 ■
5:	∞ 22
6:	∞ 19 ■

**Distance:**

0: 0  
1: ~~2~~ 2.  
2: ~~6~~ 6.  
3: ~~7~~ 7.  
4: ~~17~~ 17.  
5: ~~22 from (2 + 5 + 15) vs. 23 from (2 + 5 + 10 + 6) vs. 25 from (2 + 5 + 10 + 2 + 6)~~ 22 from (2 + 5 + 15) vs. 23 from (2 + 5 + 10 + 6) vs. 25 from (2 + 5 + 10 + 2 + 6)  
6: ~~19~~ 19.



Output

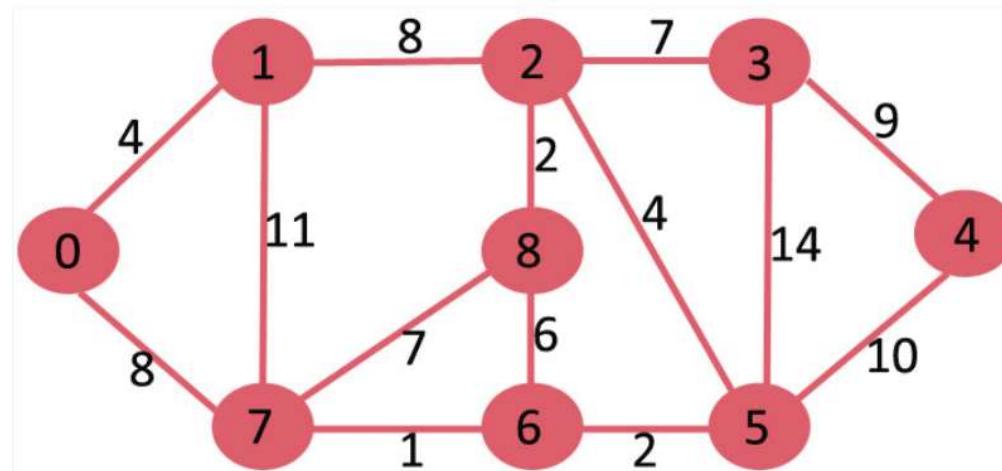
**Distance:**

0: 0  
1: ~~2~~ 2.  
2: ~~6~~ 6.  
3: ~~7~~ 7.  
4: ~~17~~ 17.  
5: ~~22~~ 22.  
6: ~~19~~ 19.

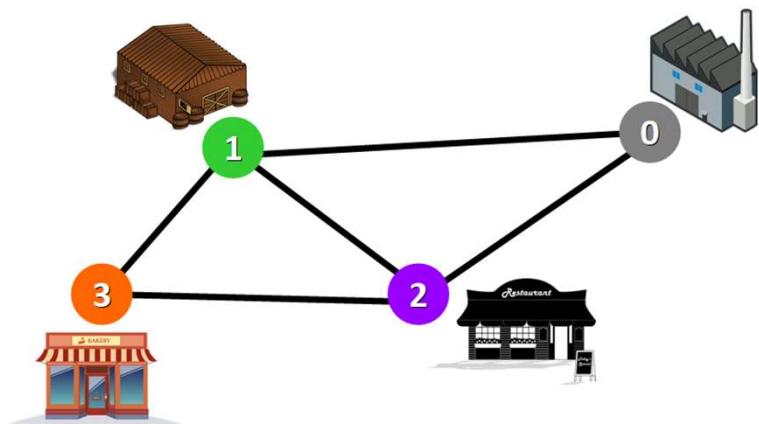
# Practice Problem

Output

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14



## Implementation & Run Time

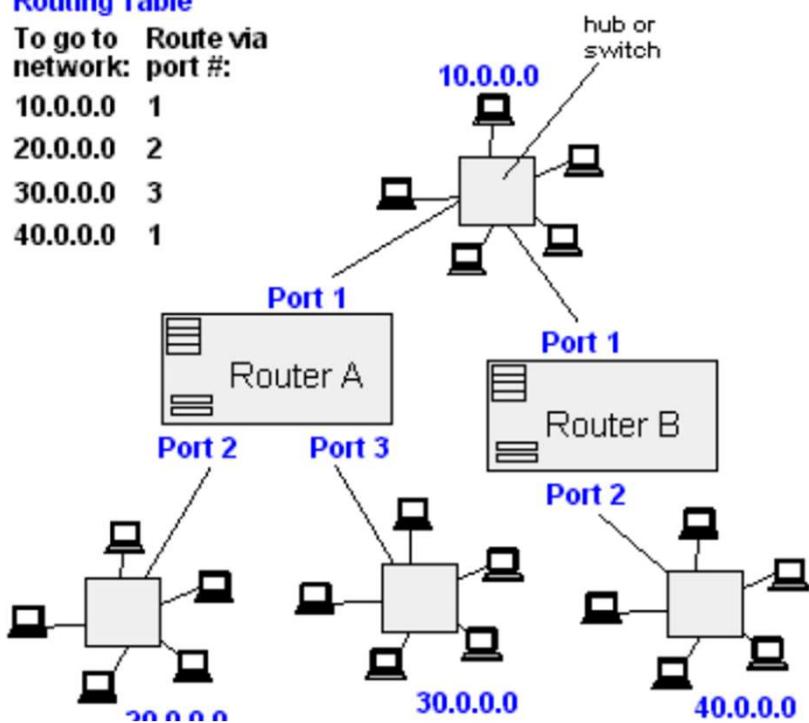


- The simplest implementation is to store vertices in an Array or Linked list. This will produce a running time of  $O(|V|^2 + |E|)$
- For graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a Binary Heap or Priority Queue. This will produce a running time of  $O((|E|+|V|) \log |V|)$

### Router A Routing Table

To go to network: route via port #:

10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1

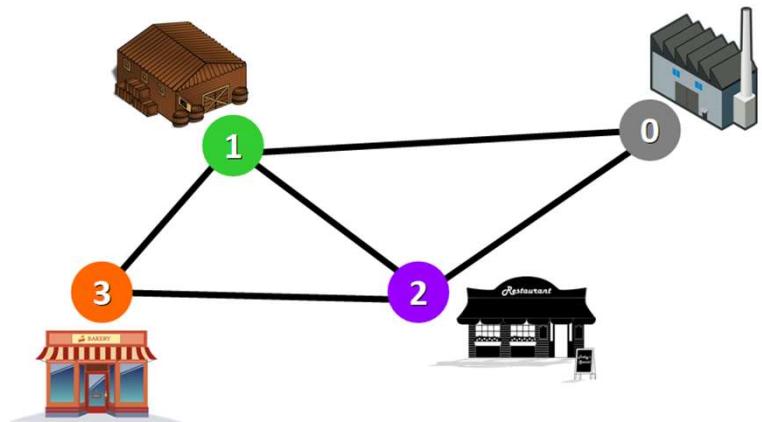


## Applications

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems.
- Social Networking Applications



## References



- Dijkstra's original paper: E. W. Dijkstra. (1959) A Note on Two Problems in Connection with Graphs. *Numerische Mathematik*, 1. 269-271.
- MIT OpenCourseware, 6.046J Introduction to Algorithms.
- Wikipedia.org
- Dijkstra's slide prepared by Rashik Ishrak Nahian



*Thank you!!*

