FISF130020: Introduction to Computer Science

Lecture 6: Algorithm II

Hui Xu

xuh@fudan.edu.cn

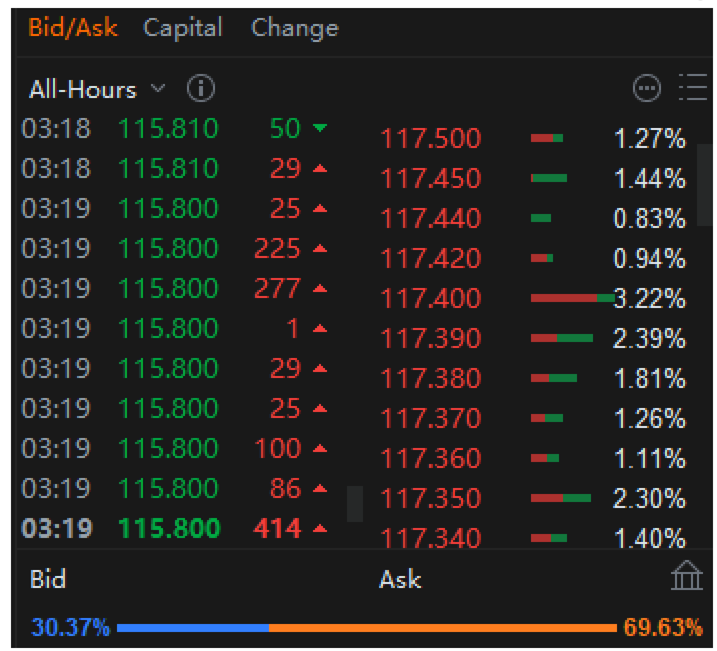
Outline

- 1. Binary Search Tree
- 2. Graph Analysis
- 3. In-class Practice

1. Binary Search Tree

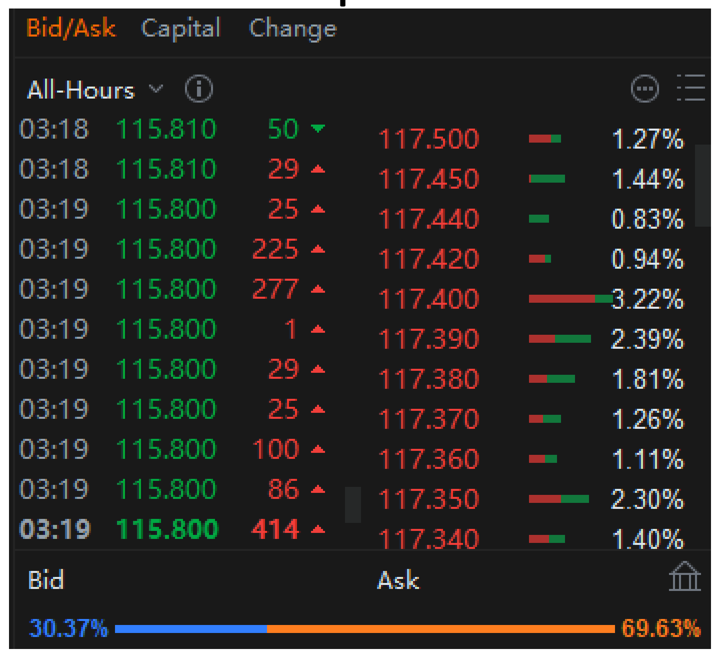
Problem

- The stock exchange problem involves handling the concurrent arrival of ask (sell) and bid (buy) orders for various stocks.
  - Ask order: The seller specifies the minimum price they are willing to sell.
  - Bid order: The buyer specifies the maximum price they are willing to pay.
- The goal is to design a server or system that can efficiently manage these orders in real time, ensuring that trades are executed according to predefined rules.



Principle of Order Matching

- Orders are matched according to the price-time priority:
  - Price priority: Higher bid prices have precedence over lower ones, and lower ask prices have precedence over higher ones.
  - Time priority: If multiple orders have the same price, the one that was received earlier takes precedence.
- A trade is executed if the bid price is greater than or equal to the ask price.



Problem Specification

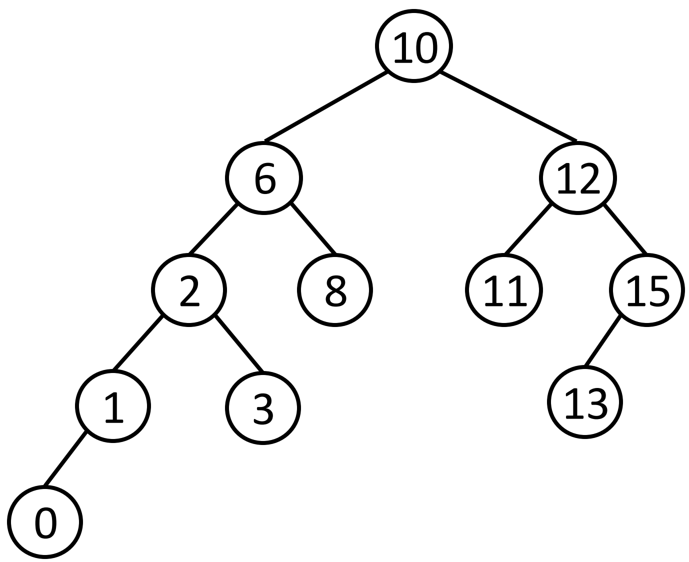
- **Input:**
  - A stream of ask orders and bid orders arriving concurrently.
  - Each order specifies the quantity of shares, and the price per share.
- **Output:**
  - The matched orders result in a trade, updating the remaining unmatched orders in the order book.
  - A record of each trade transaction, including the matched price and volume, should be maintained.
- System Requirements: Minimal latency in order processing.

Which Data Structure Can We Use?

- **Array**? The cost of insertion/deletion is high.
- **List**? The cost of search is high.
- **Queue**? We may use a time-priority queue to manage ask/bid orders of the same price.
- **HashMap**?
- **Tree**? We may use a binary search tree to manage ask/bid orders of different prices.

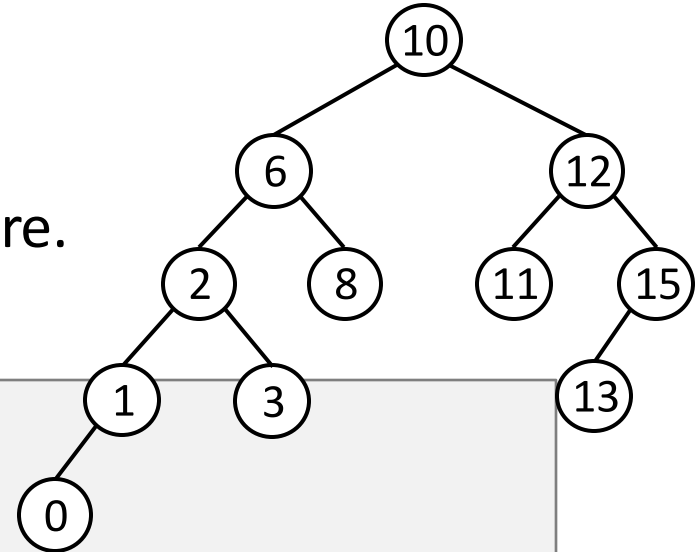
Binary Search Tree

- A binary search tree has some special rules to organize the data:
  - Each node has two children at most: left and right.
  - All nodes in the left subtree of a node are less than that node.
  - All nodes in the right subtree of a node are greater than that node.



Construct a BST: Algorithm

- The order of arrival affects the tree structure.

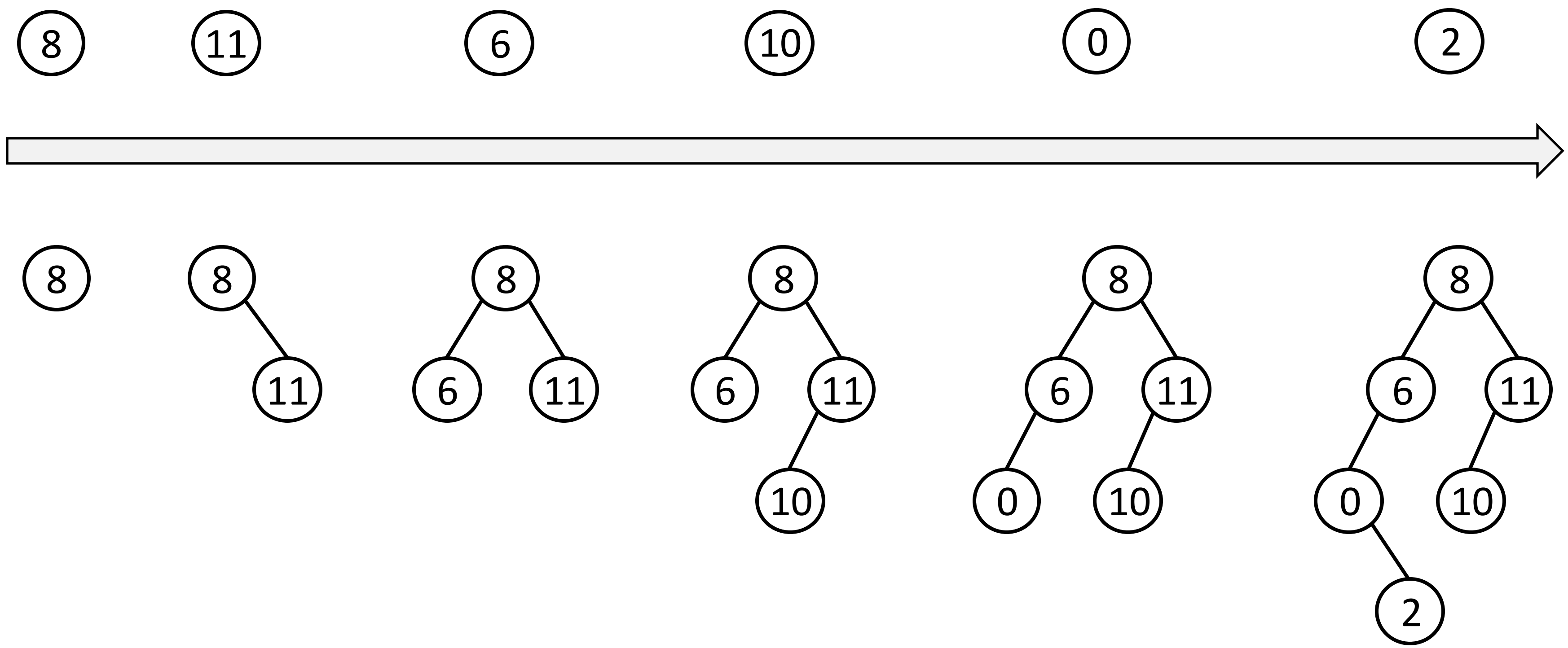



```
//Pseudo Code
Function Insert(node, new_value):
    If node is NULL:
        Create a new node with value = new_value
        Return the new node
    If new_value < node.value:
        node.left_child = Insert(node.left_child, new_value)
    Else if new_value > node.value:
        node.right_child = Insert(node.right_child, new_value)
    Else:
        // If new_value is equal to node.value, do nothing

    Return node
```

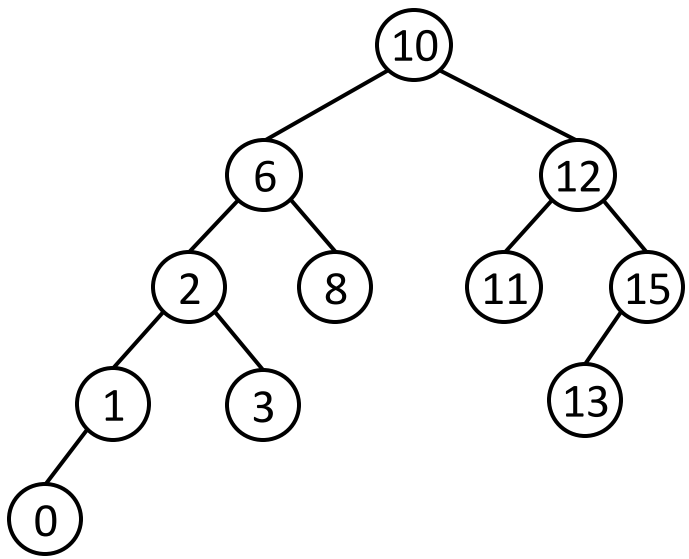
Construct a BST: Example

Suppose the following arrival order:



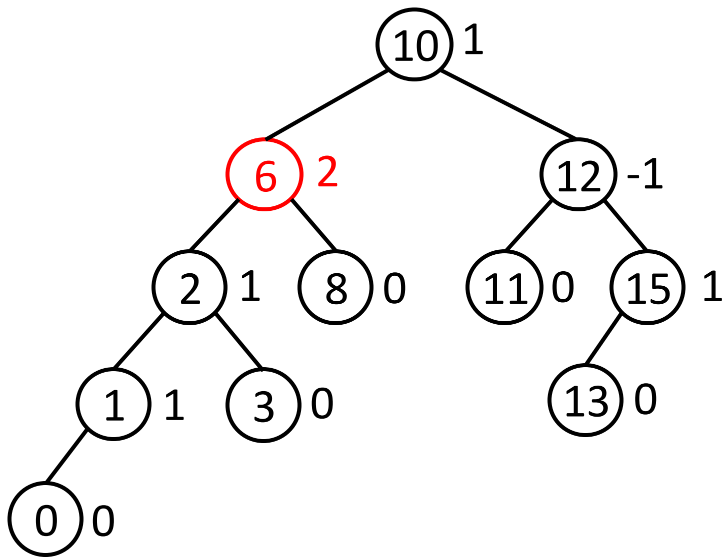
Cost of Operations on a BST

- Search/Insertion/deletion: height of the tree
  - Average: O(log n)
  - If the tree is highly imbalanced (worst case): O(n)

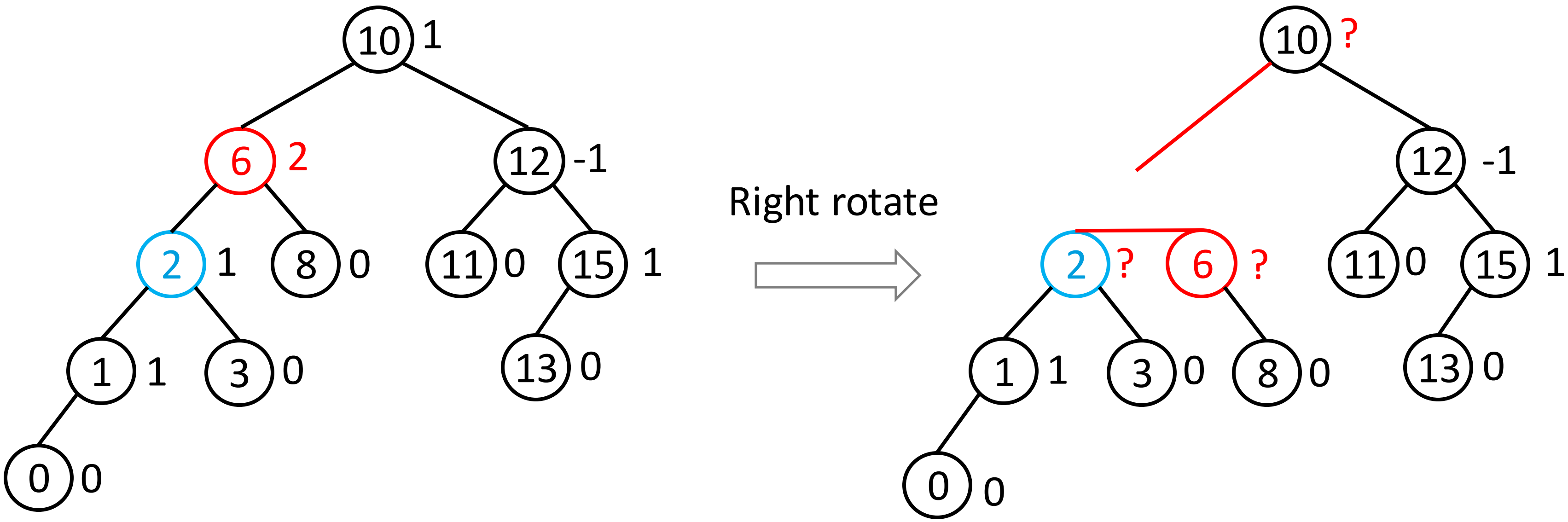


Self-balancing BST: AVL (Adelson-Velsky, Landis) Tree

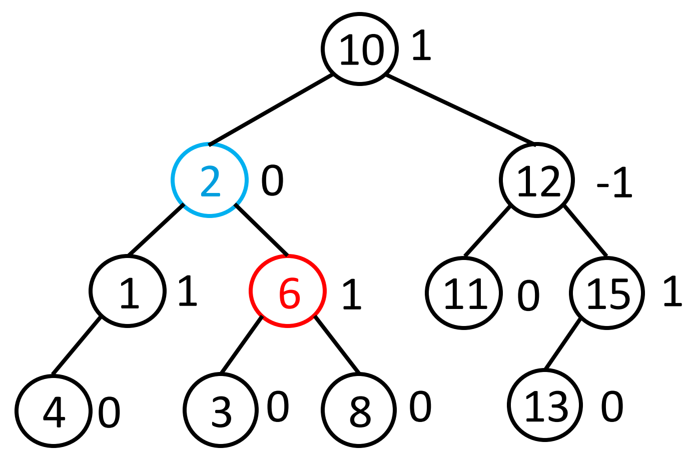
- The heights of the two child subtrees of any node differ by at most one.
- Maintain a balancing factor for each node:
  - $BF(x) = Height(x \to left) - Height(x \to right)$
- Rotate the subtree if $|BF(x)| > 1$



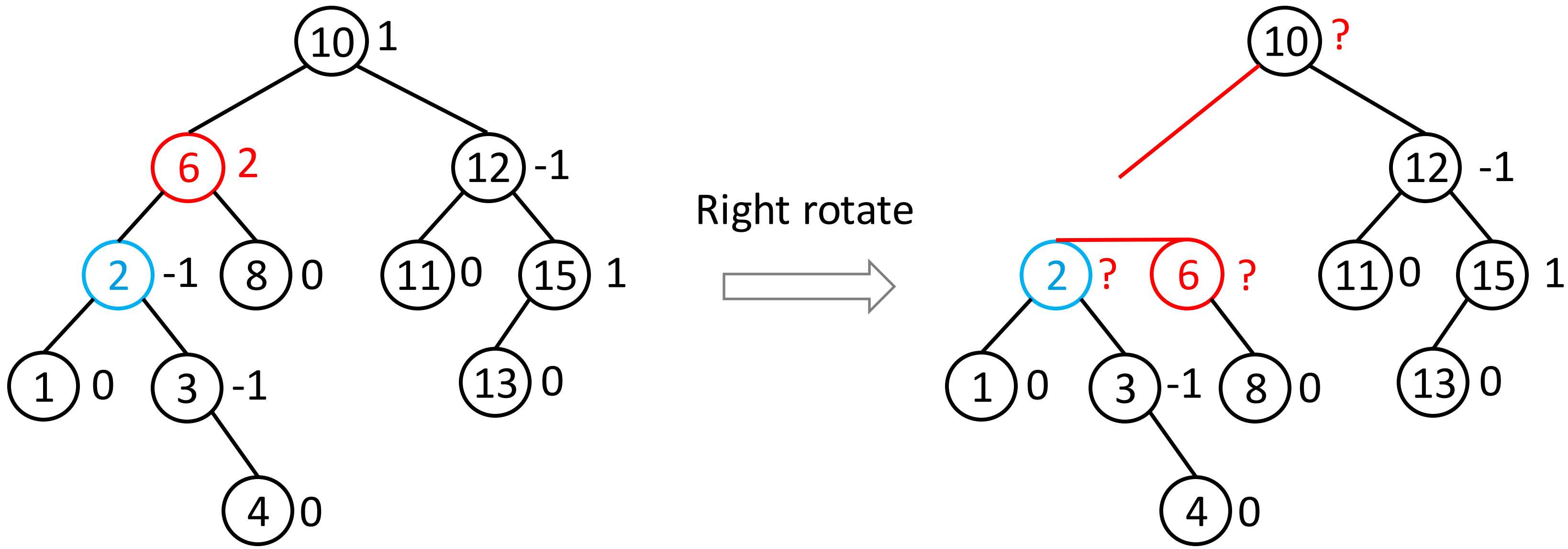
Imbalance Handling for Pattern 1: Rotation



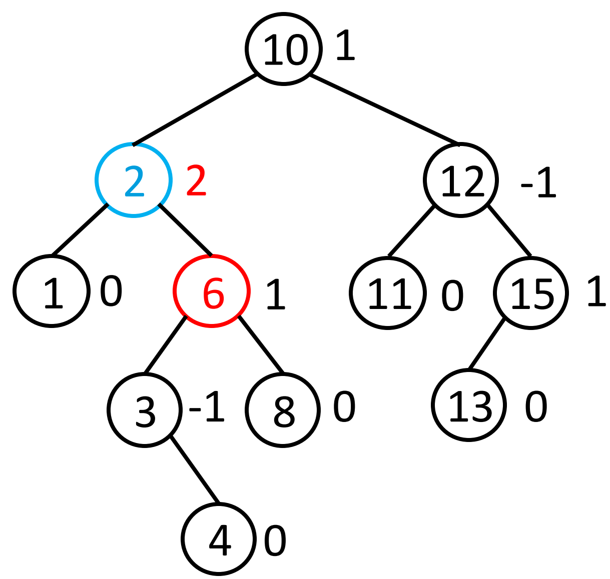
The **left** subtree of imbalanced node 6 is higher.
The **left** subtree of node 2 is higher.



Imbalance Handling for Pattern 2: Rotation

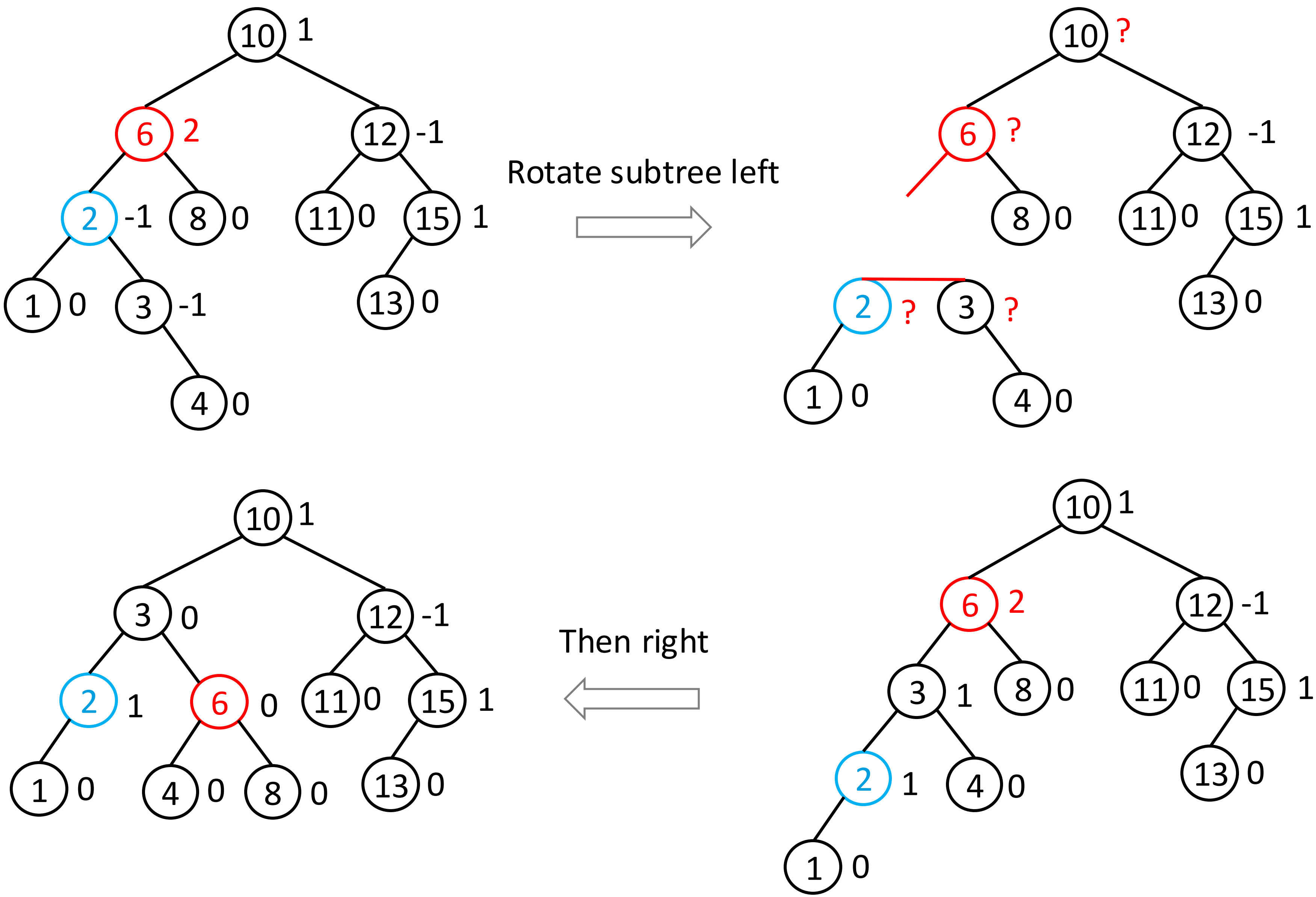


The **left** subtree of imbalanced node 6 is higher.
The **right** subtree of node 2 is higher.



Still Imbalance

Imbalance Handling for Pattern 1: Rotation

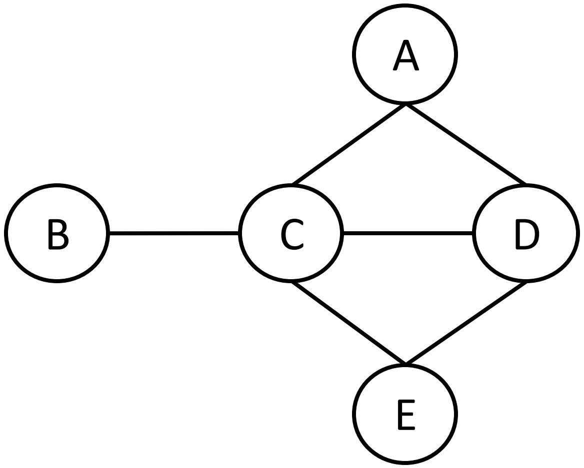


Rotation for Different Imbalance Patterns

	Which subtree of imbalanced node is higher?	Which child of the imbalanced node has a higher subtree?	Rotation Operation
Pattern 1	left	left	right rotation
Pattern 2	left	right	left-right rotation
Pattern 3	right	right	left rotation
Pattern 4	right	left	right-left rotation

2. Graph Analysis

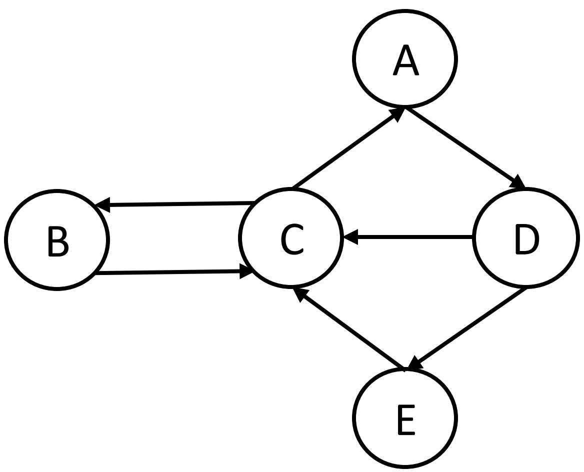
Recall: Graphs



undirected graph

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

adjacent matrix



directed graph

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

adjacent matrix

Graph Connectivity Problem

- Given an adjacent matrix, determine whether the graph is connected, *i.e.*, there is a path between every pair of vertices; if not, how many sub graphs are there?

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

{A, B}

{A, C}

{B, C}

{B, H}

{D, E}

{D, F}

{E, F}

{G, H}

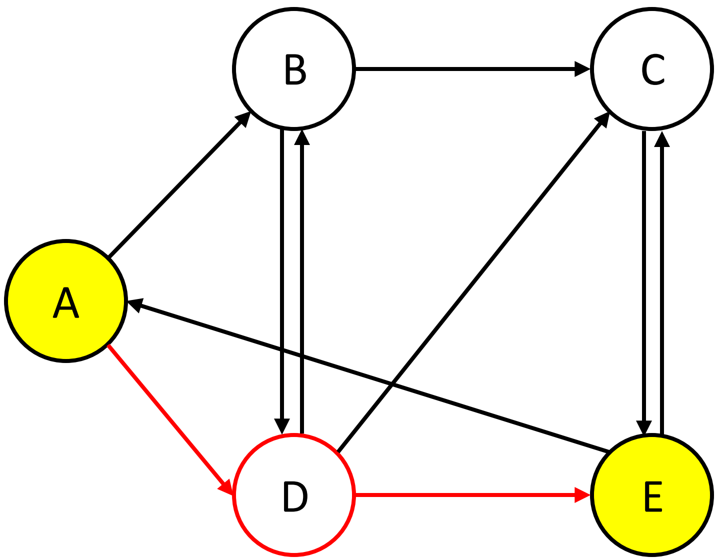
Union-Find Algorithm

- **Find**: Determine which set a particular element belongs to.
- **Union**: Merge two sets into a single set.

{A, B}	{A, B}
{A, C}	{A, B} {A, C} => {A, B, C}
{B, C}	{A, B, C} {B, C} => {A, B, C}
{B, H}	{A, B, C} {B, H} => {A, B, C, H}
{D, E}	{A, B, C, H} {D, E} => {A, B, C, H} {D, E}
{D, F}	{A, B, C, H} {D, E} {D, F}=> {A, B, C, H} {D, E, F}
{E, F}	{A, B, C, H} {D, E, F} {E, F} => {A, B, C, H} {D, E, F}
{G, H}	{A, B, C, H} {D, E, F} {G, H} => {A, B, C, H, G} {D, E, F}

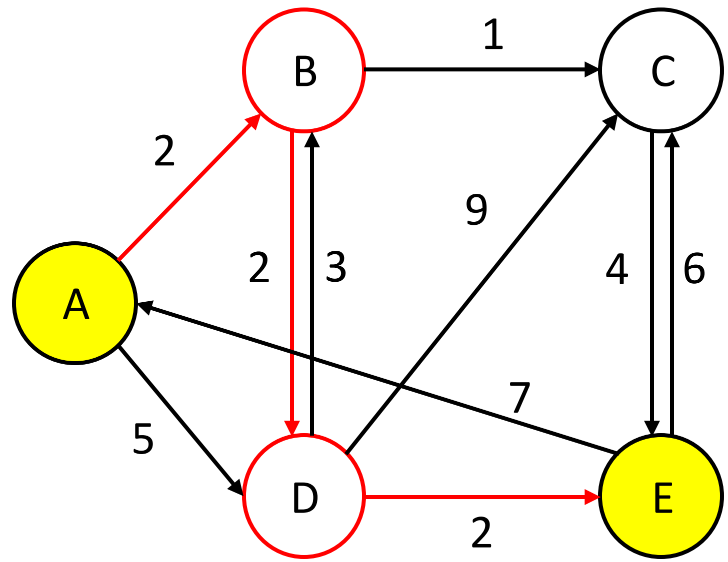
Shortest Path Problem

- Finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.



directed graph

shortest path: A-D-E
(cost: 2)

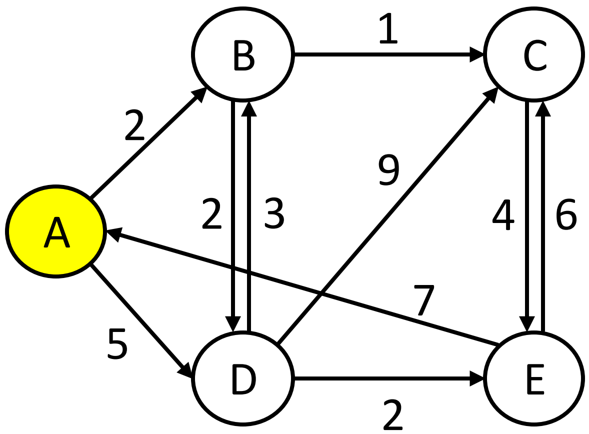


weighted directed graph

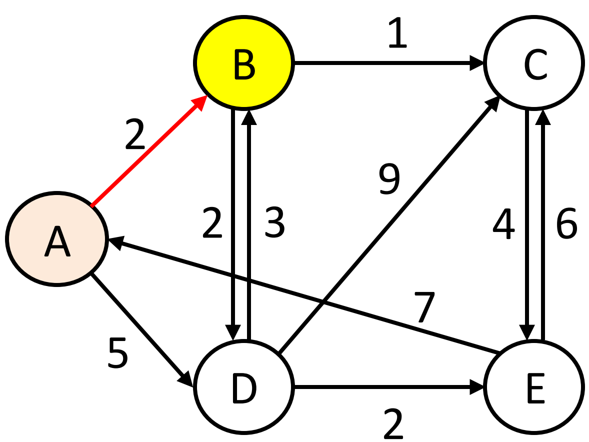
shortest path: A-B-D-E
(cost: 6)

Dijkstra's Algorithm

- Each time select a node with the small distance from visited nodes.

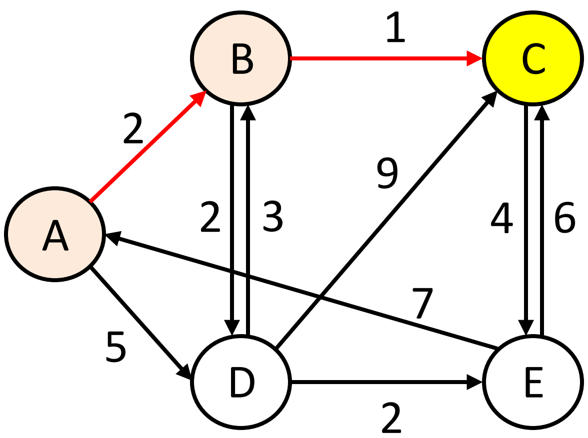


	A	B	C	D	E
Visited?	Y	N	N	N	N
Distance	0	2	∞	5	∞

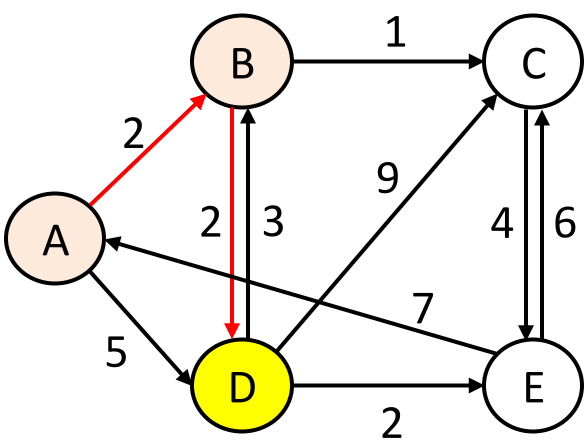


	A	B	C	D	E
Visited?	Y	Y	N	N	N
Distance	0	2	3	4	∞
Previous		A	B		

Dijkstra's Algorithm



	A	B	C	D	E
Visited?	Y	Y	Y	N	N
Distance	-	2	3	4	7
Previous		A	B	B	C



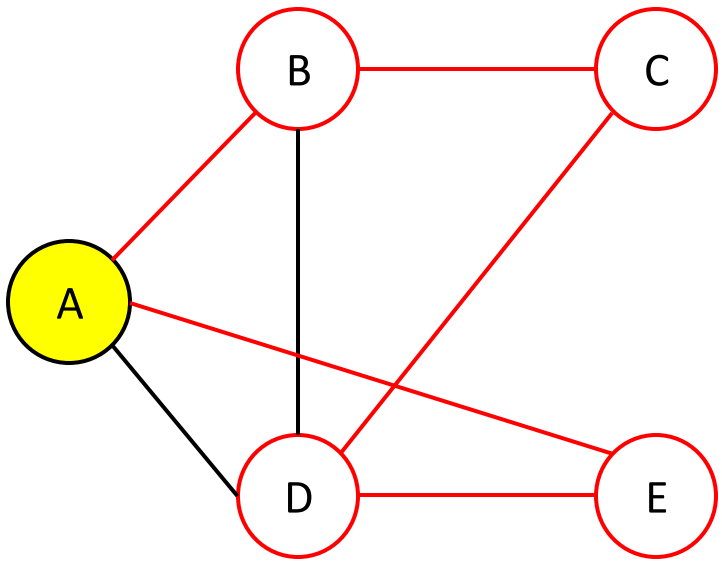
	A	B	C	D	E
Visited?	Y	Y	Y	N	N
Distance	-	2	3	4	6
Previous		A	B	B	D

Complexity of Dijkstra's Algorithm

- Supposing there are N nodes
  - We outdate each node in each round
  - In each round, we calculate the distance of the node to the rest nodes
- Complexity: $O(n^2)$

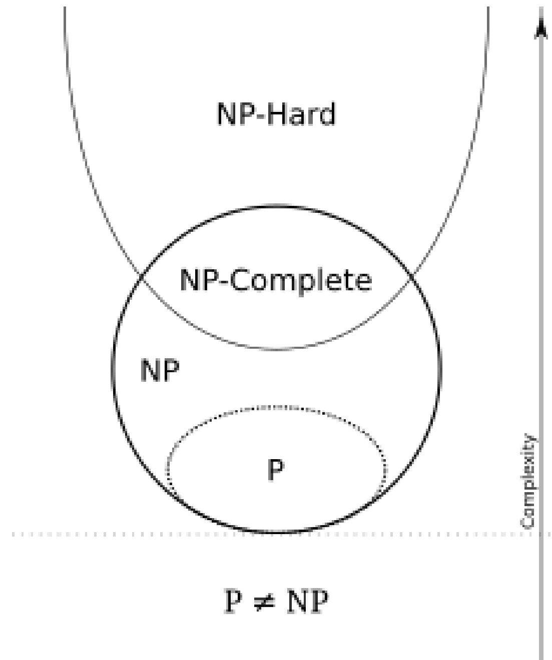
The Problem becomes more Difficult

- Travelling salesman problem (TSP): find the shortest path that visit each node exactly once (and returns to the original node).
- Hamiltonian cycle problem: TSP on unweighted graphs



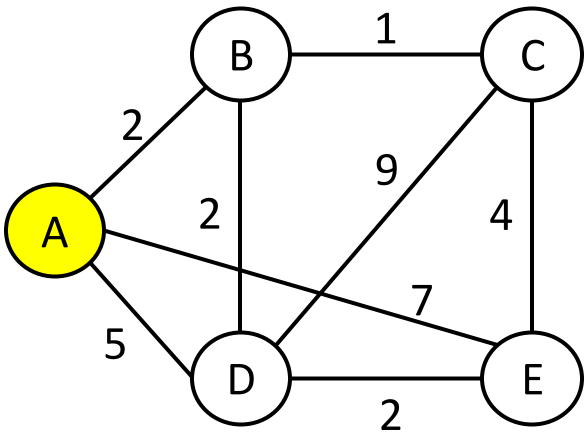
NP-Hard Problems

- P (polynomial time): the problem can be solved by a deterministic Turing machine in polynomial time.
- NP (nondeterministic polynomial time): the problems for which a solution can be verified by a deterministic Turing machine in polynomial time.

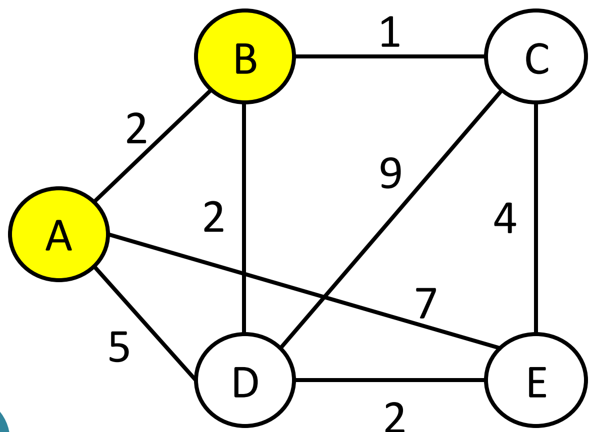


Solve TSP with Greedy Algorithm

- Heuristic: starting from the current nodes, select an edge to a new node with the smallest weight; fallback if the node forms a loop.

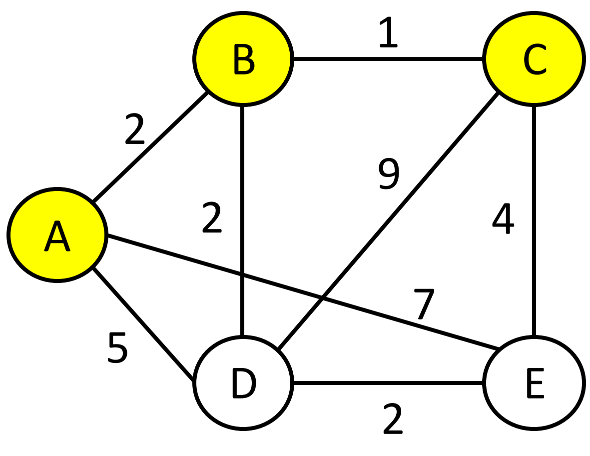


	A	B	C	D	E
Visited?	Y	N	N	N	N
Distance	0	2	∞	5	7

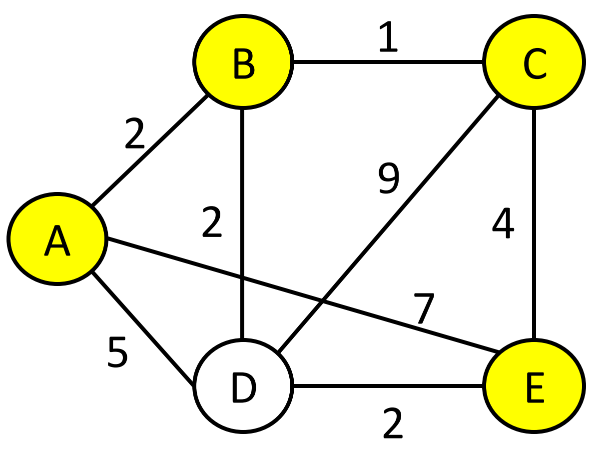


	A	B	C	D	E
Visited?	Y	Y	N	N	N
Distance	0	0	1	2	∞

Solve TSP with Greedy Algorithm



	A	B	C	D	E
Visited?	Y	Y	Y	N	N
Distance	0	0	0	9	4



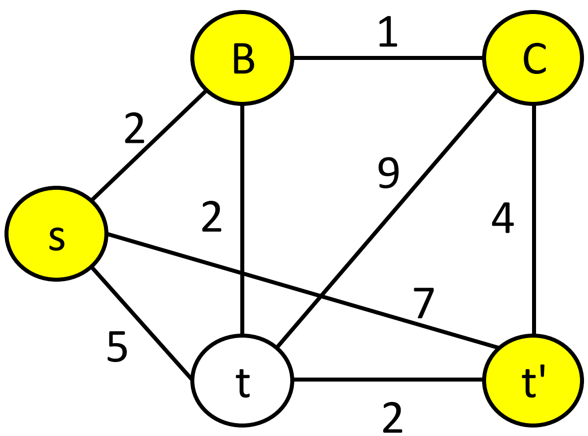
	A	B	C	D	E
Visited?	Y	Y	Y	N	N
Distance	0	0	0	2	0

Cost: 2+1+4+2+5 = 14

Solve TSP with Dynamic Programming

- Supposing the final solution is {s, …, t’, t}

$$Cost(S,t) = \begin{cases} \min_{t' \in S}\left(Cost(S \setminus \mathrm{t}, t') + w(t', t)\right) & if\,|S| > 1 \\ 0 & otherwise \end{cases}$$



3. In-class Practice

In-class Practice

- Design a program that can simulate stock exchange behavior.
  - The program should include a data structure for managing ask and bid orders.
  - Execute the program with a sequence of ask and bid orders.
  - Orders should be matched according to price-time priority.
  - A trade is executed if:
    - The bid price is greater than or equal to the ask price.
    - The ask price is less than or equal to the bid price.