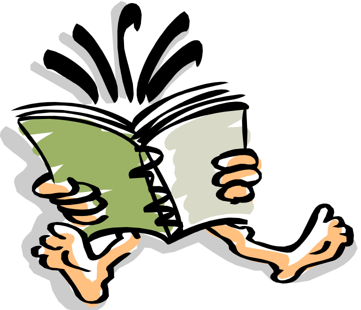
Analysis of Algorithms

Revision



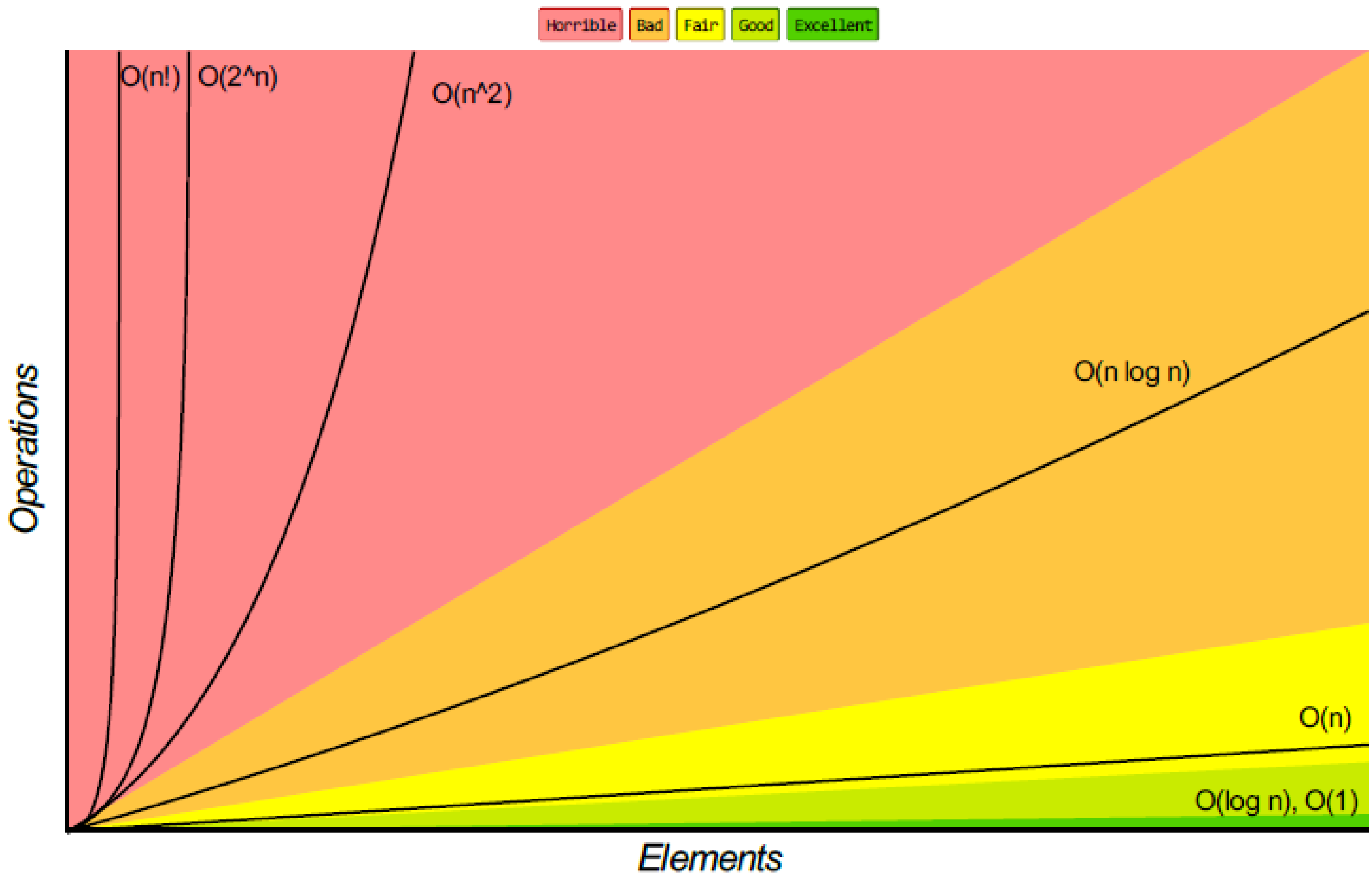
Asymptotic Notation

- O notation (Upper Bound): asymptotic “less than”:
  - f(n)=O(g(n)) implies: f(n) “≤” g(n)
- Ω notation (Lower Bound): asymptotic “greater than”
  - f(n)= Ω (g(n)) implies: f(n) “≥” g(n)
- Θ notation (Tight Bound): asymptotic “equality”
  - f(n)= Θ (g(n)) implies: f(n) “=” g(n)

Big-O Common Names

constant:	$O(1)$	
logarithmic:	$O(\log n)$	
linear:	$O(n)$	
log-linear:	$O(n \log n)$	
superlinear:	$O(n^{1+c})$	(c is a constant > 0)
quadratic:	$O(n^2)$	
polynomial:	$O(n^k)$	(k is a constant)
exponential:	$O(c^n)$	(c is a constant > 1)

Big-O Complexity Chart



Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Sorting Algorithms

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$
- Bubble Sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$
- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$
- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: $\Theta(n\lg n)$

Iteration Method – Example

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- s(n)

= n + s(n-1)

= n + n-1 + s(n-2)

= n + n-1 + n-2 + s(n-3)

= n + n-1 + n-2 + n-3 + s(n-4)

= …

= n + n-1 + n-2 + n-3 + … + n-(k-1) + s(n-k)

Substitution Method- Example

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

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le c\, n^2$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n-1) \le c(n-1)^2$ for all $k < n$
- Proof of induction goal:

$$T(n) = T(n-1) + n \le c\,(n-1)^2 + n$$

$$= cn^2 - (2cn - c - n) \le cn^2$$

$$\text{if: } 2cn - c - n \ge 0 \Leftrightarrow c \ge n/(2n-1) \Leftrightarrow c \ge 1/(2 - 1/n)$$

  - For $n \ge 1 \Rightarrow 2 - 1/n \ge 1 \Rightarrow$ any $c \ge 1$ will work

Master's Method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:

$$T(n) = \Theta(f(n))$$

regularity condition

Master's Method- Example

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

$a = 2$, $b = 2$, $\log_2 2 = 1$

Compare n with $f(n) = n^2$

$\Rightarrow f(n) = \Omega(n^{1+\varepsilon})$ Case 3 $\Rightarrow$ verify regularity cond.

$a\, f(n/b) \le c\, f(n)$

$\Leftrightarrow 2\, n^2/4 \le c\, n^2 \Rightarrow c = \frac{1}{2}$ is a solution $(c<1)$

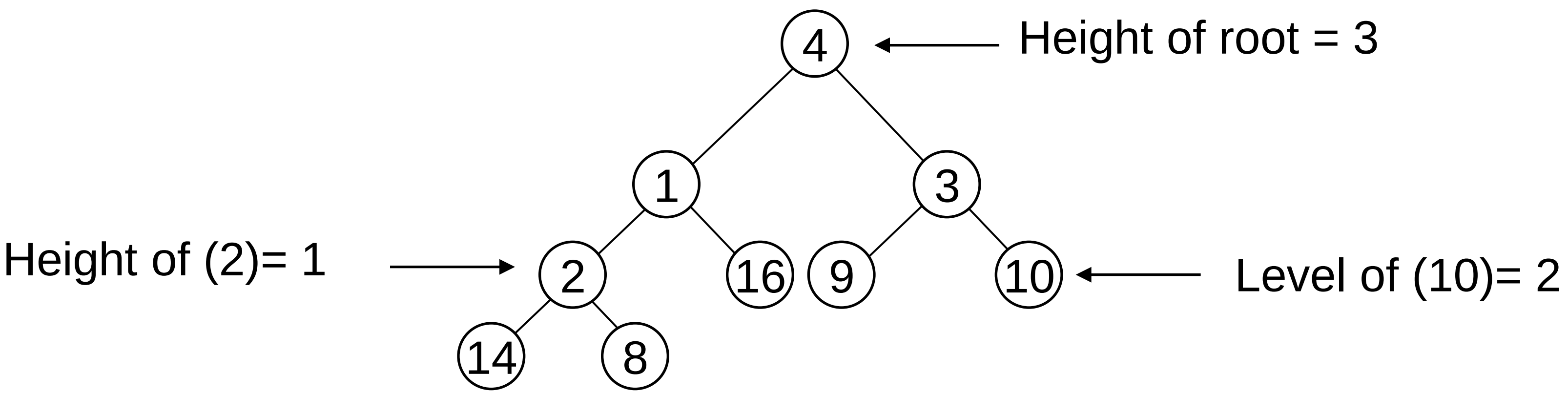
$\Rightarrow T(n) = \Theta(n^2)$

Heap Sort

- Combines the better attributes of merge sort and insertion sort.
- Like merge sort, running time is O(n lg n).
- • Like insertion sort, sorts in place.
- To manage information during the execution of an algorithm data structure (binary heap) is used, which has 2 properties:
- Shape property
- Heap property

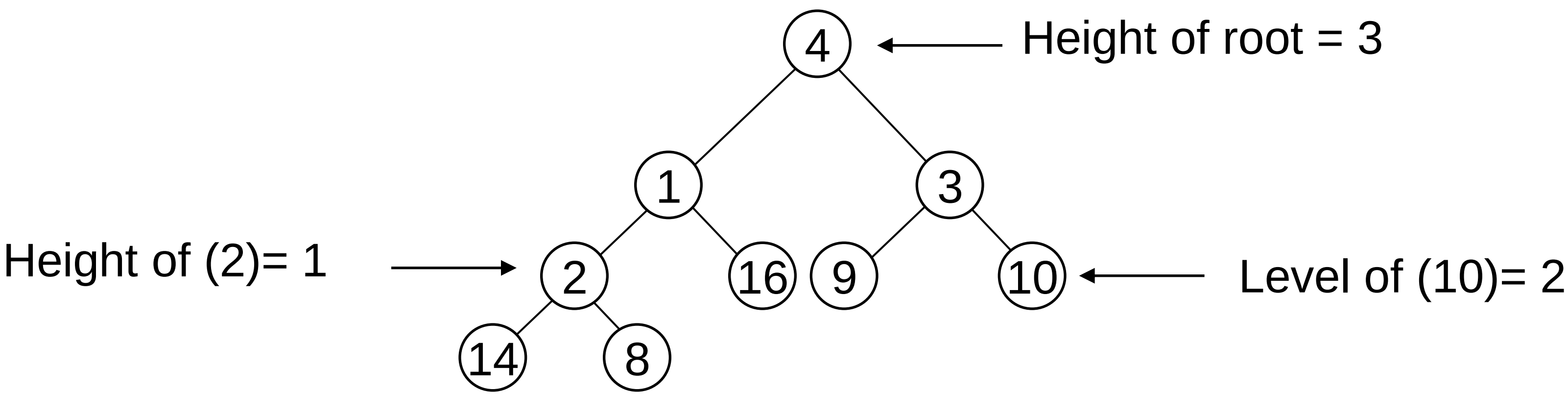
Tree Characteristics

- **Height** of a node = the number of edges on the longest simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node



Tree Charcteristics

- 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** $\lfloor lgn \rfloor$

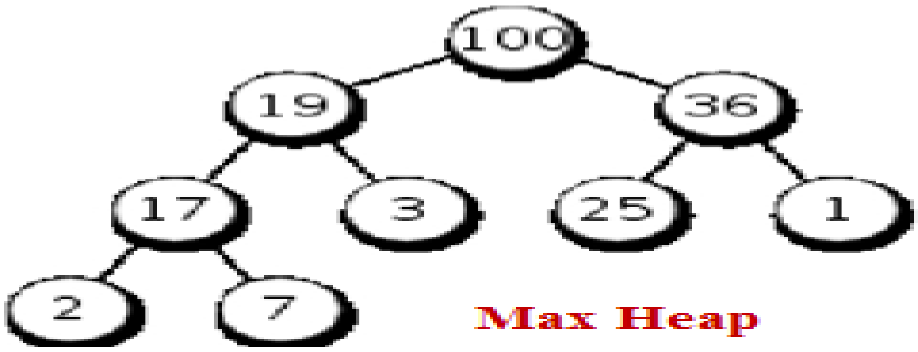


Max Heap

- **Max-heaps** have the *max-heap property:*
  - for all nodes i, excluding the root:

$$A[PARENT(i)] \geq A[i]$$

  - Largest element is stored at the root.
  - In any subtree, no values are larger than the value stored at subtree root.

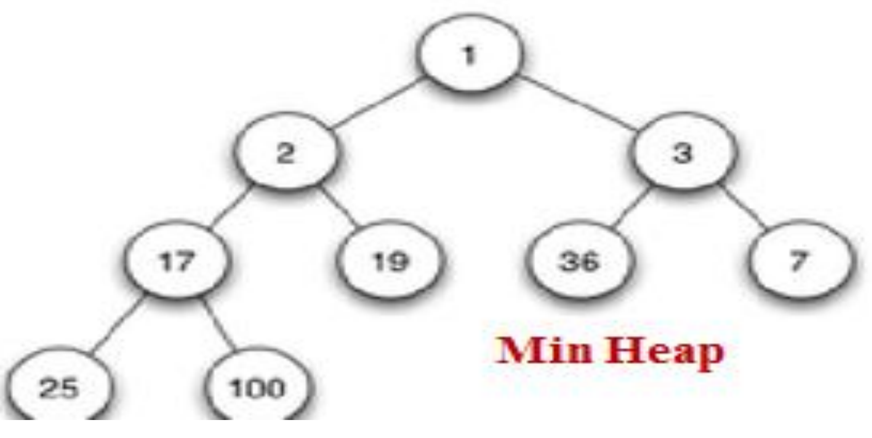


Min Heap

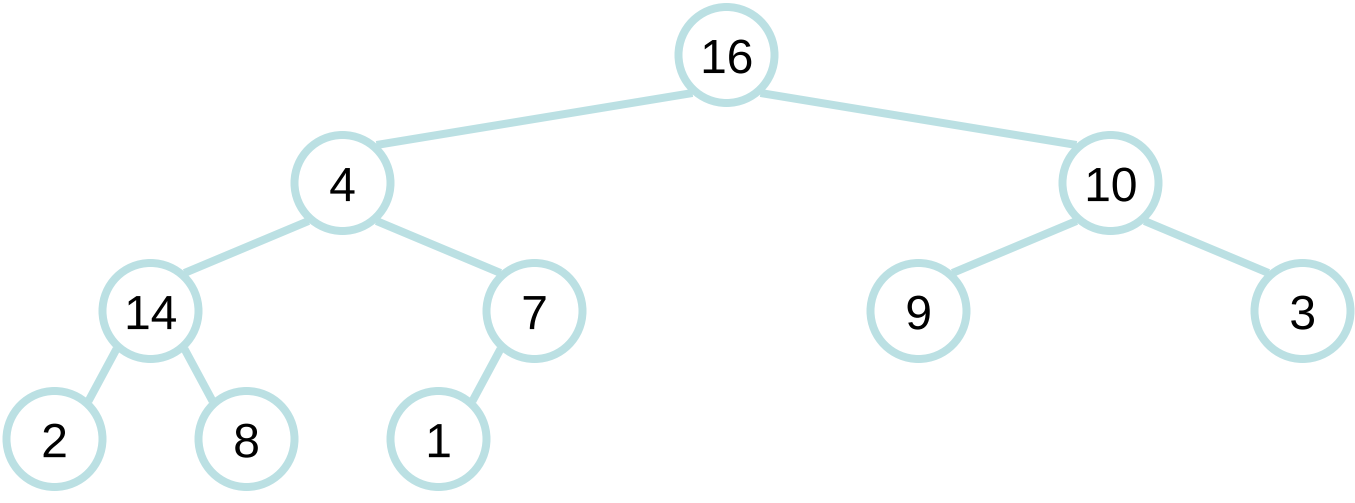
- **Min-heaps** have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[PARENT(i)] \leq A[i]$$

  - Smallest element is stored at the root.
  - In any subtree, no values are smaller than the value stored at subtree root.

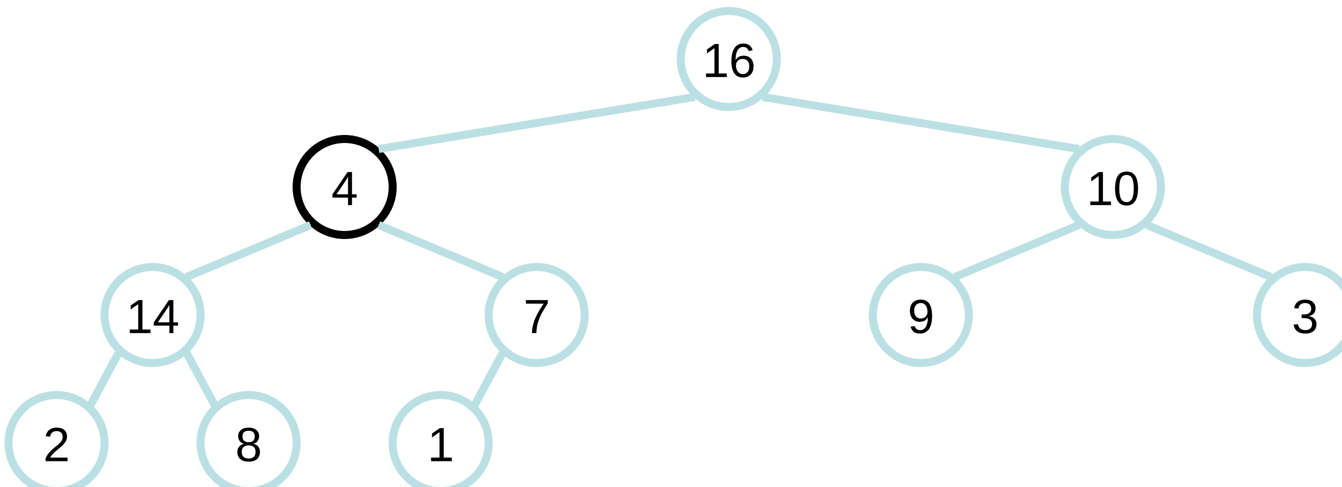


Heapify() Example



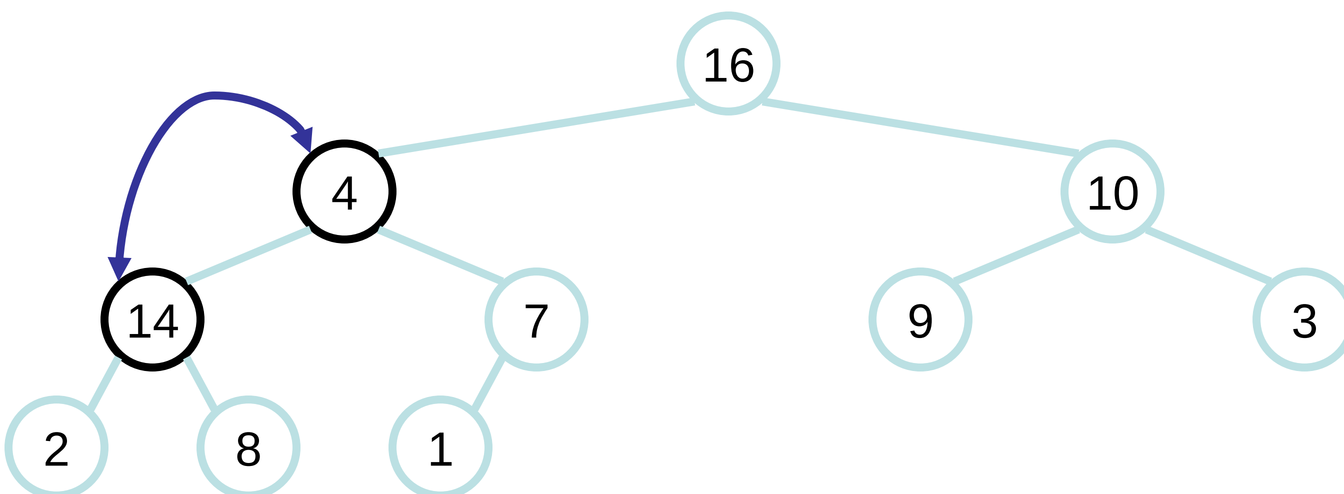
A =	16	4	10	14	7	9	3	2	8	1

Heapify() Example



A =	16	4	10	14	7	9	3	2	8	1

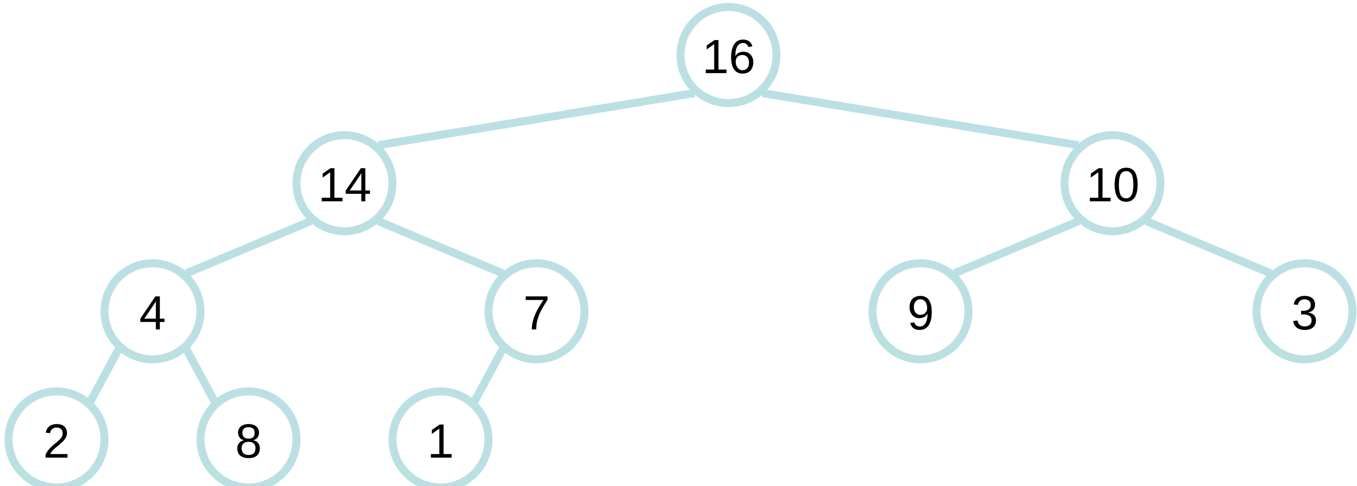
Heapify() Example



A =	16	4	10	14	7	9	3	2	8	1

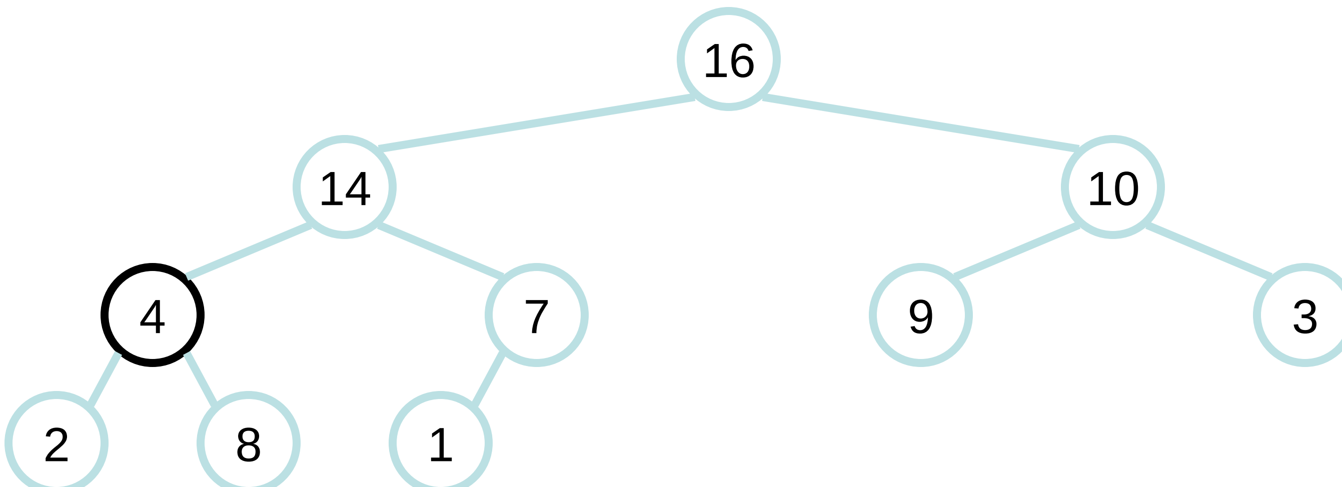
David Luebke

Heapify() Example



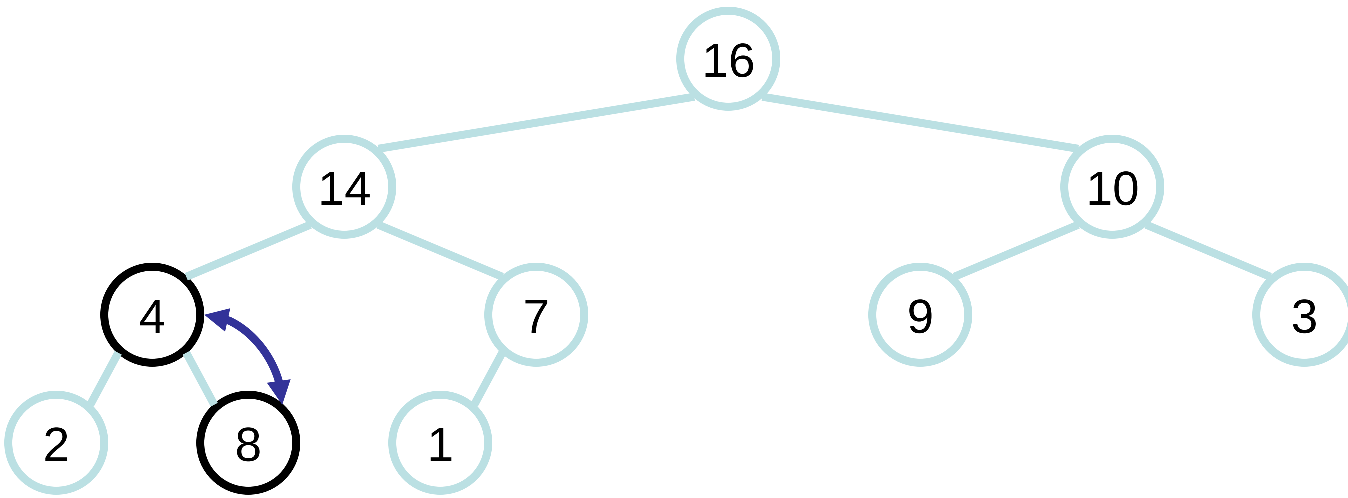
A =	16	14	10	4	7	9	3	2	8	1

Heapify() Example



A =										
	16	14	10	$\boxed{4}$	7	9	3	2	8	1

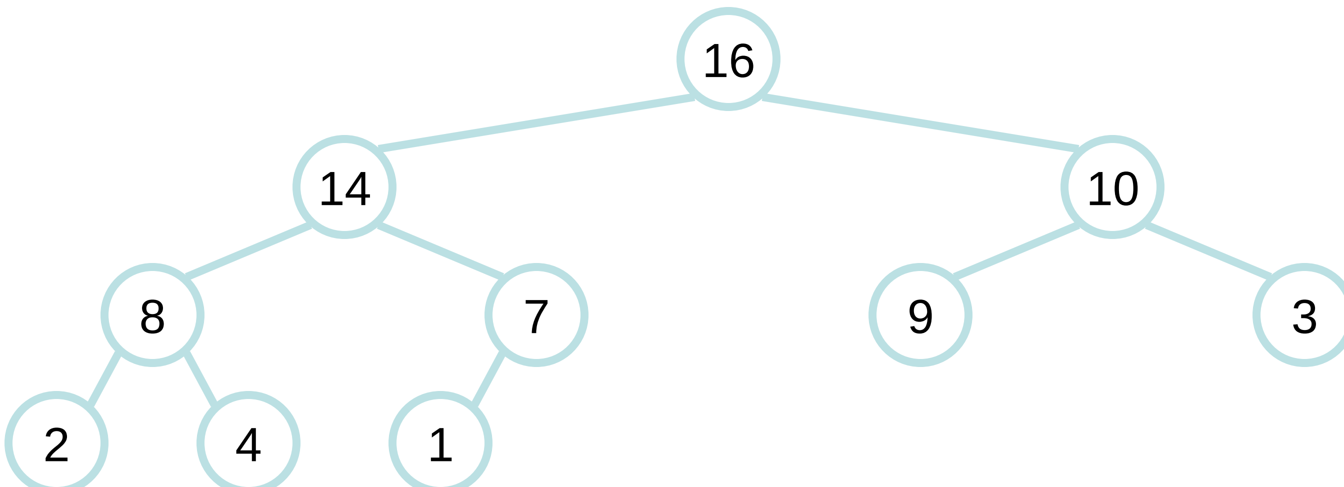
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

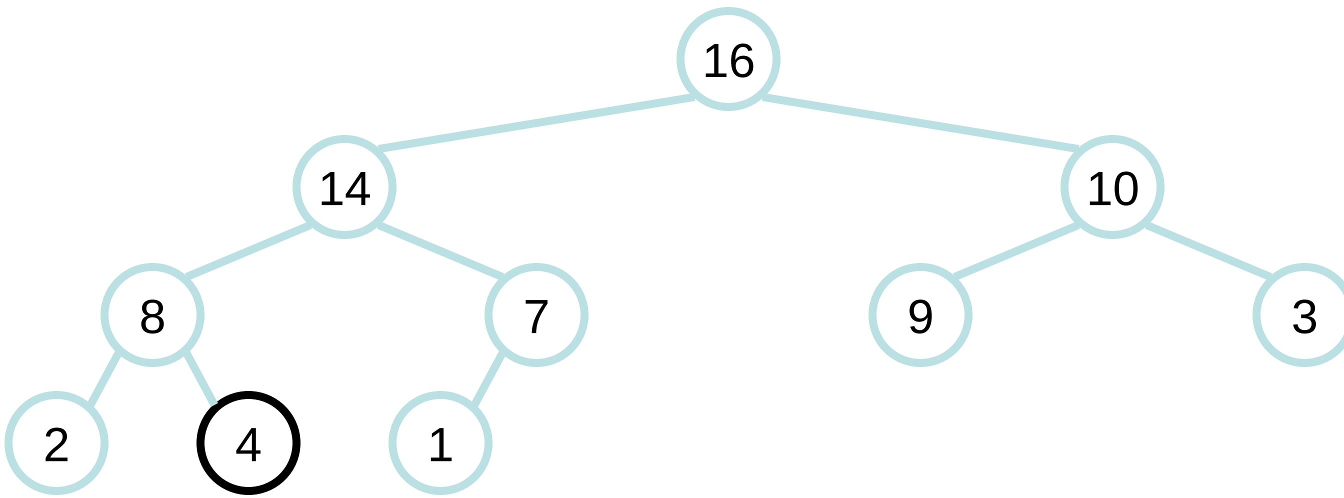
Heapify() Example



A =

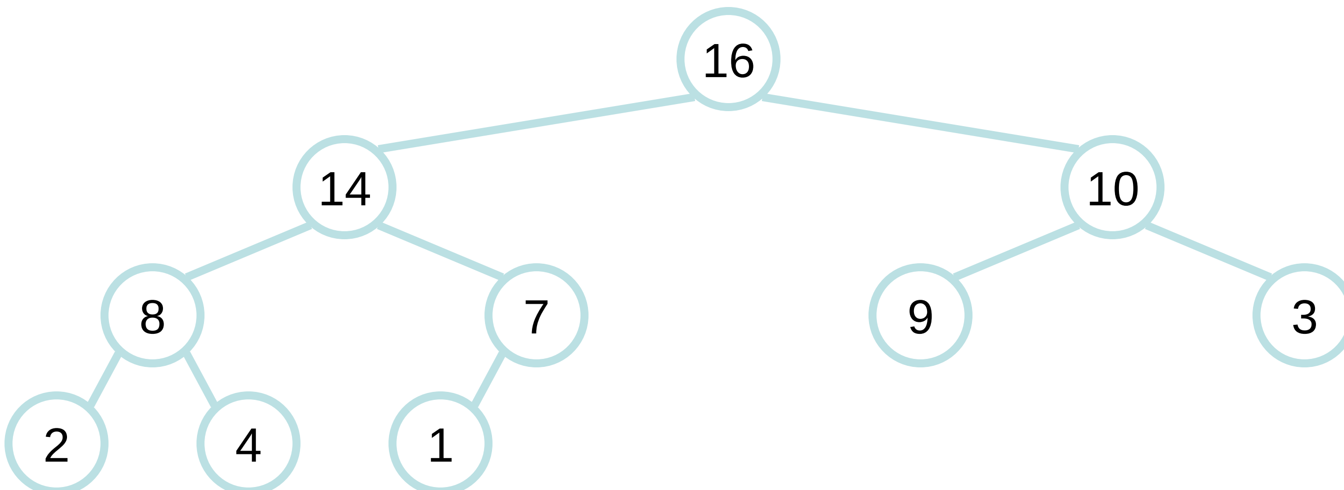
16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

Heapify() Example

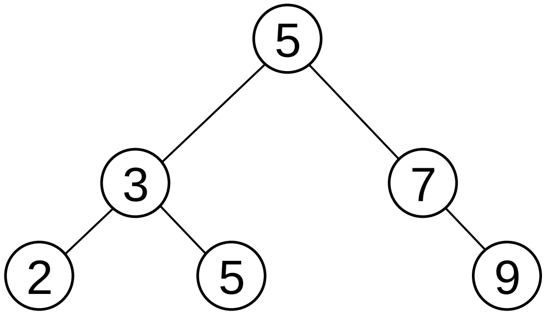


A =

16	14	10	8	7	9	3	2	4	1

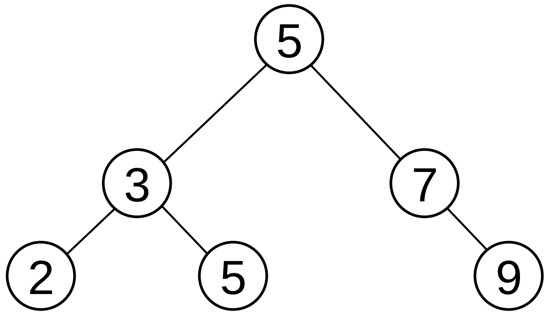
Binary Search Tree Property

- Binary search tree property:
  - If y is in left subtree of x,
    then key [y] ≤ key [x]
  - If y is in right subtree of x,
    then key [y] ≥ key [x]



Traversing a Binary Search Tree

- **Inorder** tree walk:
  - Root is printed between the values of its left and right subtrees: left, root, right
  - Keys are printed in sorted order
- **Preorder** tree walk:
  - root printed first: root, left, right
- **Postorder** tree walk:
  - root printed last: left, right, root



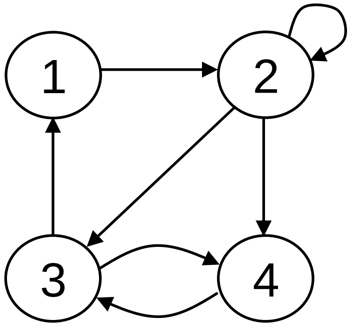
Inorder: 2 3 5 5 7 9

Preorder: 5 3 2 5 7 9

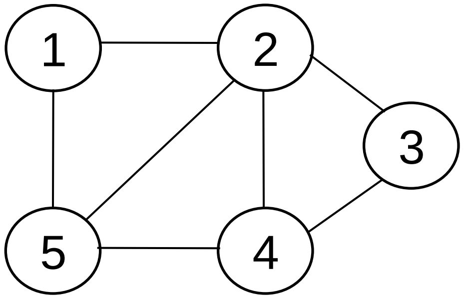
Postorder: 2 5 3 9 7 5

Adjacency Representation

- Sum of "lengths" of all adjacency lists
  - Directed graph: $|E|$
    - edge (**u**, **v**) appears only once (i.e., in the list of **u**)
  - Undirected graph: $2|E|$
    - edge (**u**, **v**) appears twice (i.e., in the lists of both **u** and **v**)

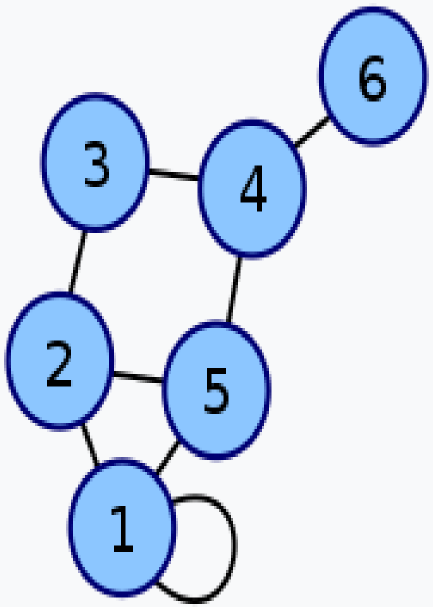


Directed graph



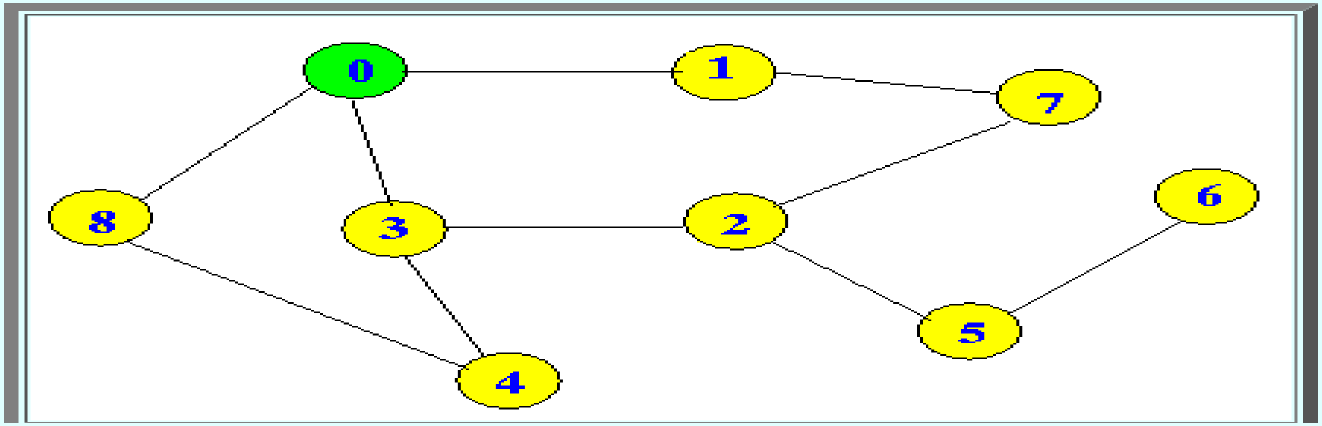
Undirected graph

Adjacency Representation

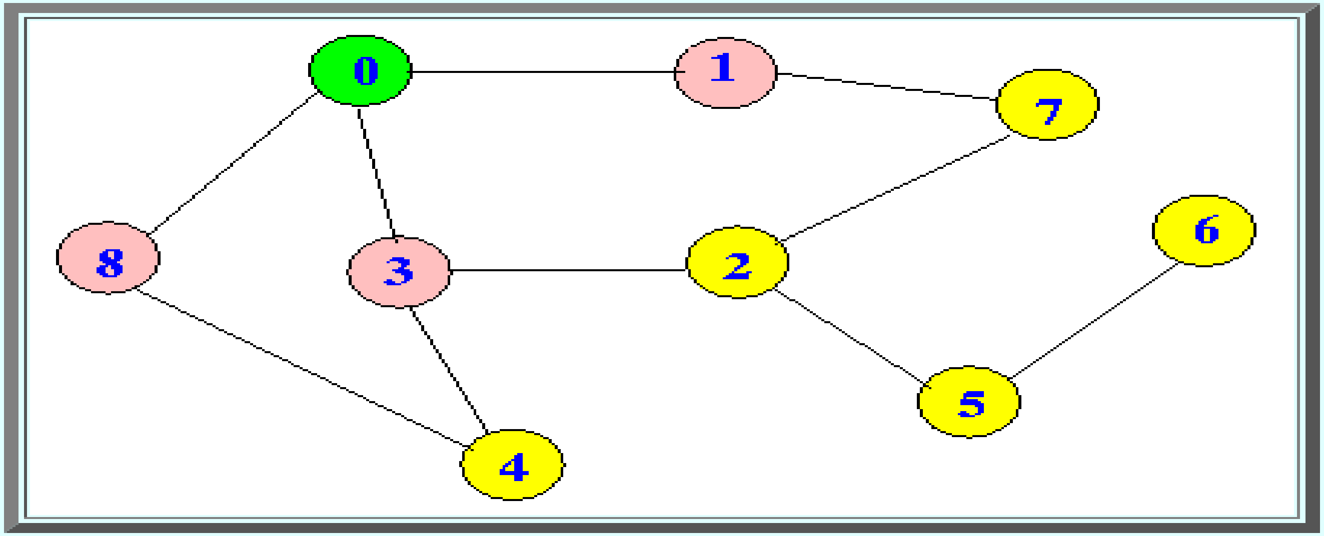
Labeled graph	Adjacency matrix
	$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ Coordinates are 1–6.

Breadth First Search

- Start at some node (e.g., **node 0**):

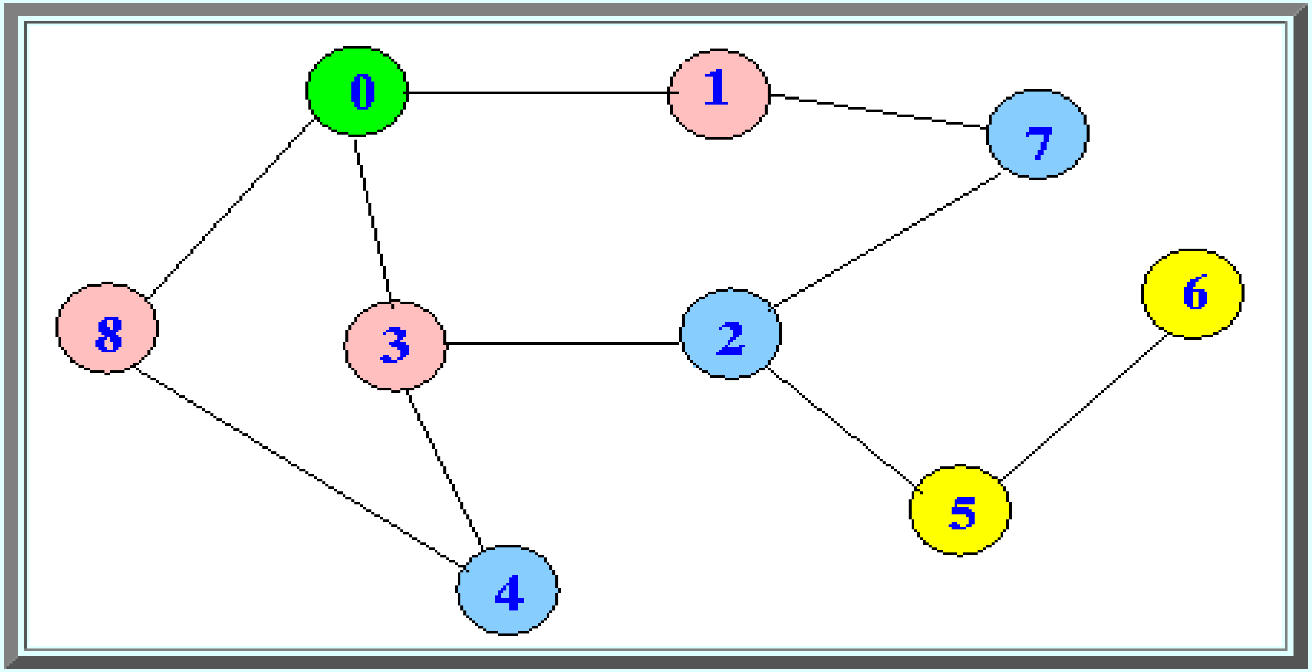


- Visit ***all* the neighbors** of **node 0** first:



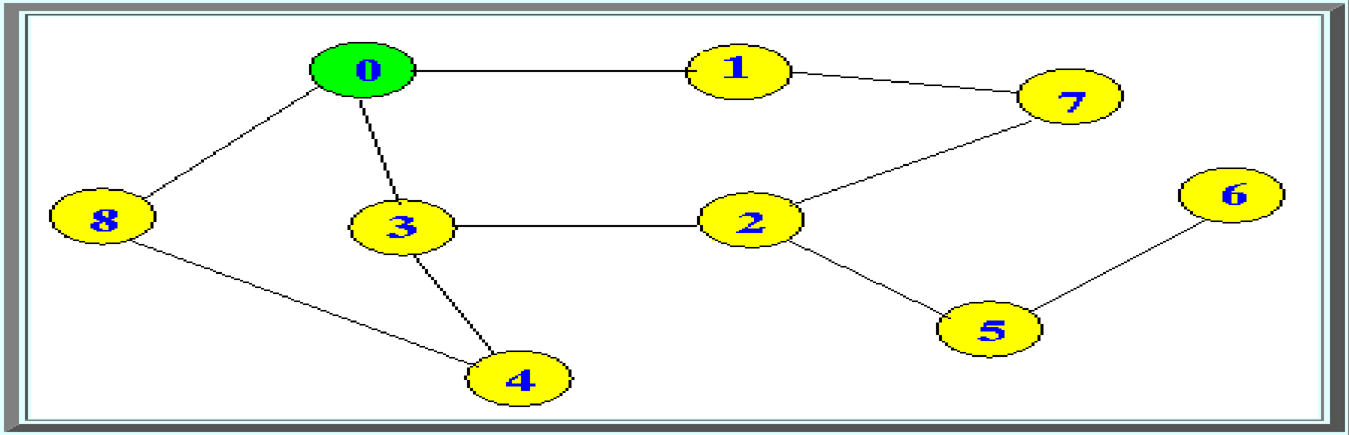
Breadth First Search

- **Then** visit the ***neighbors'* neighbors**:

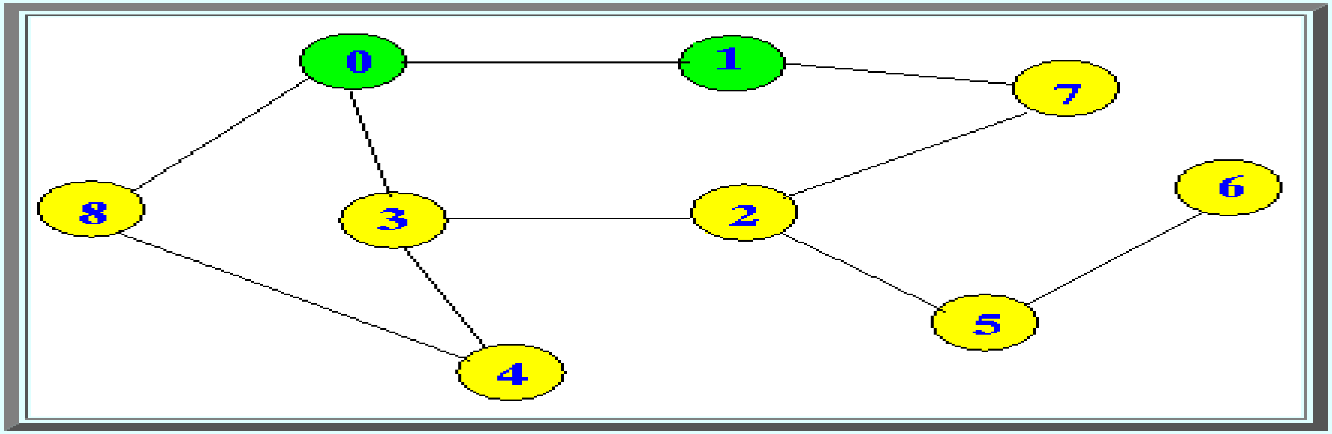


Depth First Search

- Start at some node (e.g., **node 0**):

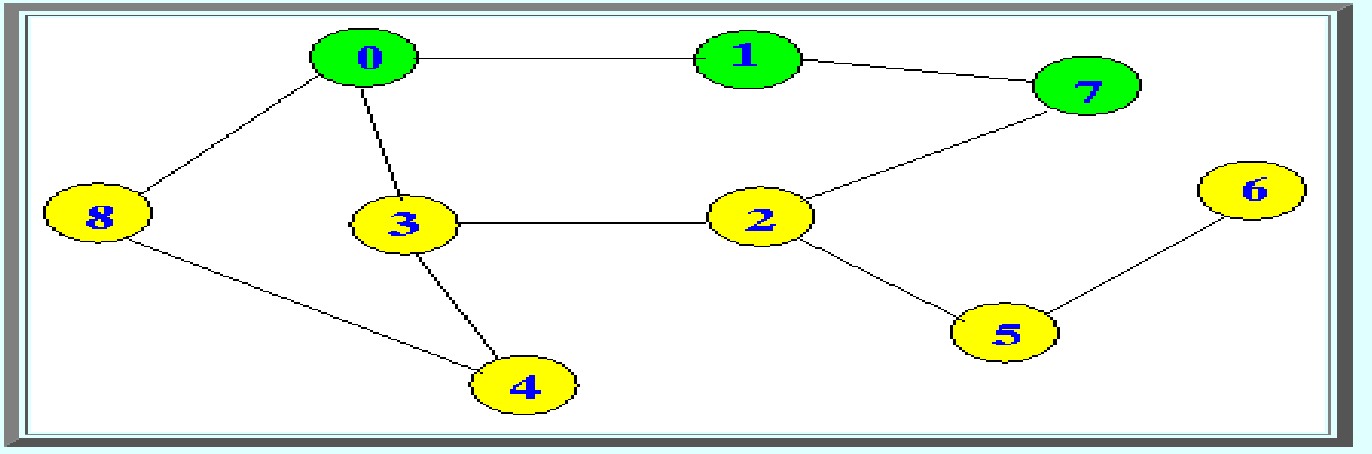


- Visit ***one* of the *unvisited* neighbors** of **node 0**:

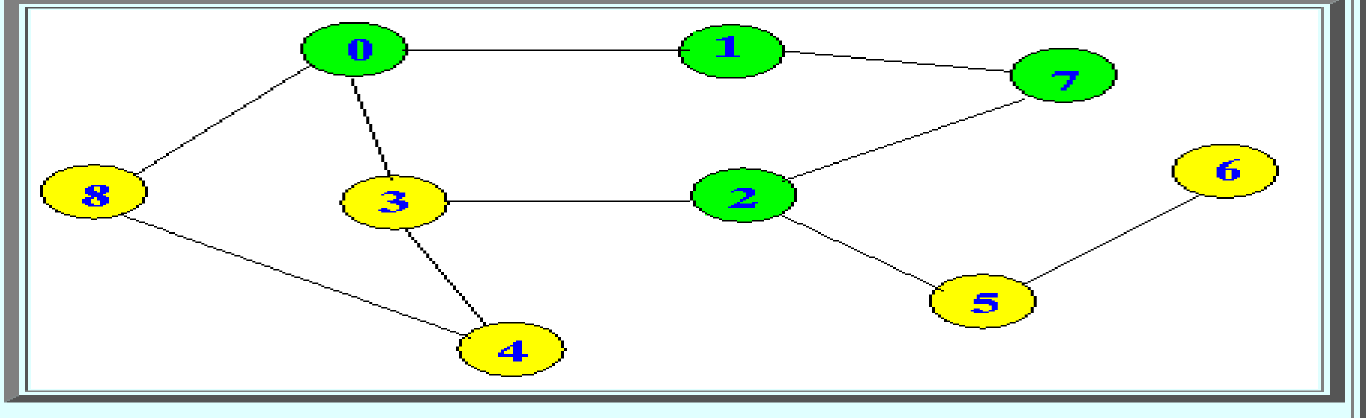


Depth First Search

- Then visit *one* of the *unvisited* neighbors of node 1:



- Then visit *one* of the *unvisited* neighbors of node 7:



- And so on

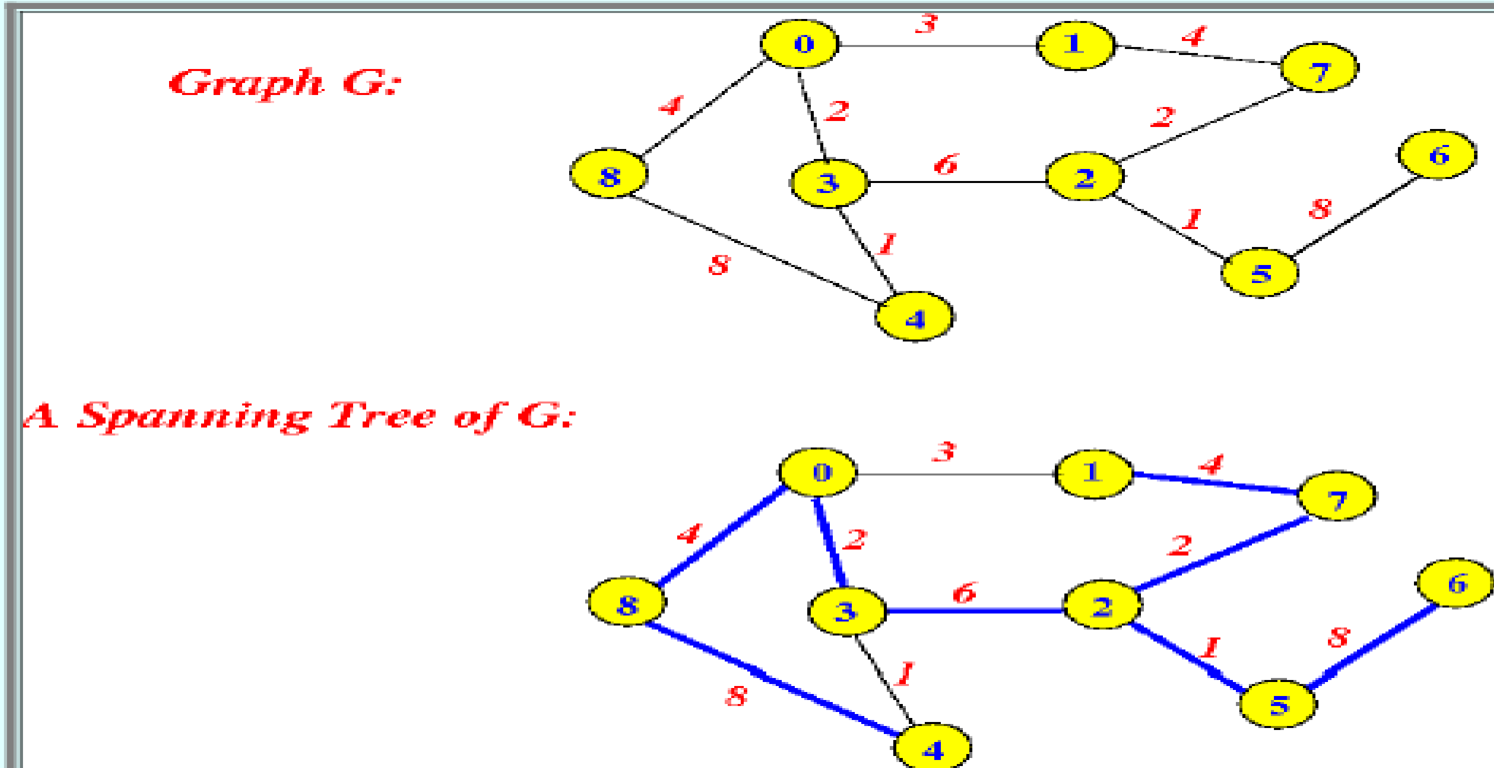
Summary

Basis for comparison	BFS	DFS
Basic	Vertex-based algorithm	Edge-based algorithm
Data structure used to store the nodes	Inefficient	Efficient
Structure of the constructed tree	Wide and short	Narrow and long
Traversing fashion	Oldest unvisited vertices are explored at first.	Vertices along the edge are explored in the beginning.
Optimality	Optimal for finding the shortest distance, not in cost.	Not optimal
Running Time	**O(V + E)**, with V being the number of vertices and E the number of edges in the graph.	**O(V + E)**, with V being the number of vertices and E the number of edges in the graph.

Minimum Spanning Tree

- Spanning Tree of a graph G = a tree (= no cycles) that includes:
  - *All* vertices of the graph G
  - some or all of the edges of the graph G

Example:



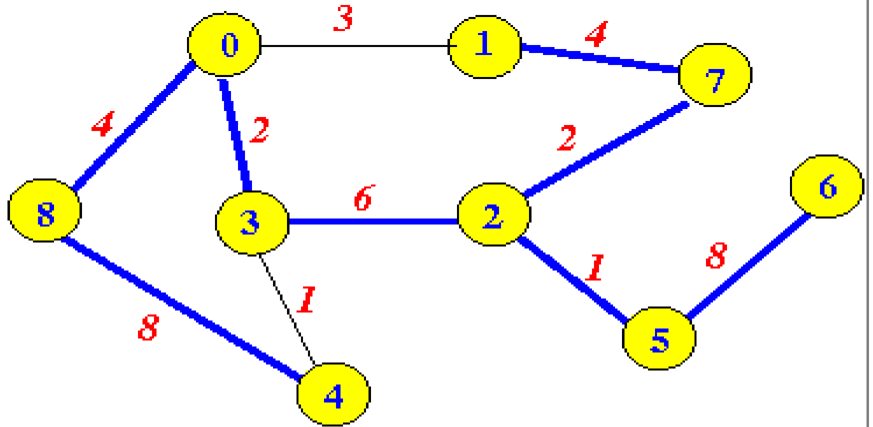
The *edges* of the Spanning tree is depicted in blue

Minimum Spanning Tree

- **Cost of a Tree**
  - ***Definition:*** **cost of a tree**
    - **Cost of a Tree** = the **sum of the cost** of **all the edges** in the **tree**

Example:

A Spanning Tree of G:

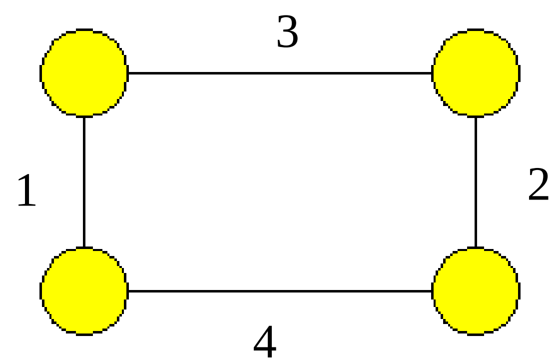


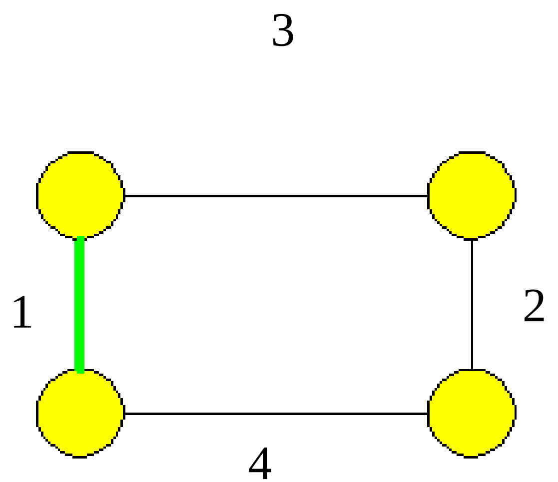
Cost of Tree = 8 + 4 + 2 + 3 + 6 + 4 + 2 + 1 + 8
= 38

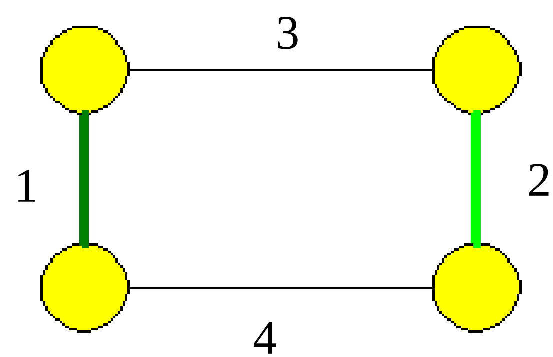
Kruskal's Algorithm

- Take the smallest edge that does not induce a
- cycle, and insert it into our subgraph.
- Do this until all nodes are connected
- A naive way to make sure an edge does not
- induce a cycle is by using DFS or BFS from one
- of the edge's vertices, and seeing if we reach
- the other. If we do, adding that edge would
- create a cycle.

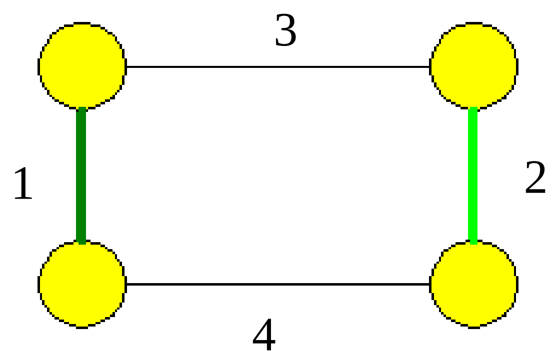
Kruskal's- Example

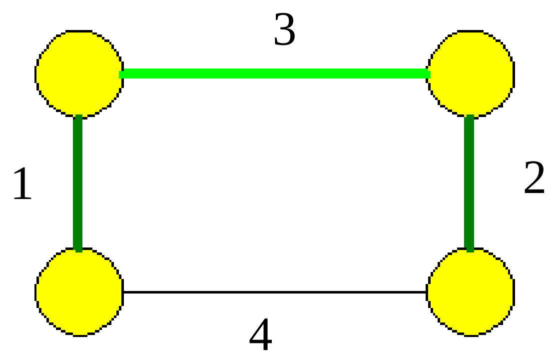






Kruskal's- Example

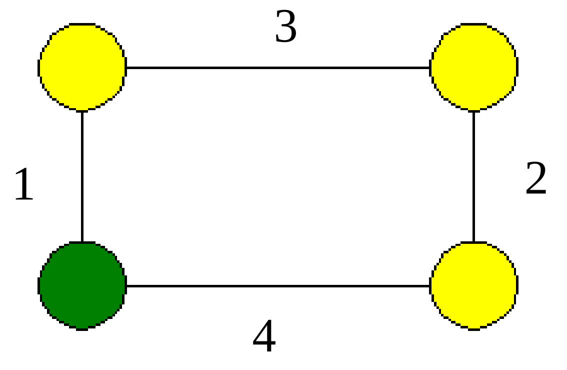


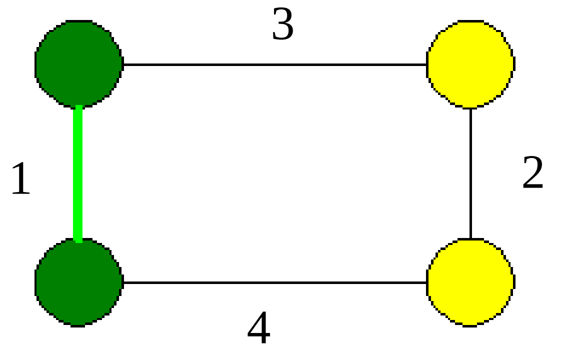


Prim's Algorithm

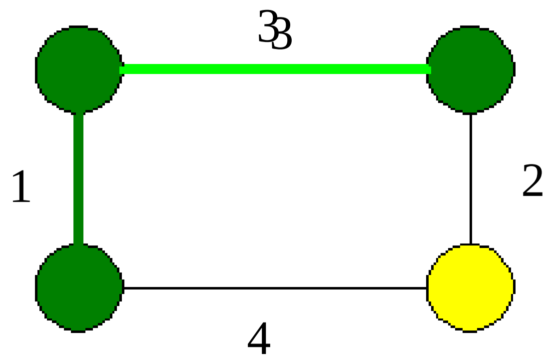
- Mark a vertex.
- while we still don't have a spanning tree
- Take the least edge that is between a marked
- and unmarked vertex
- mark the unmarked vertex

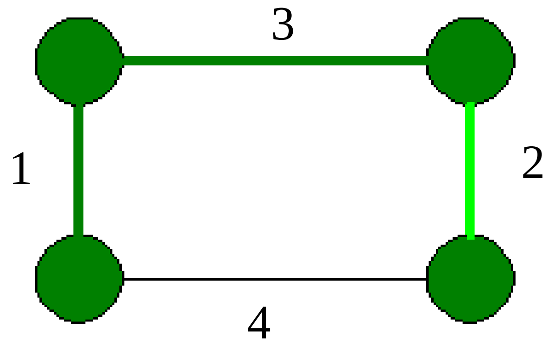
Prim's- Example





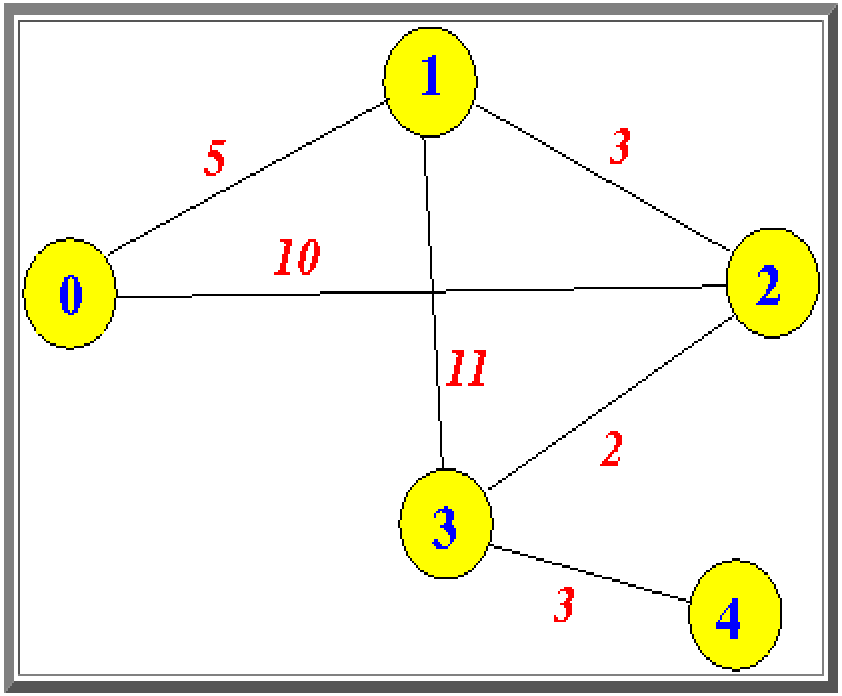
Prim's- Example





Shortest Path Tree

- Minimum cost paths from a vertex to all other vertices
  - Consider:

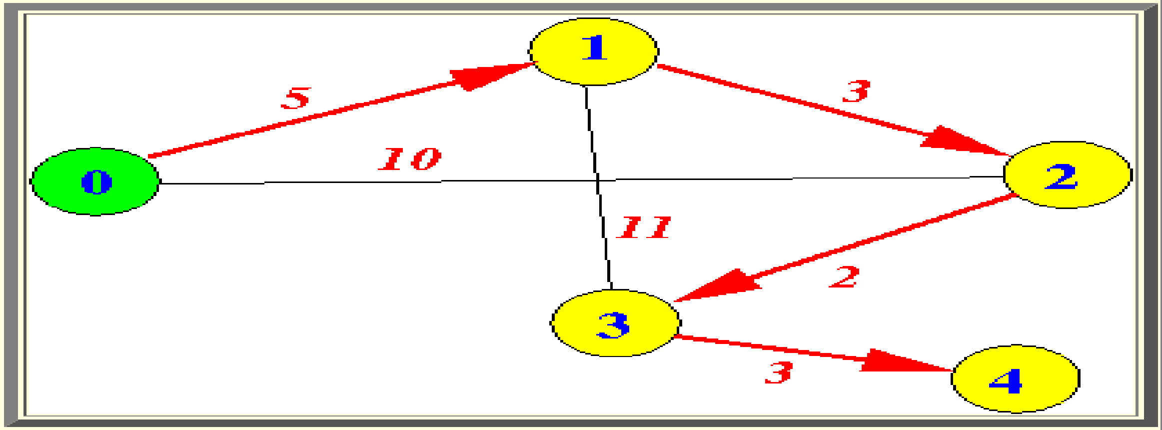


Problem:

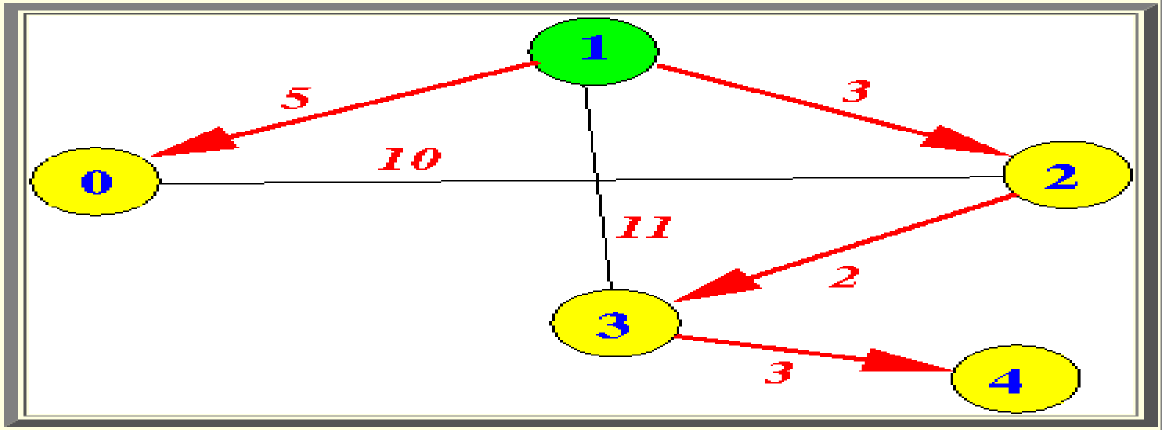
- Compute the minimum cost paths from a node (e.g., node *1*) to *all* other node in the graph

Shortest Path Tree

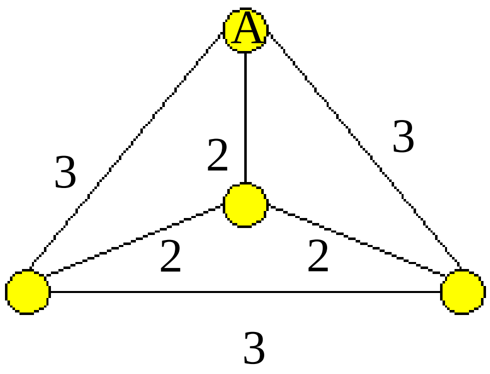
- Shortest paths from **node *0*** to ***all*** other nodes:

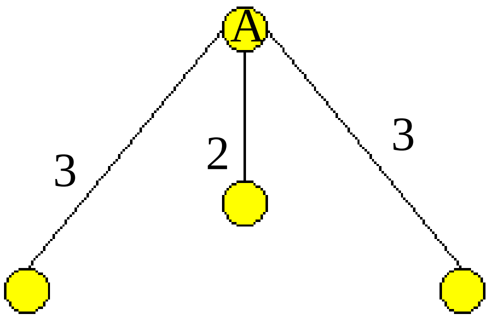


- Shortest paths from **node *1*** to ***all*** other nodes:

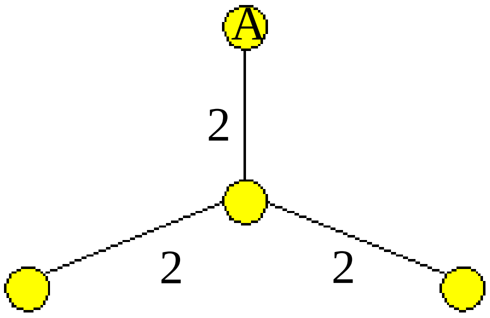


MST vs SPT





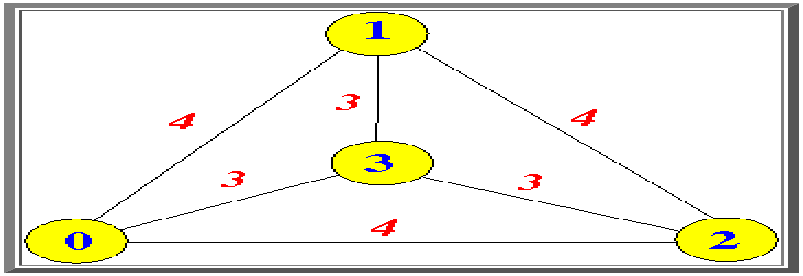
Shortest path tree from A
Total Cost: 8
Total Cost of Paths from A:
3+3+2=8



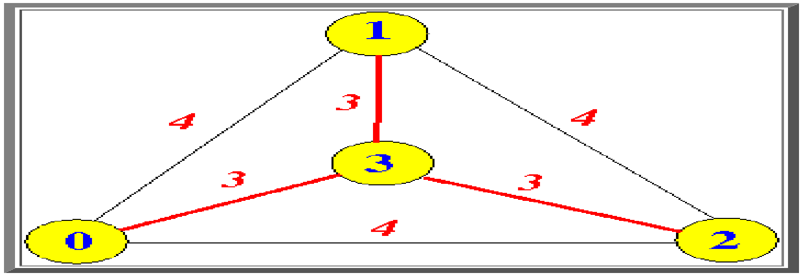
Minimum Spanning tree
Total Cost: 6
Total of Paths from A:
2+4+4=10

MST vs. SPT

- Shortest path is *not* the same as Minimum cost spanning tree
  - **Consider** the following **graph**:

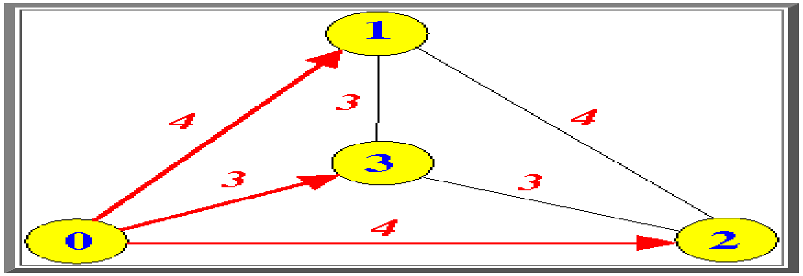


  - The ***Minimum coast*** **spanning tree** of this **graph** is:



(The **MST** is given with **red edges**)

  - The **shortest path** from **node *0*** to ***all*** **other nodes** is:



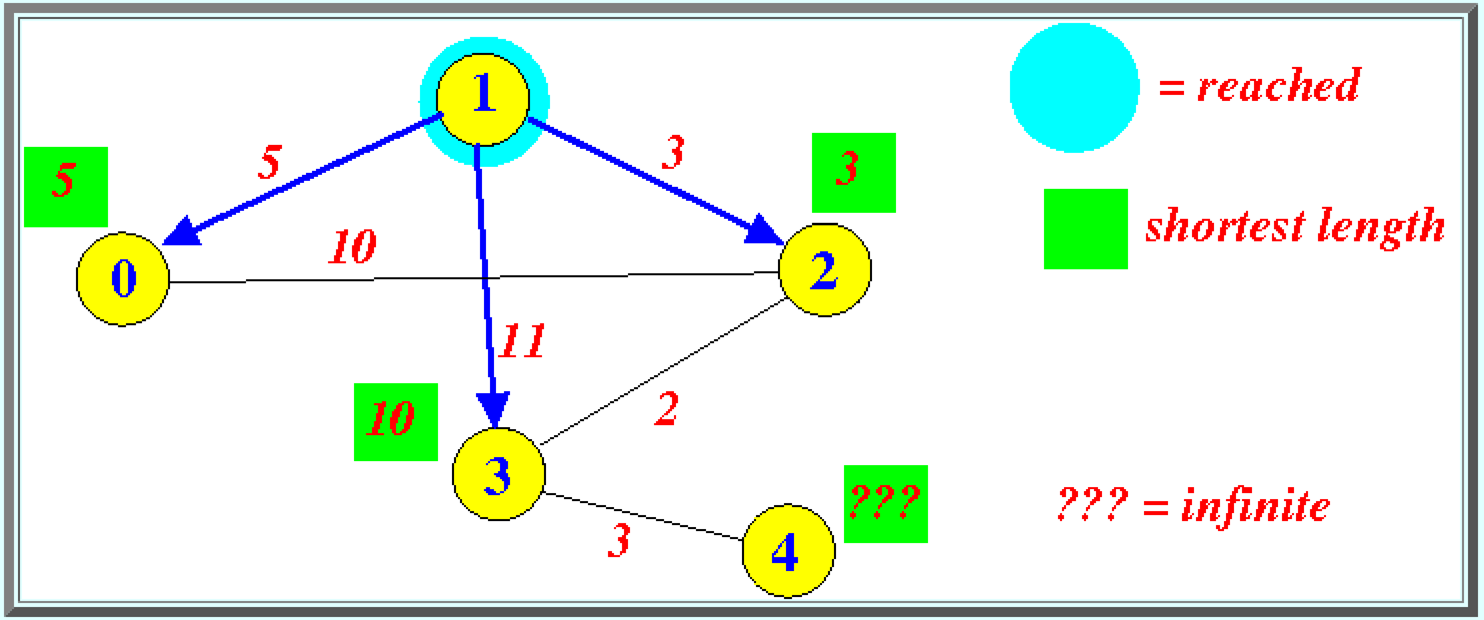
You can see that the **shortest path** uses ***different*** **edges** than the **minimum cost spanning tree** !!!

Dijkstra's Algorithm

- ***Initilaization***:

  - **Label** the **source node (node *1*)** as ***reached***
  - **Label** all the **other nodes** as ***unreached***
  - Use **each edge** from the **source node (node *1*)** as the **shortest path** to **nodes** that you can **reach *immediately***

Result:



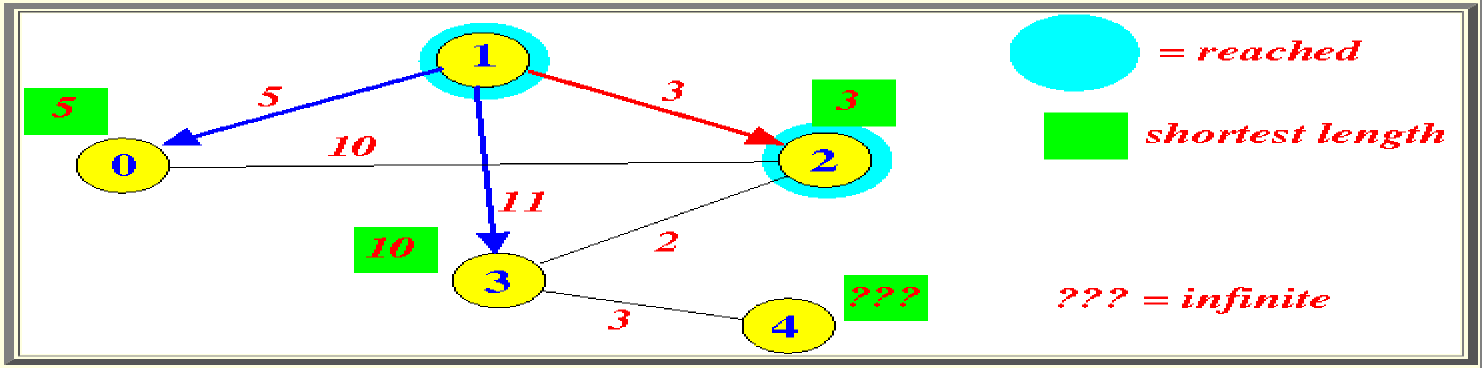
(The ***reached* node(s)** have a ***cyan* circle** as **marking**.

The ***unreached* nodes** are **unmarked**)

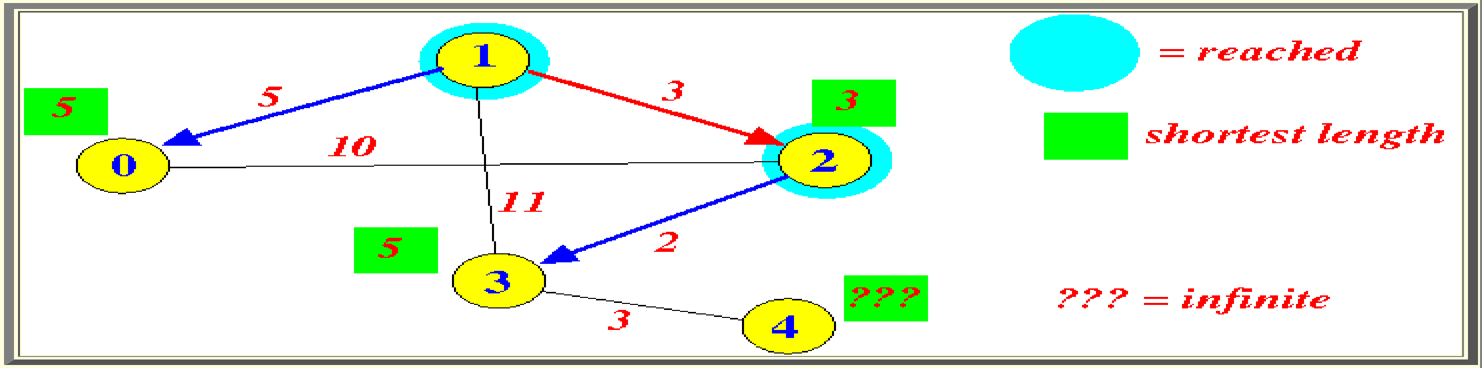
Dijkstra's Algorithm

- *Iteration (1)*:
  - Find the *unreached* **node** *m* that has the *shortest* **path** from the **source node**:
    - *m* = 2 (with **path length** = 3)
  - **Added** the **edge** you used to reach *m* to the **shortest path**

    Label the **node** *m* as *reached*



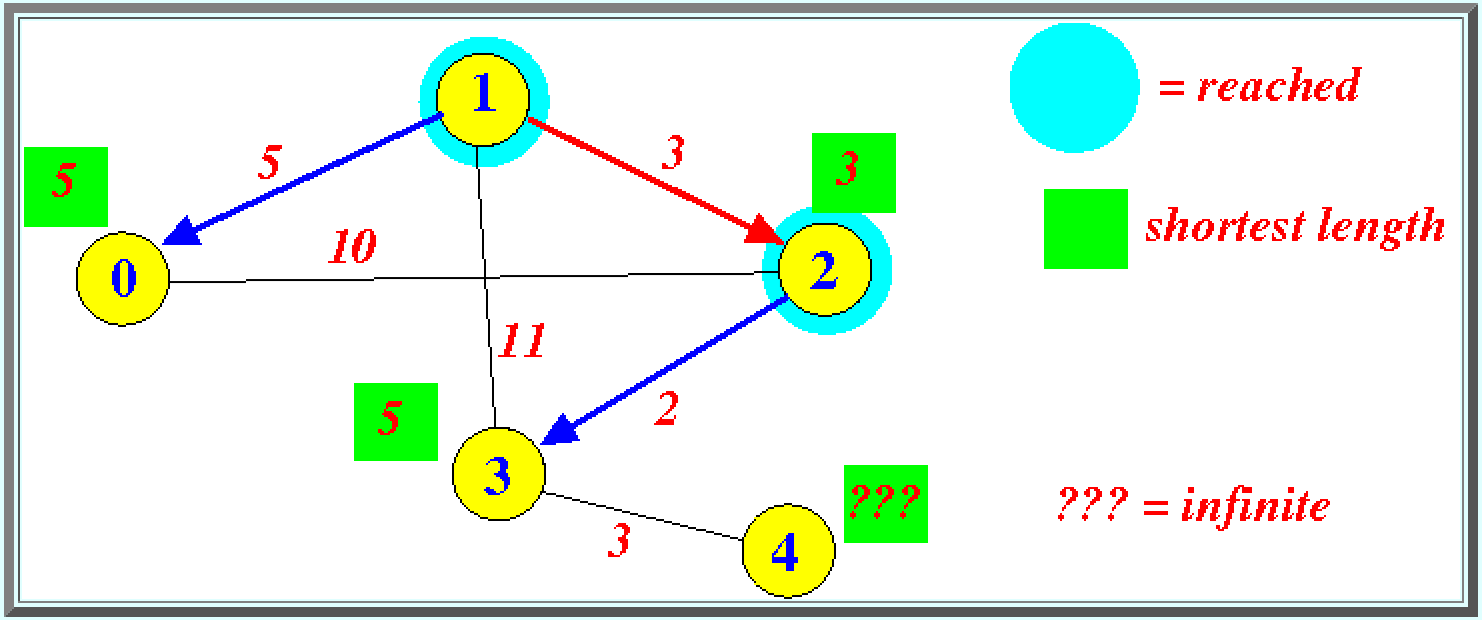
  - *Recompute* the *shortest paths* of **nodes** that can be **reached** via *m* (if possible)



We can **reach** the **node** *3* via *node 2* through a *shorter* **path** !!!

Dijkstra's Algorithm

Result at the end of the iteration:



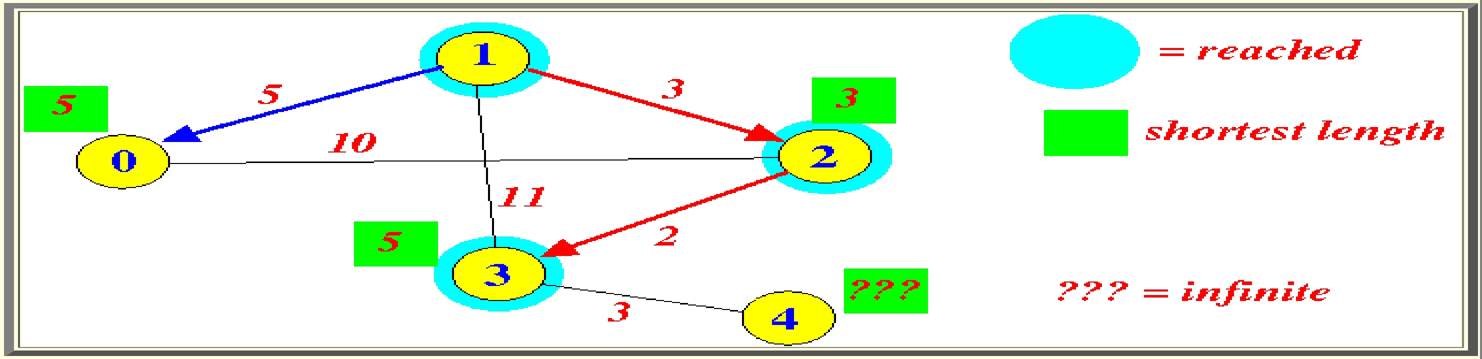
Dijkstra's Algorithm

- ***Iteration (2)***:
  - Find the ***unreached* node *m*** that has the ***shortest* path** from the **source node**:
    - *m* = 3 (with **path length** = 5)

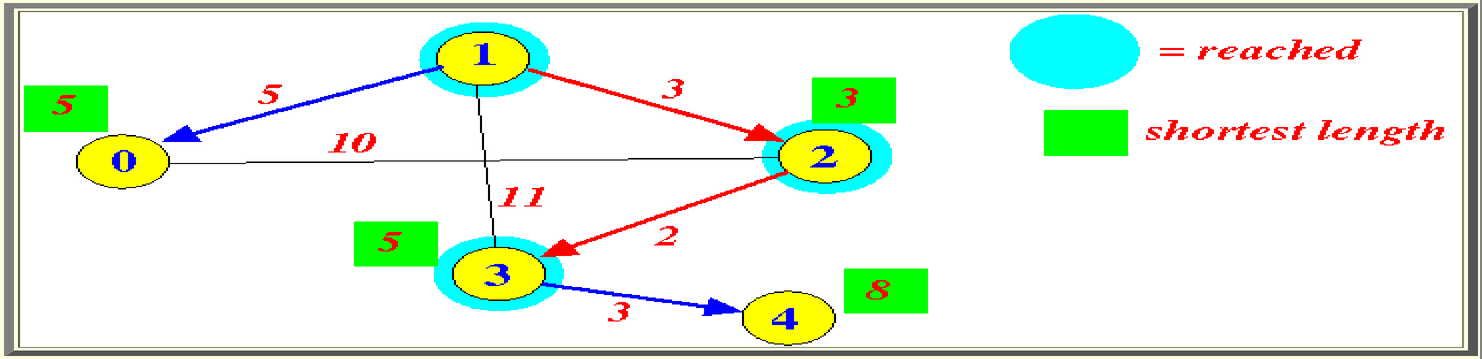
      (Node **0** will work also, but I picked node **3**)

  - **Added** the **edge** you used to reach ***m*** to the **shortest path**

    Label the **node *m*** as ***reached***

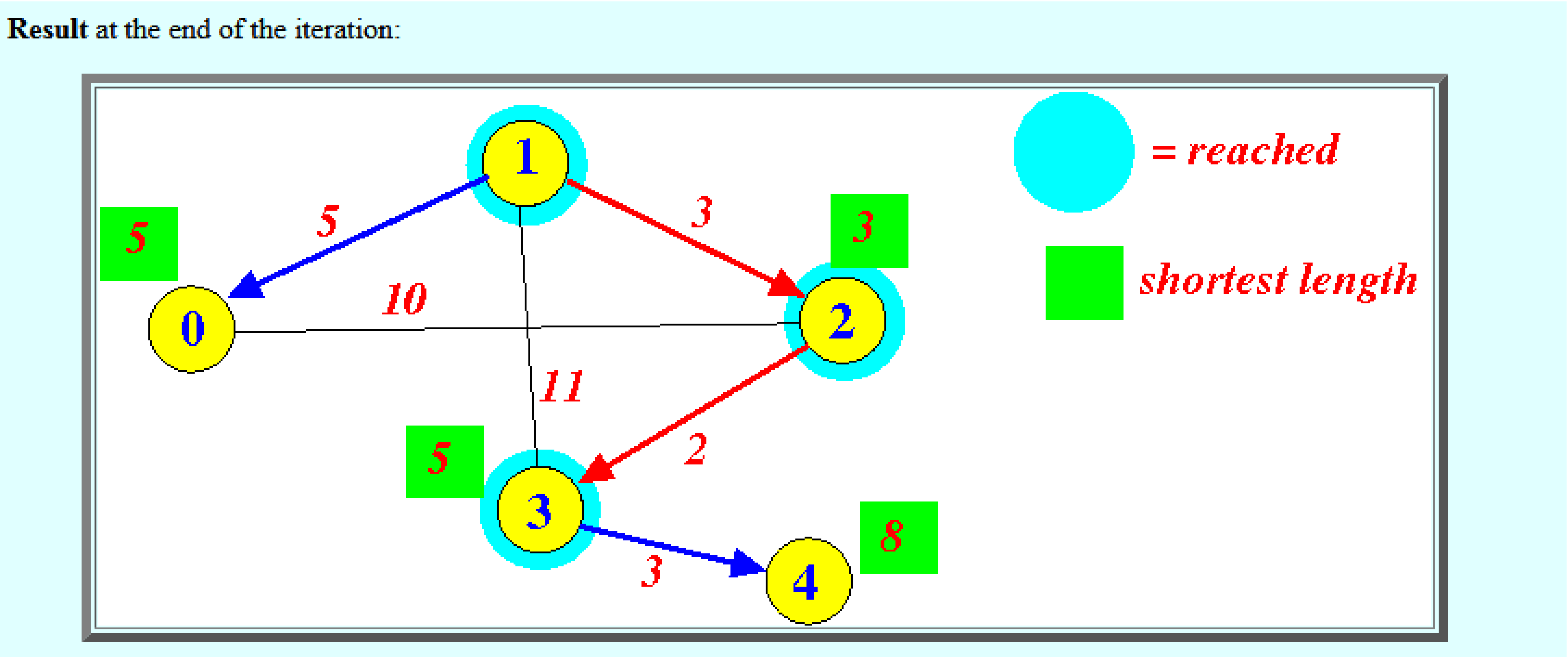


  - ***Recompute*** the ***shortest paths*** of **nodes** that can be **reached** via ***m*** (if possible)



We can **reach** the **node *4*** via ***node 3*** through a **path** of length ***8*** !!!

Dijkstra's Algorithm

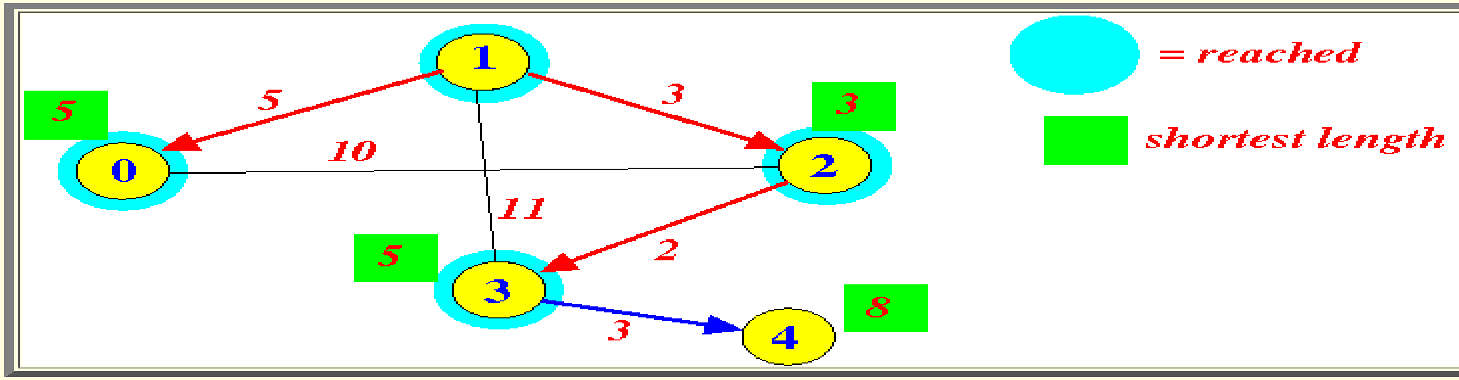


Dijkstra's Algorithm

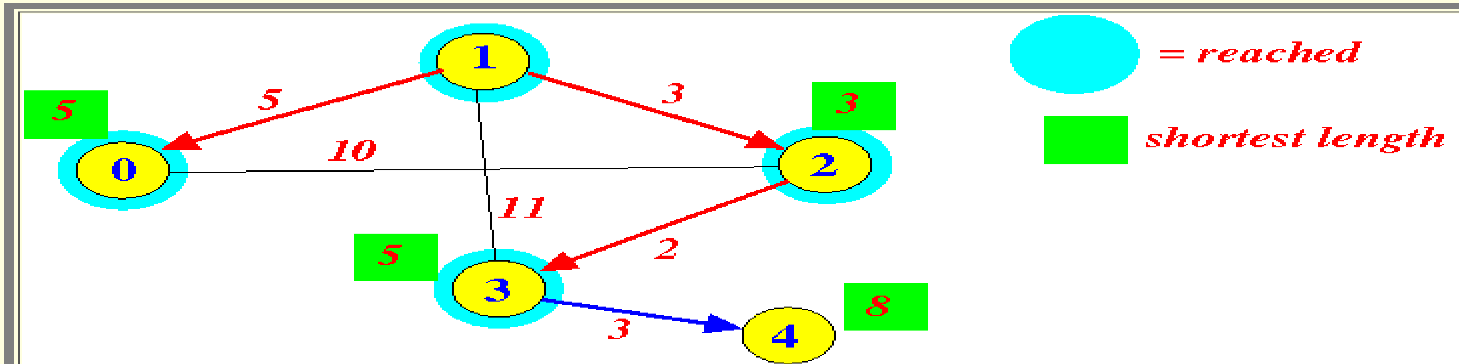
- ***Iteration (3)***:
  - Find the ***unreached* node *m*** that has the ***shortest* path** from the **source node**:
    - $m = 0$ (with **path length** = 5)

  - **Added** the **edge** you used to reach m to the **shortest path**

    Label the **node** m as ***reached***



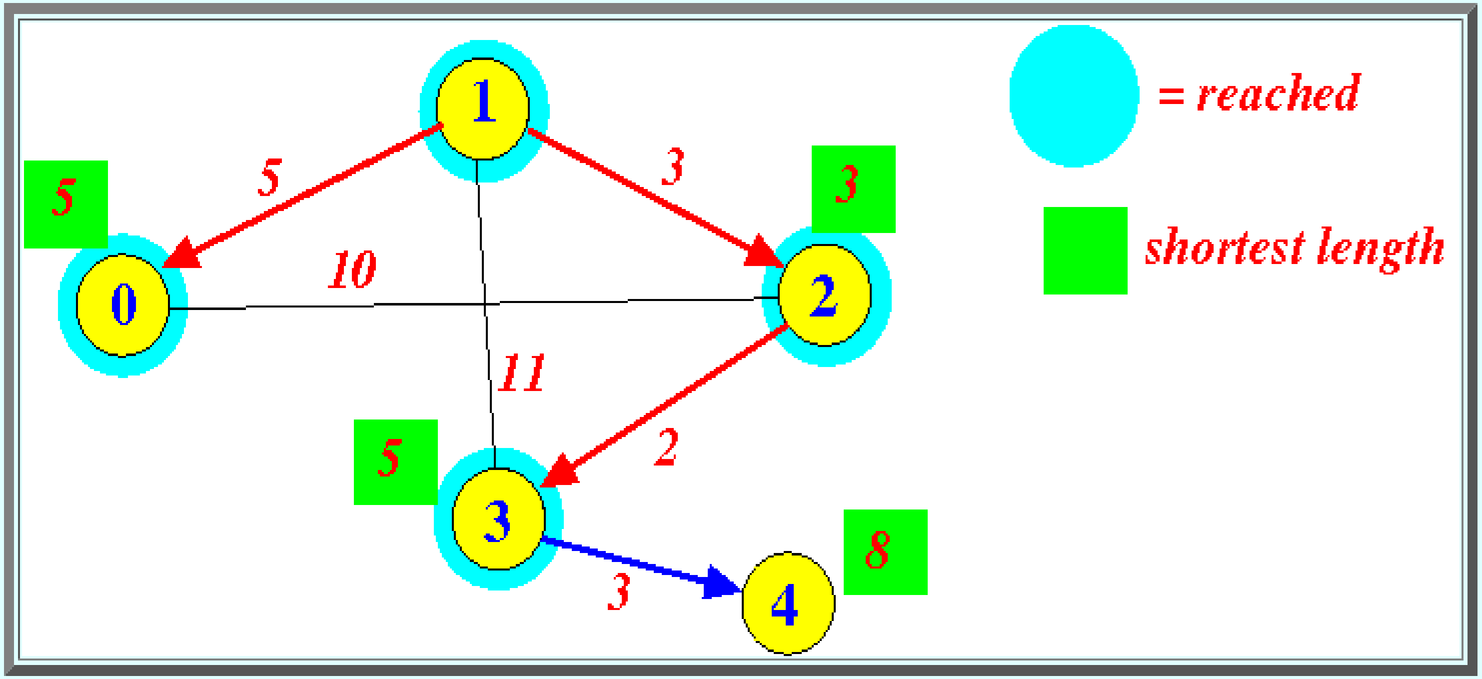
  - ***Recompute*** the ***shortest paths*** of **nodes** that can be **reached** via m (if possible)



There are ***no* improvements**....

Dijkstra's Algorithm

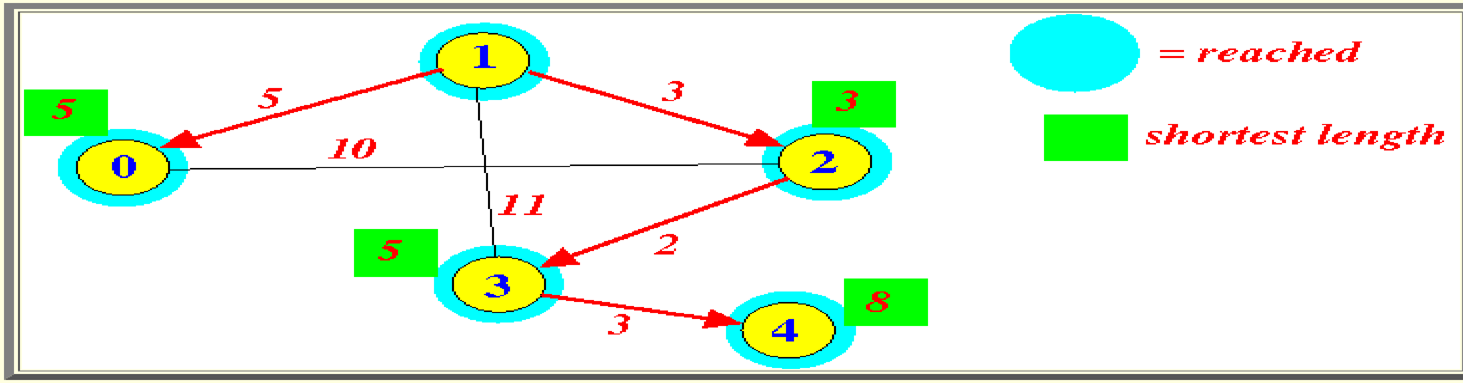
Result at the end of the iteration:



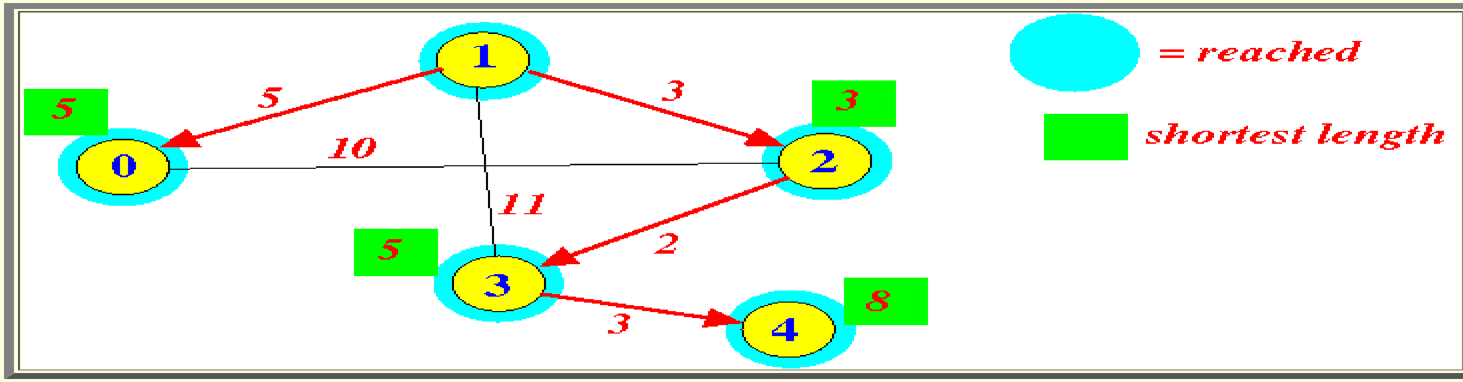
Dijkstra's Algorithm

- ***Iteration (4):***
  - Find the ***unreached* node *m*** that has the ***shortest* path** from the **source node**:
    - ***m* = 4** (with **path length = 8**)
  - **Added** the **edge** you used to reach ***m*** to the **shortest path**

    Label the **node *m*** as ***reached***



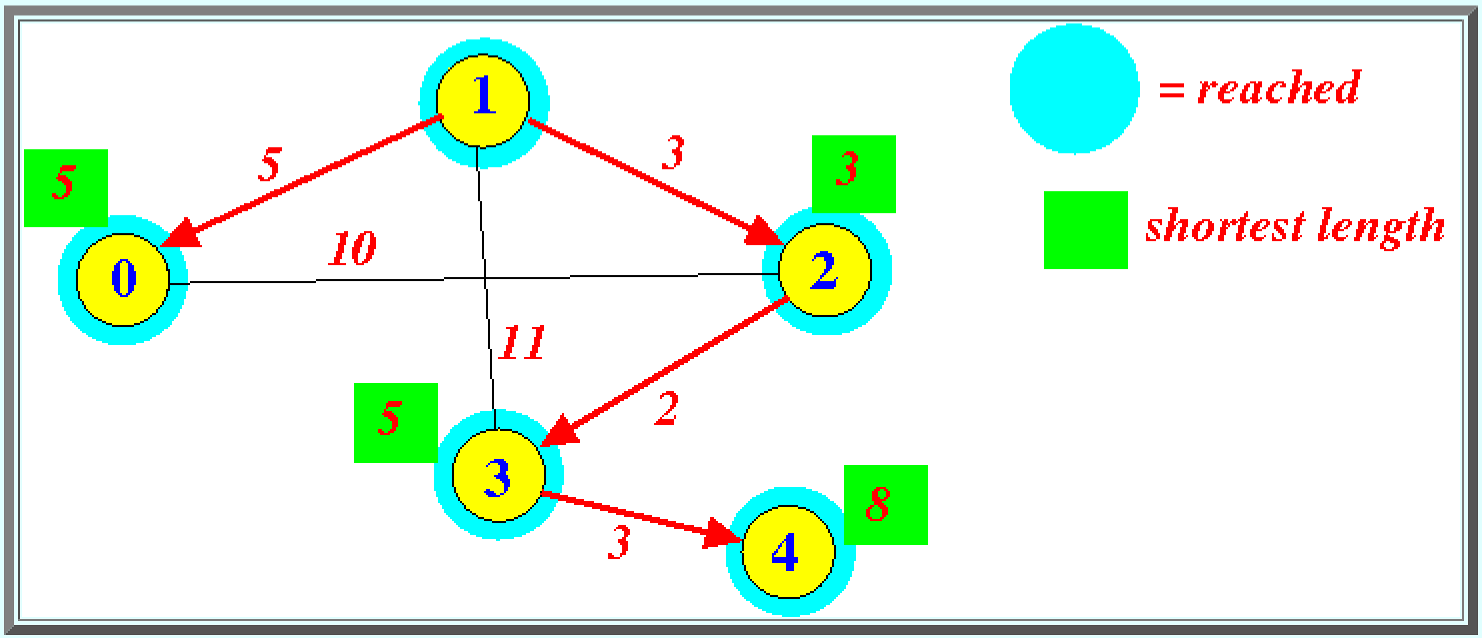
  - ***Recompute*** the ***shortest paths*** of **nodes** that can be **reached** via ***m*** (if possible)



    Again, ***no* improvements**.... (because there are **no more *unreached* nodes** !!!)

Dijkstra's Algorithm

Result at the end of the iteration:



Done !!!

Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

Brute-Force Solution

- For every subsequence of X, check whether it's a subsequence of Y
- There are 2^m subsequences of X to check
- Each subsequence takes $\Theta(n)$ time to check
  - scan Y for first letter, from there scan for second, and so on
- Running time: $\Theta(n \cdot 2^m)$

LCS

What is Longest Common Subsequence: A longest subsequence is a sequence that appears in the same relative order, but not necessarily contiguous(not substring) in both the string.

Example:

```
String A = "acbaed";
String B = "abcadf";
```

String A	a	c	b	a	e	d
String B	a	b	c	a	d	f

```
Longest Common Subsequence(LCS):     acad, Length: 4
```

LCS

Start comparing strings in reverse order one character at a time.

Now we have 2 cases –

1. Both characters are same
    1. add 1 to the result and remove the last character from both the strings and make recursive call to the modified strings.
2. Both characters are different
    1. Remove the last character of String 1 and make a recursive call and remove the last character from String 2 and make a recursive and then return the max from returns of both recursive calls. see example below

LCS- Example

```
Case 1:

String A: "ABCD", String B: "AEBD"

LCS("ABCD", "AEBD") = 1 + LCS("ABC", "AEB")

Case 2:

String A: "ABCDE", String B: "AEBDF"

LCS("ABCDE", "AEBDF") = Max(LCS("ABCDE", "AEBD"), LCS("ABCD", "AEBDF"))
```

0-1 Knapsack

In 0/1 Knapsack Problem,

- As the name suggests, items are indivisible i.e. we can not take the fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using dynamic programming approach.

Steps for solving 0/1 Knapsack Problem using Dynamic Programming Approach-

Consider we are given-

- A knapsack of weight capacity ‘w’
- ‘n’ number of items each having some weight and value

0-1 Knapsack

Step-01:

- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown-

	0	1	2	3		W
0	0	0	0	0	......	0
1	0					
2	0					
	......					
n	0					

T-Table

0-1 Knapsack

Step-02:

- Start filling the table row wise top to bottom from left to right.
- Use the following formula-

$$T(i, j) = \max \{ T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i) \}$$

$T(i, j)$ = maximum value of the selected items if we can take items 1 to i and we have weight restrictions of j.

Step-03:

After filling the table completely, value of the last box represents the maximum possible value that be put in the knapsack.

Step-04:

To identify the items that must be put in the knapsack to obtain the maximum profit,

- Considering the last column of the table, start scanning the entries from bottom to top.
- If an entry is encountered whose value is not same as the value which is stored in the entry immediately above it, then mark the row label of that entry.
- After scanning all the entries, the marked labels represent the items that must be put in the knapsack.