MF20006: Introduction to Computer Science

Lecture 7: Algorithm II

Hui Xu

xuh@fudan.edu.cn

Outline

1. Max/Min Heap

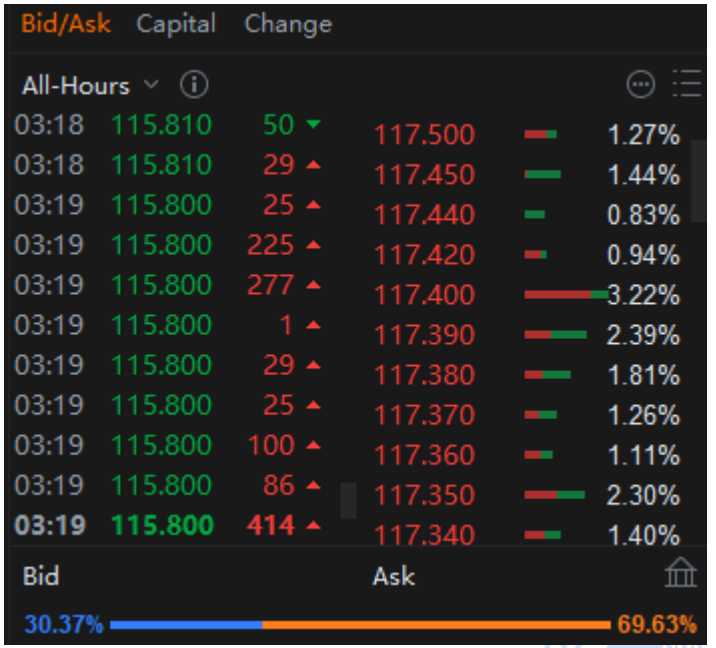
2. Binary Search Tree

3. Problems on Graphs

1. Max/Min Heap

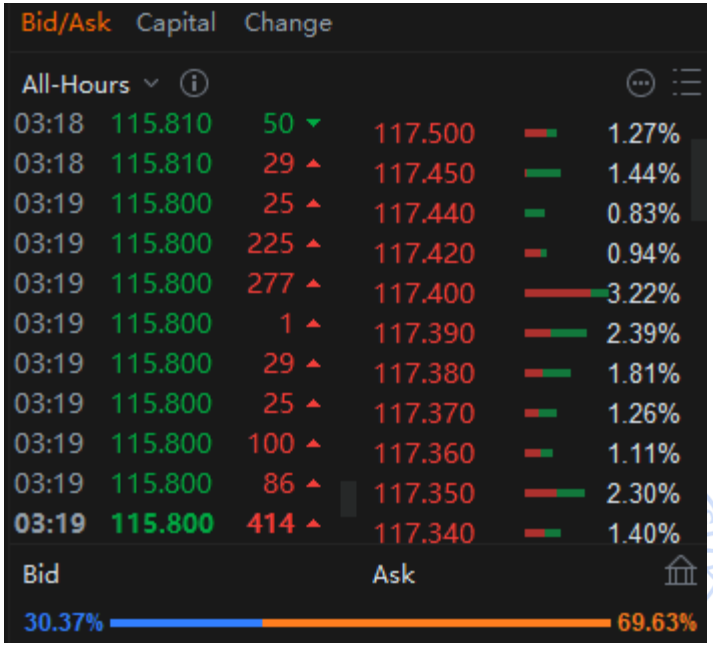
Problem

- 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 min price they are willing to sell.
- Bid order: The buyer specifies the max price they are willing to pay.



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.



Trade Price

- **The trade price is typically determined based on the existing order.**
- **At the ask price:**
  - There are ask orders in the book (with prices higher than existing bid orders).
  - When a new bid order arrives with a price higher than or equal to the best ask, a trade is executed at the ask price.
- **At the bid price:**
  - There are bid orders in the book (with prices lower than existing ask orders).
  - When a new ask order arrives with a price lower than or equal to the best bid, a trade is executed at the bid price.

Problem

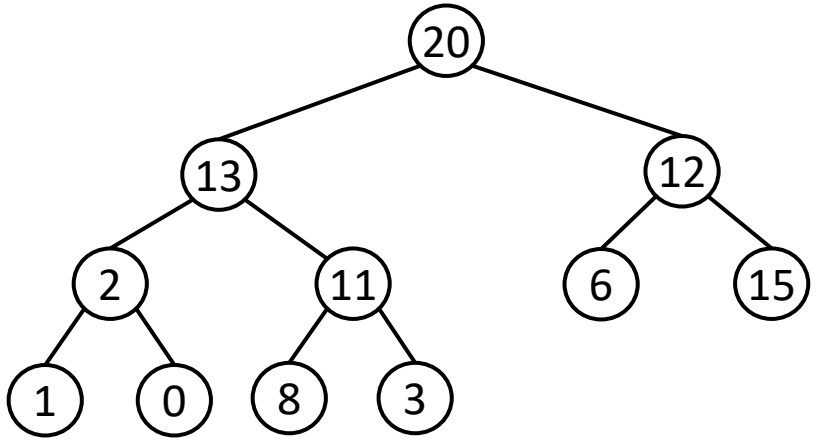
- **Design an algorithm that efficiently manages incoming orders in real time, ensuring that trades are executed according to predefined matching rules.**
- **Input:**
  - A continuous stream of ask and bid orders arriving concurrently.
  - Each order specifies the quantity of shares and the price per share.
- **Output:**
  - Two dynamically maintained data structures, one for asks and one for bids.
  - Each structure is sorted by price and arrival time for orders with the same price.
  - These structures should support efficient order matching.

Sorting?

- Cost of retrieval the top-k elements is constant.
- However, the cost of inserting an element is n/2.

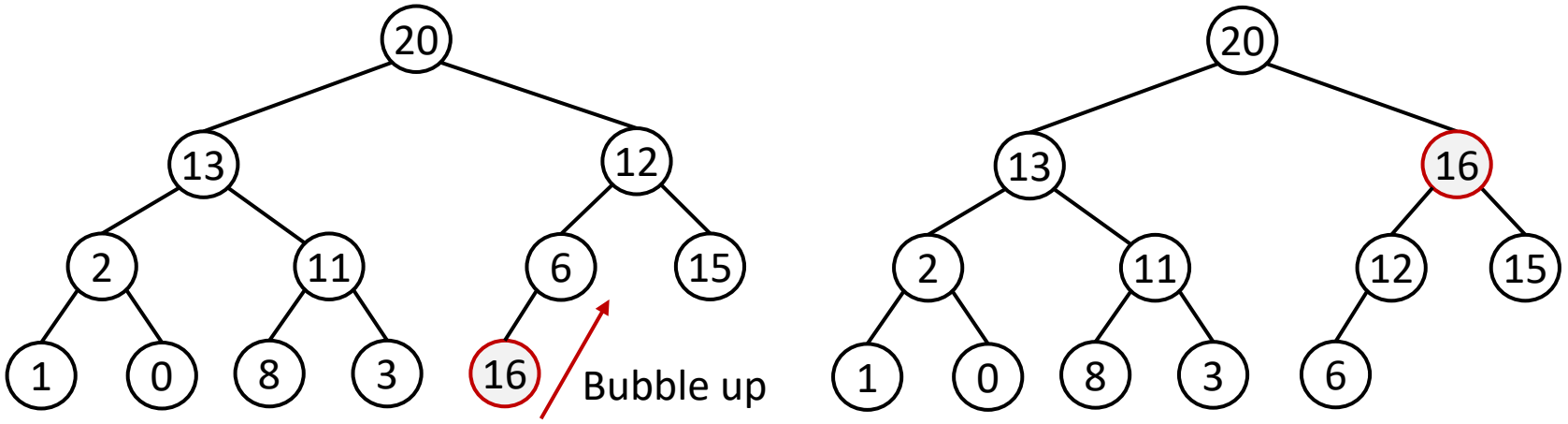
Max Heap

- A max heap is a type of binary tree.
- Heapify: Every node's value is greater than or equal to the values of its children.
- Completeness: Every level of the tree is filled from left to right.



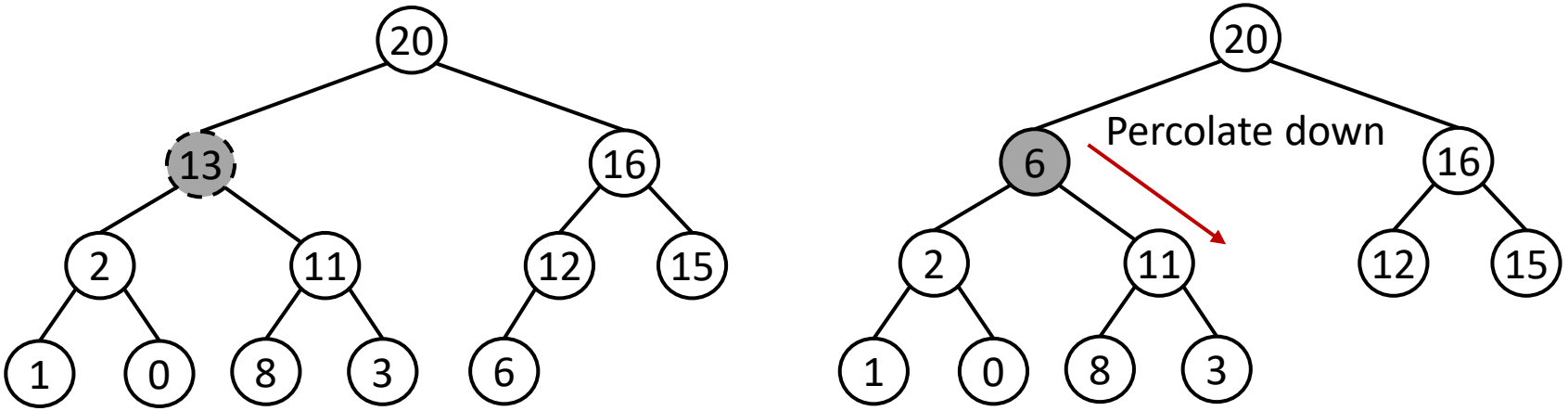
Insert an Element

- To maintain completeness: Add a new value at the end.
- Heapify: Swap with parent while larger ("bubble up").



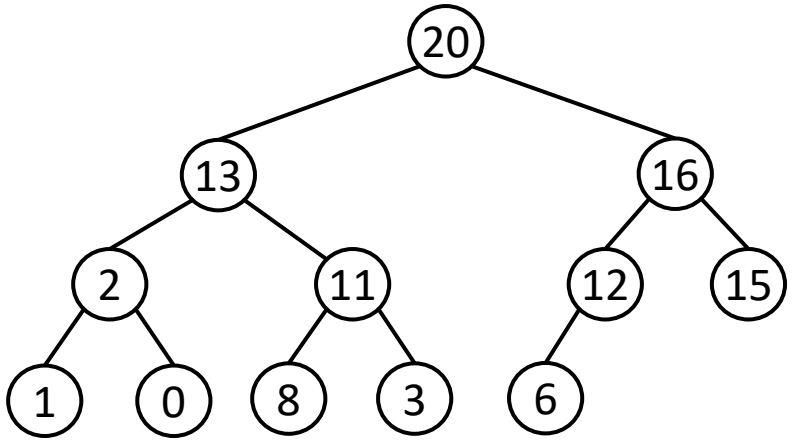
Delete an Element

- Replace it with the last element (to maintain the complete tree).
- Remove the last element.
- Restore the heap property by either:
  - Heapify down towards the larger child if the new value is smaller than children.
  - Heapify up if the new value is larger than parent.



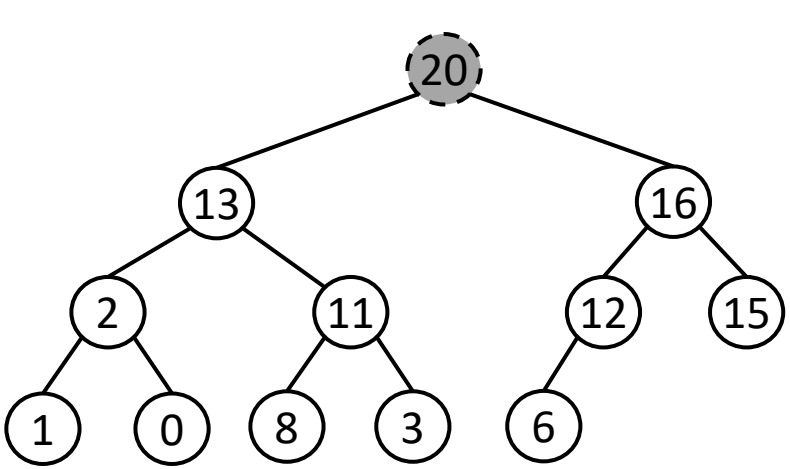
Find the Top-K Element

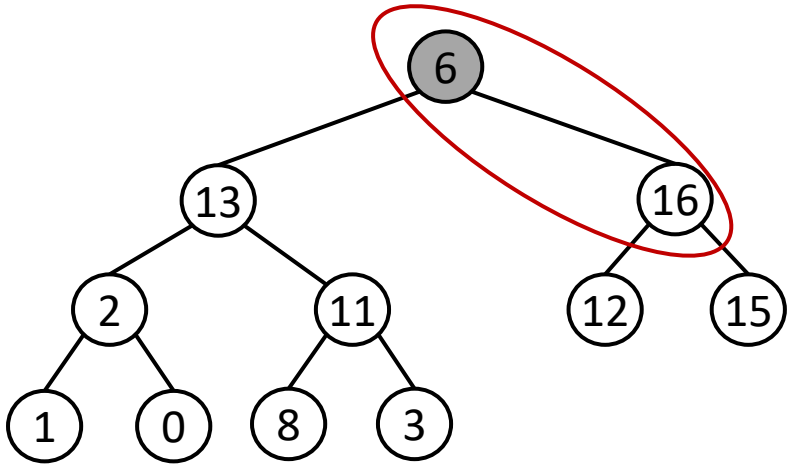
- Top 1: Just the root. (20)
- Top 2: One of the children of the root. (20) (16)
- Top 3: One of the children of top-2 elements. (20) (16) (15)
- Top 4: ...

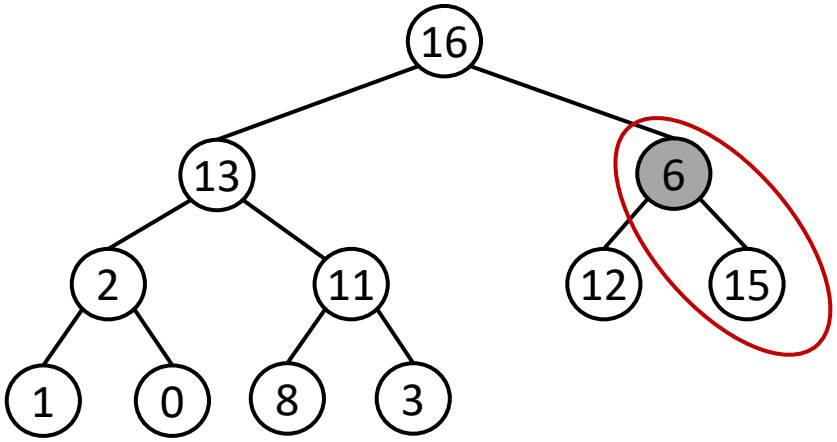


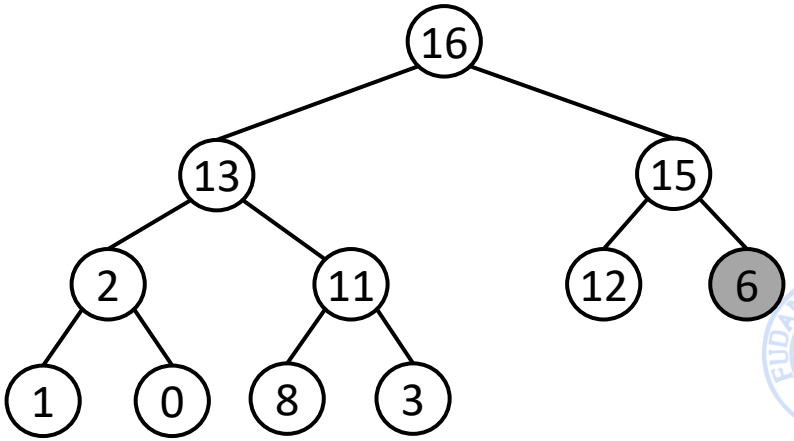
Find the Top-K Element

- In the case of trading, we should delete the matched elements.
- Top k: Repeatedly remove the root k times



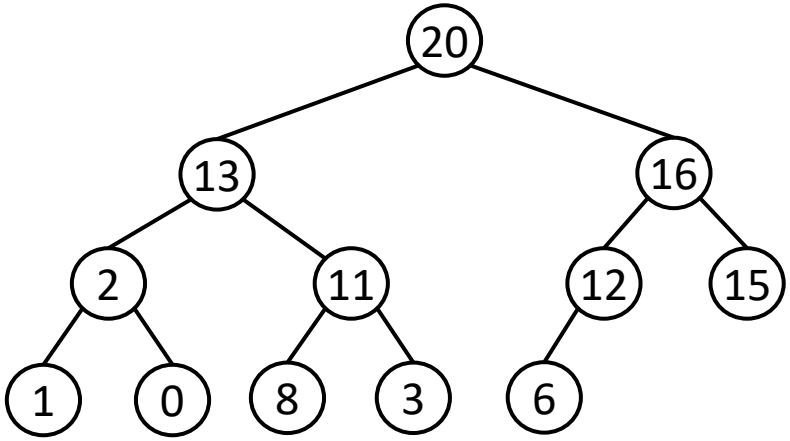


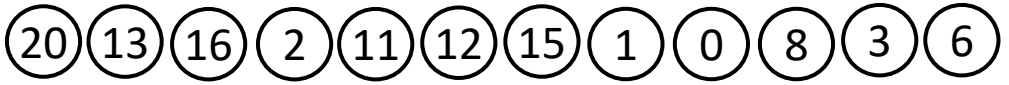




Implementation based on List

- Store the tree nodes in level-order traversal (top-down) as a list.
- The index of the children node given a node i:
  - Left: 2*i + 1
  - Right: 2*i + 2
- The index of the parent node given a node i:
  - (i-1)/2

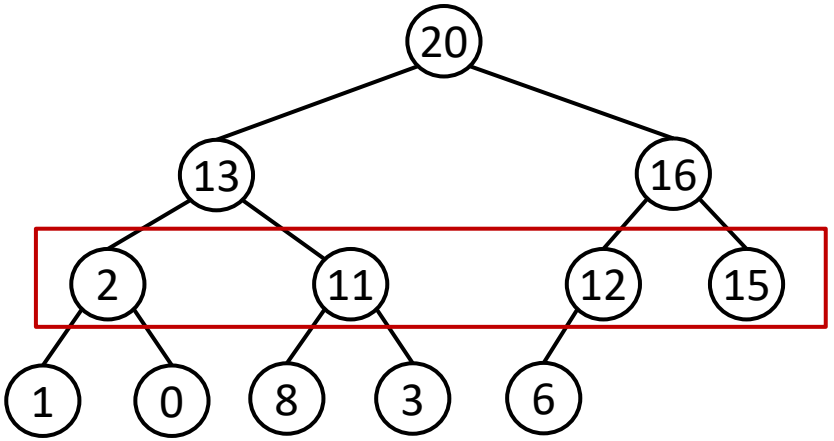




level-order traversal

Proof?

- **Suppose the number of elements in the parent layer is n.**
  - The index should be [n-1,…,2n-2].
  - Because the number of elements in previous layers is n-1.
- **The number of elements (if full) in the child layers is 2n.**
  - The index should be [2n-1,…,4n-2]



n-1 elements in previous layers

n elements in this layer

2n elements in the child layer

Implementation

```
@dataclass
class MaxHeap:
    data: List[int] = field(default_factory=list)

    def push(self, val: int):
        """Insert a new value and bubble up."""
        self.data.append(val)
        i = len(self.data) - 1
        # Bubble up
        while i > 0:
            parent = (i - 1) // 2
            if self.data[i] > self.data[parent]:
                swap(self.data[i], self.data[parent])
                i = parent
            else:
                break
```

Implementation

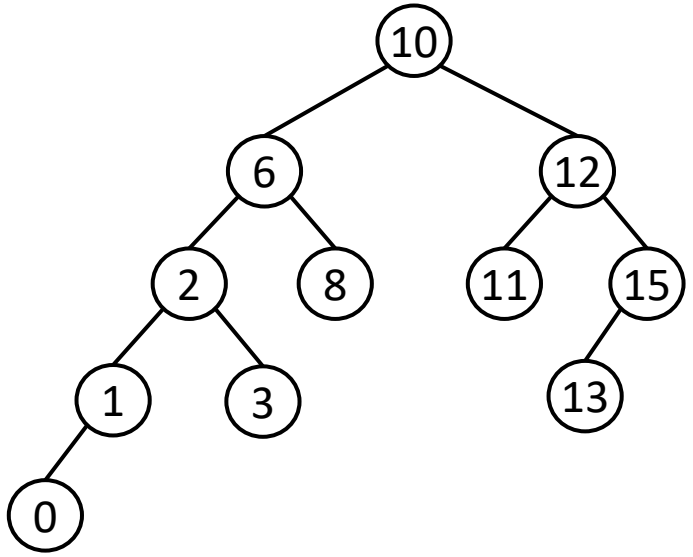
```
def pop(self) -> int:
    """Remove and return the max element."""
    if not self.data:
        raise IndexError("pop from empty heap")
    n = len(self.data)
    self.data[0], self.data[-1] = self.data[-1], self.data[0]
    max_val = self.data.pop() # pop self.data[-1]
    # Bubble down
    i = 0
    while True:
        left, right = 2 * i + 1, 2 * i + 2
        largest = i
        if left < len(self.data) and self.data[left] > self.data[largest]:
            largest = left
        if right < len(self.data) and self.data[right] > self.data[largest]:
            largest = right
        if largest == i:
            break
        self.data[i], self.data[largest] = self.data[largest], self.data[i]
        i = largest
    return max_val
```

2. Binary Search Tree

Binary Search Tree

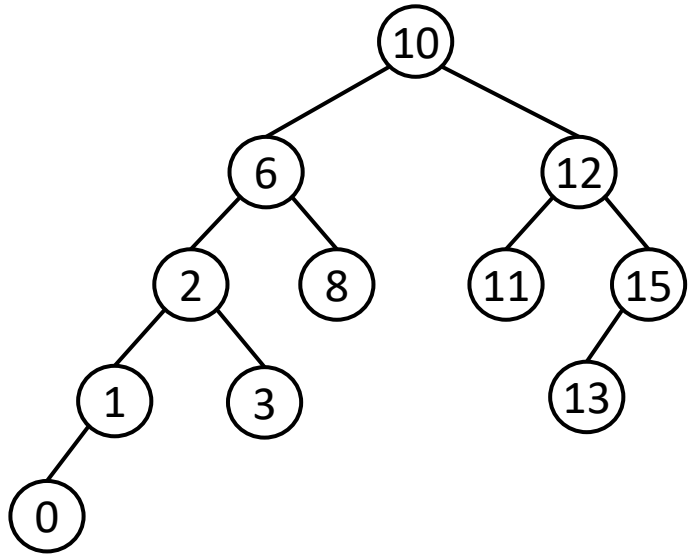
❑A binary search tree has some special rules to organize the data:

- ➢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.



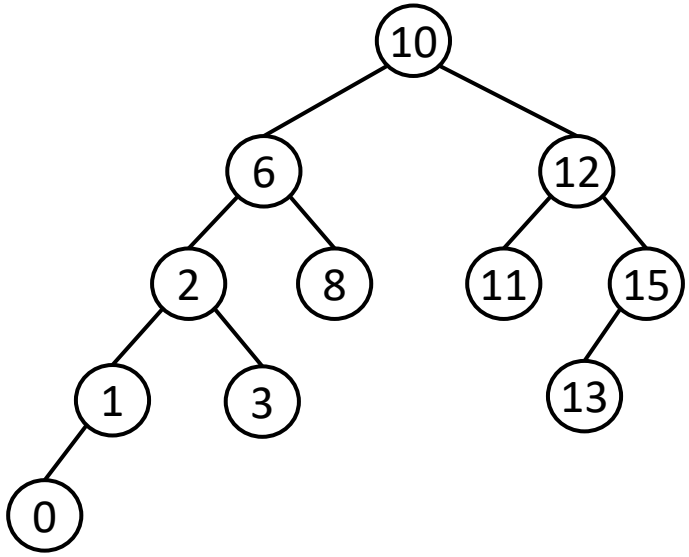
Data Structure of a BST

```
@dataclass
class Node:
    key: int
    left: Optional[Node] = None
    right: Optional[Node] = None

@dataclass
class BinarySearchTree:
    root: Optional[Node] = None
```



Construct a BST: Insert Nodes

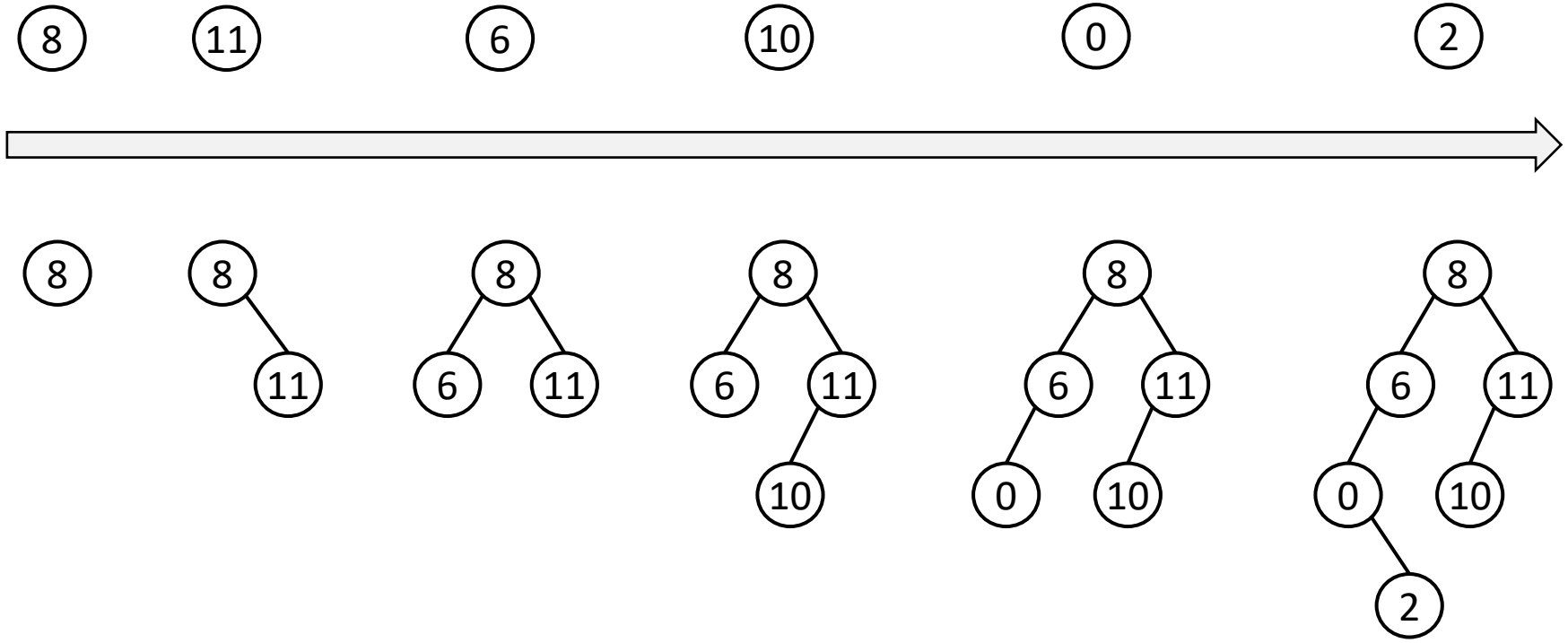
```
def insert(self, key):
    new_node = Node(key)
    if self.root is None:
        self.root = new_node
        return
    current = self.root
    while True:
        if key < current.key:
            if current.left is None:
                current.left = new_node
                return
            current = current.left
        elif key > current.key:
            if current.right is None:
                current.right = new_node
                return
            current = current.right
        else:
            return
```



Construct a BST: Example

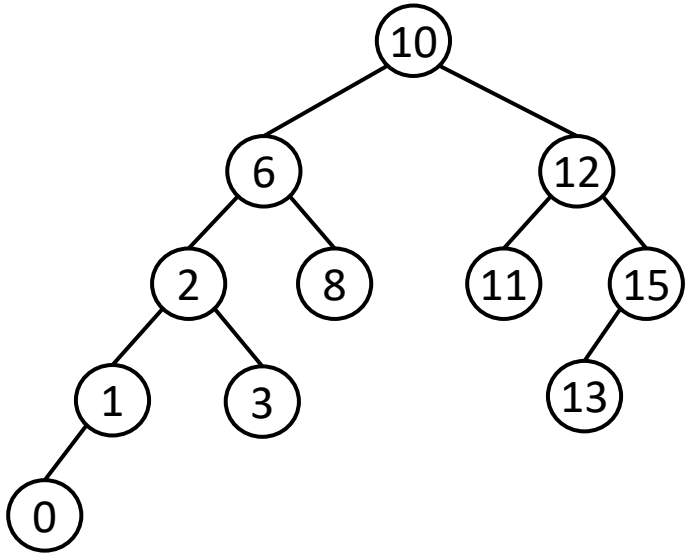
❑The order of arrival affects the tree structure.

Suppose the following arrival order:



Search on a BST

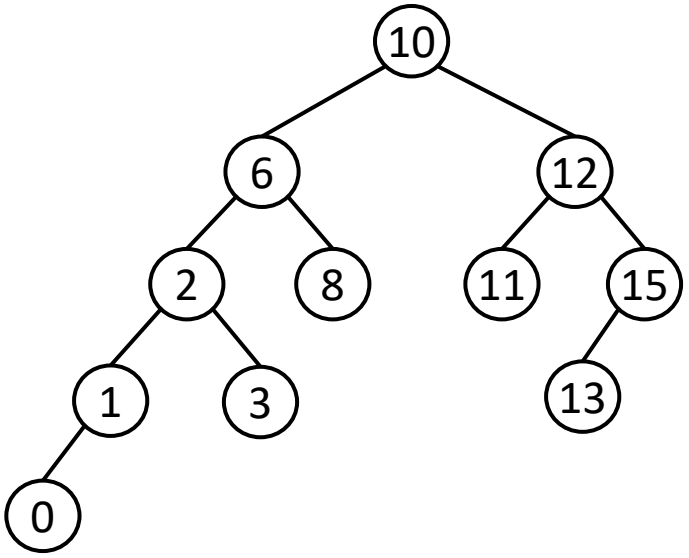
```
def search(self, key):
    current = self.root
    while current is not None:
        if key == current.key:
            return True
        elif key < current.key:
            current = current.left
        else:
            current = current.right
    return False
```

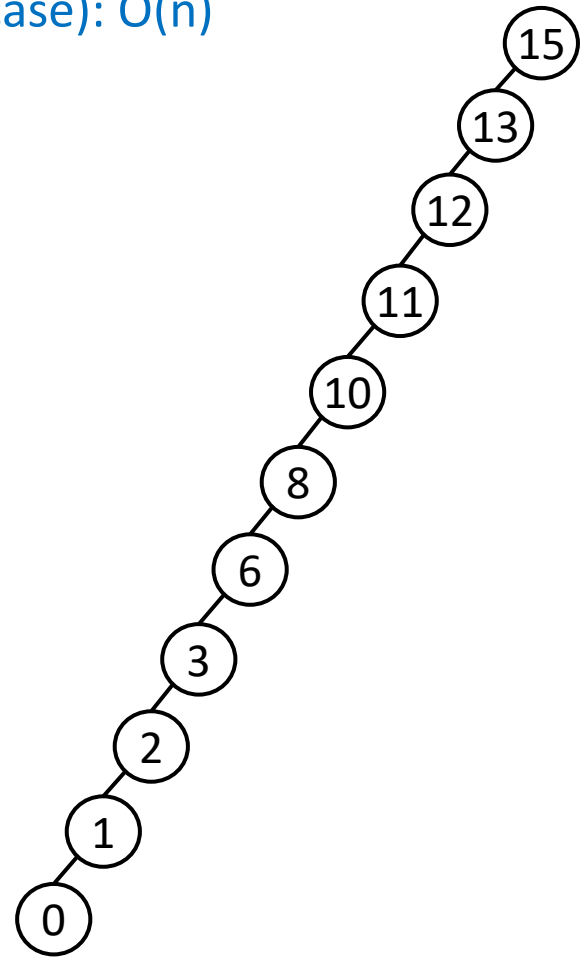


Cost of Operations on a BST

❑Search/Insertion/deletion: height of the tree

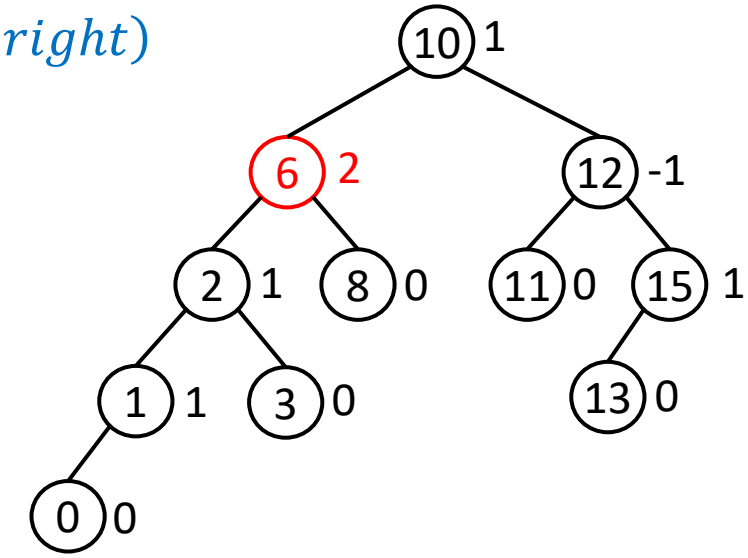
- ➢Average: O(log n)
- ➢If the tree is highly imbalanced (worst case): O(n)



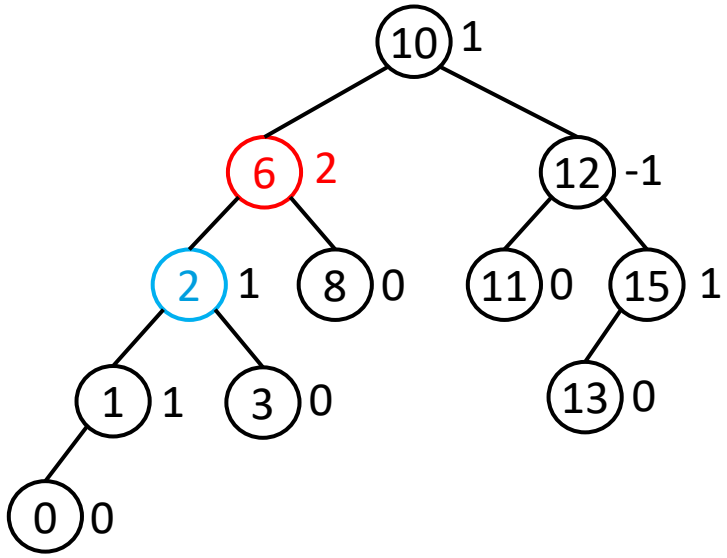


Self-Balancing BST: AVL Tree

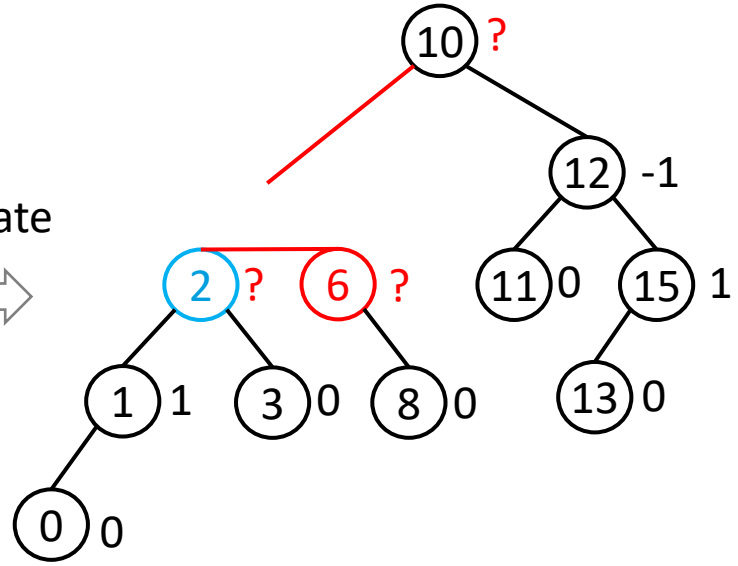
- The heights of the two child subtrees of any node differ by at most one.
- Maintain a balancing factor for each node:
  - $BF(x) = Height(x \rightarrow left) - Height(x \rightarrow right)$
- Rotate the subtree if $|BF(x)| > 1$



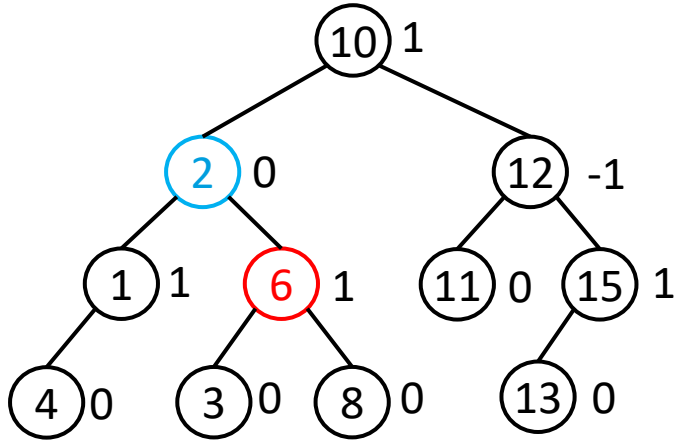
Imbalance Handling for Pattern 1: Rotation



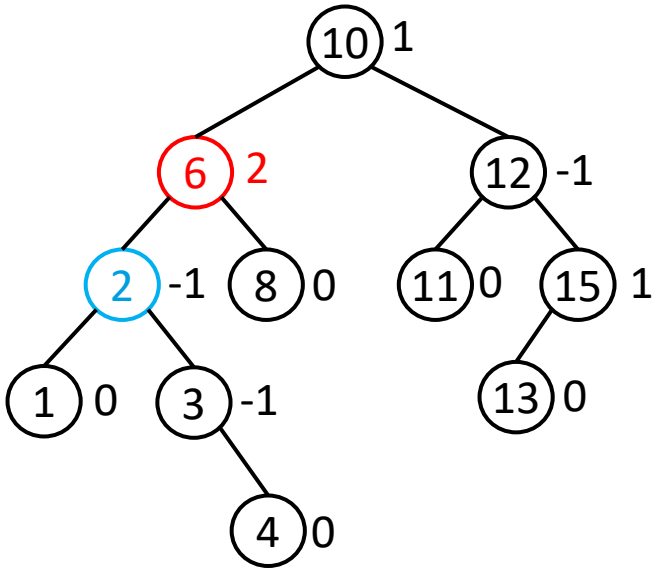
Right rotate



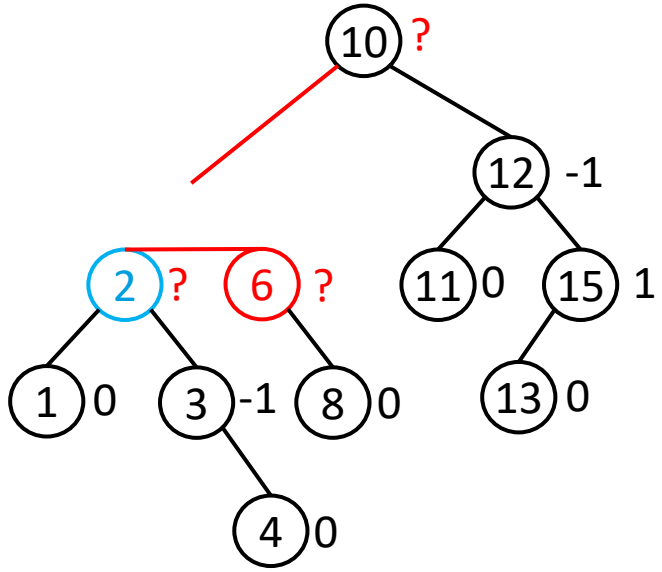
The **left** subtree of imbalanced node 6 is higher.
The **left** subtree of node 2 is higher.



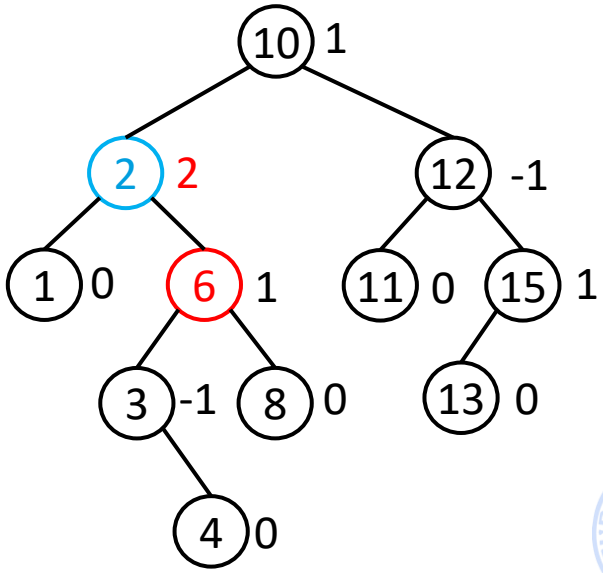
Imbalance Handling for Pattern 2: Rotation



Right rotate

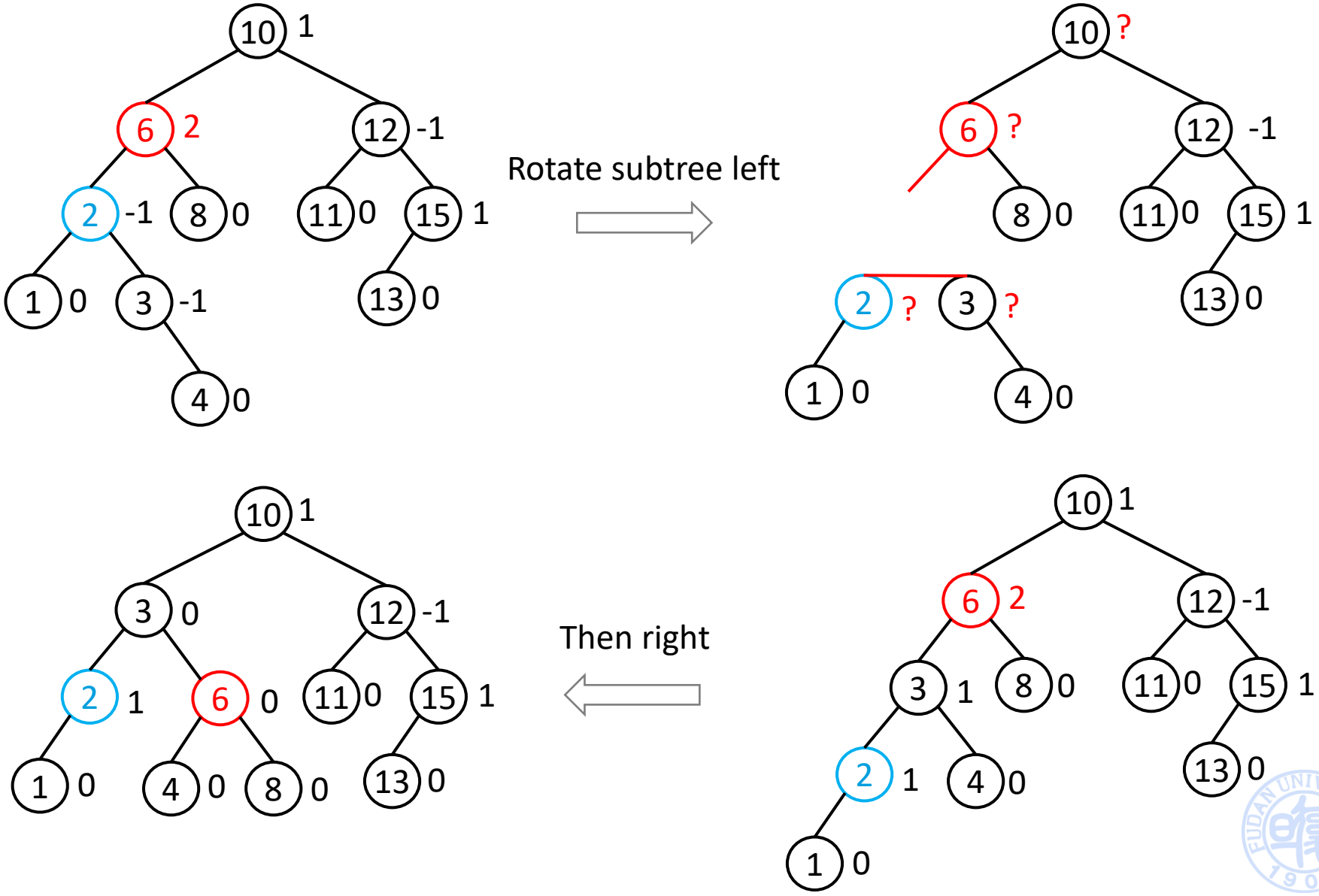


The **left** subtree of imbalanced node 6 is higher.
The **right** subtree of node 2 is higher.



Still Imbalance

Imbalance Handling for Pattern 2: Rotation



Rotation for Different Imbalance Patterns

	Which subtree of the 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

Implementation with Python

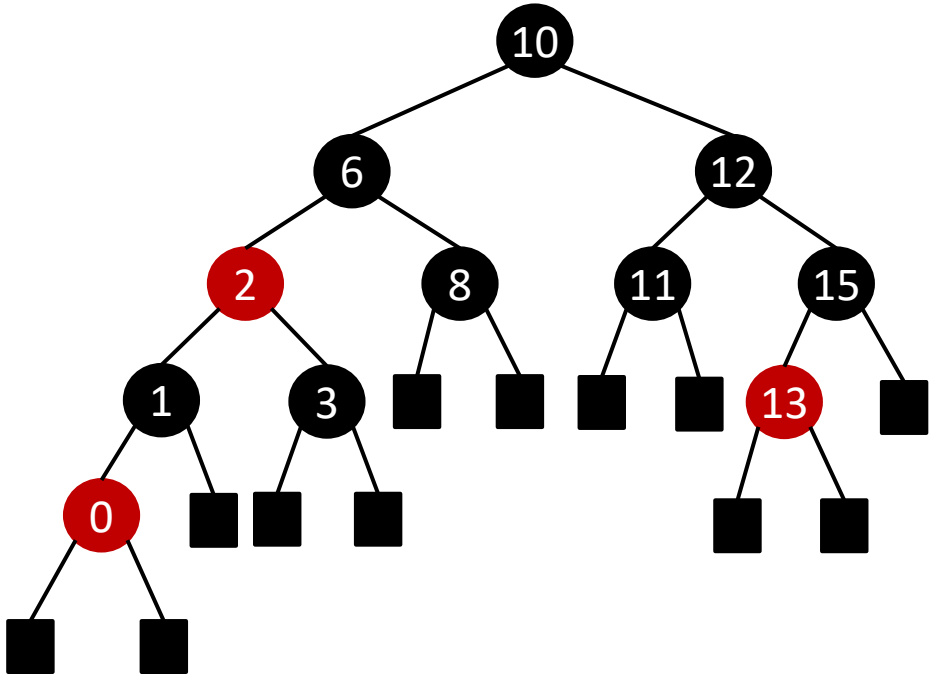
```
@dataclass
class AVLNode:
    key: int
    left: Optional[AVLNode] = None
    right: Optional[AVLNode] = None
    height: int = 1

@dataclass
class AVLTree:
    root: Optional[AVLNode] = None
```

Implementation with Python

```
def insert(self, key):
    ... # insert node
    self._update_height(node)
    balance = self._balance_factor(node)
    # Left heavy
    if balance > 1:
        if key < node.left.key:
            return self._rotate_right(node)  # LL
        else:
            node.left = self._rotate_left(node.left)  # LR
            return self._rotate_right(node)
    # Right heavy
    if balance < -1:
        if key > node.right.key:
            return self._rotate_left(node)  # RR
        else:
            node.right = self._rotate_right(node.right)  # RL
            return self._rotate_left(node)
        return node
```

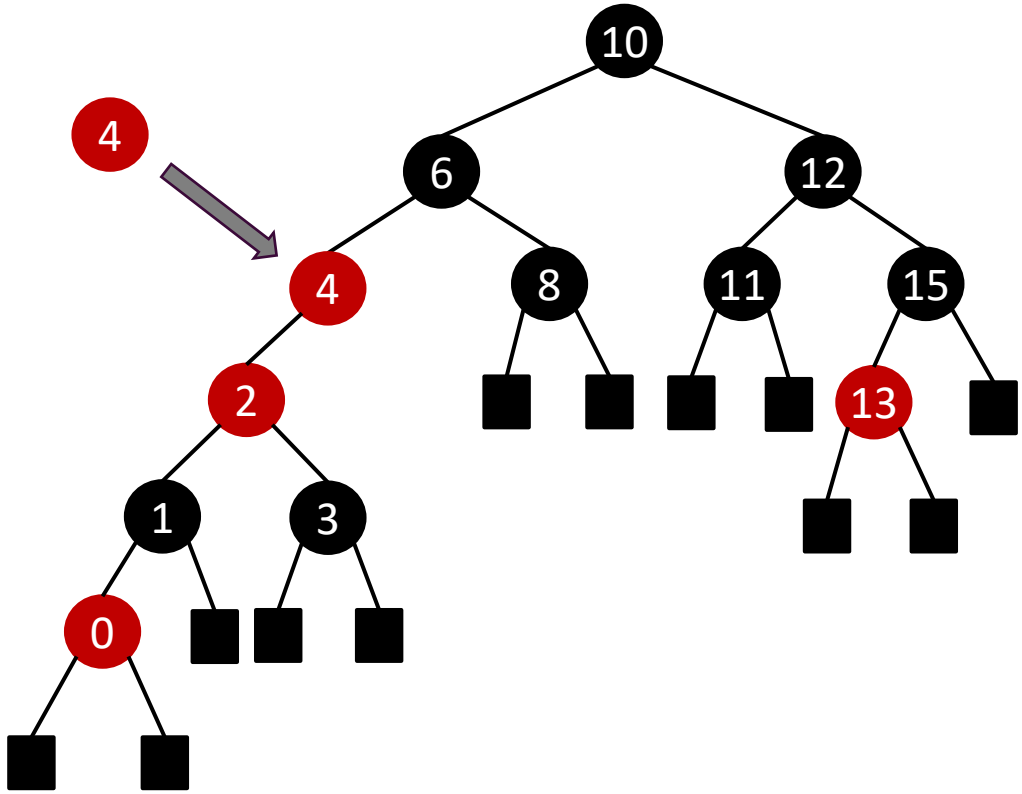
Red-black Tree

- **A BST with extra properties to ensure it stays roughly balanced.**
  - Every node is either red or black.
  - The root is black.
  - Every leaf (NIL / null pointer) is black.
  - If a node is red, then both its children are black.
  - For each node, all paths from the node to its descendant leaves have the same number of black nodes.



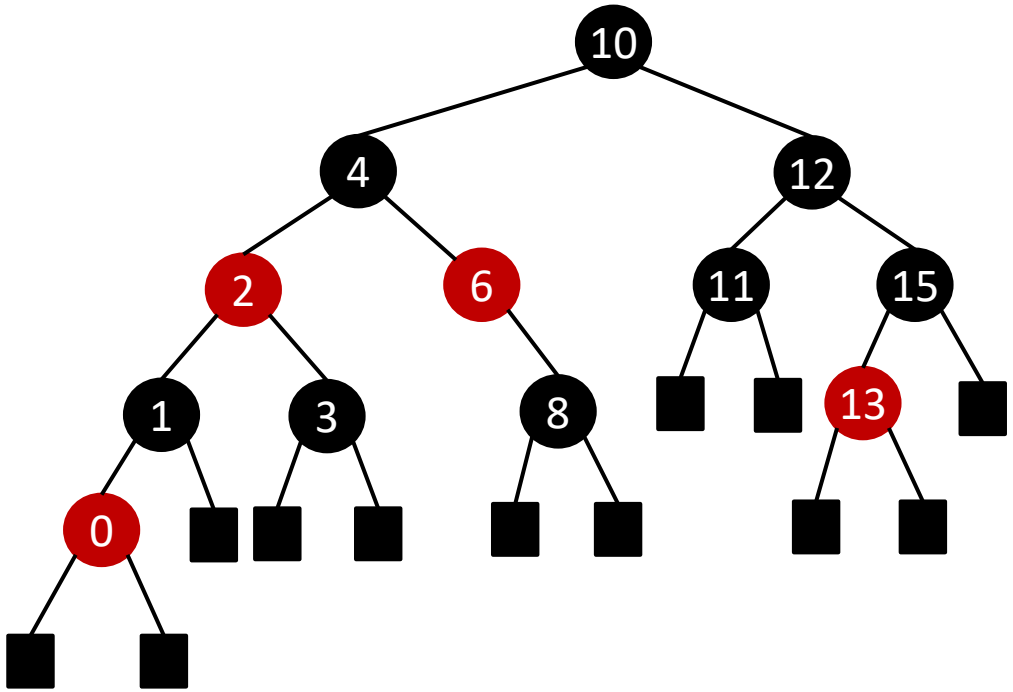
Red-black Tree: Insert a Node

- Init as red.



Red-black Tree: Insert a Node

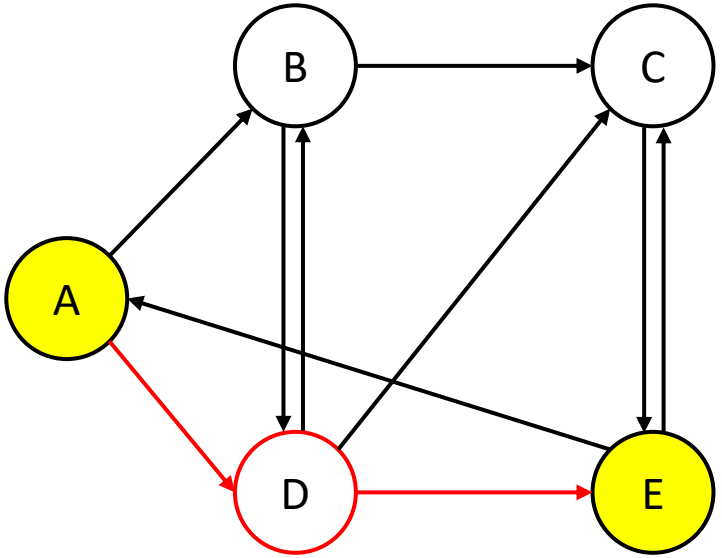
- **Local adjust based on some rules (rotations and recoloring).**
  - We do not enumerate all details.
- **Without needing to explicitly check the black-height of all paths.**
  - Compared to AVL: AVL tree needs to update the balance factor of ancestor nodes.



3. Problems on Graphs

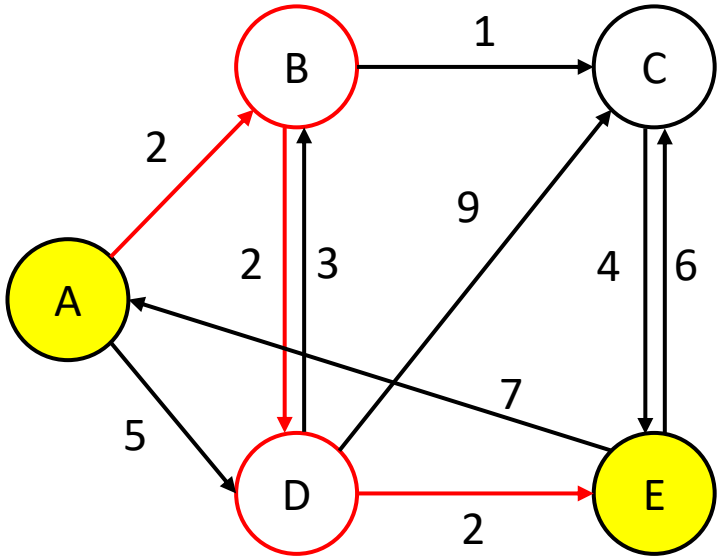
Shortest Path Problem

- **Finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.**
  - Unweighted graph: minimum number of edges.
  - Weighted graph: accumulated weight of all edges.



unweighted graph

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

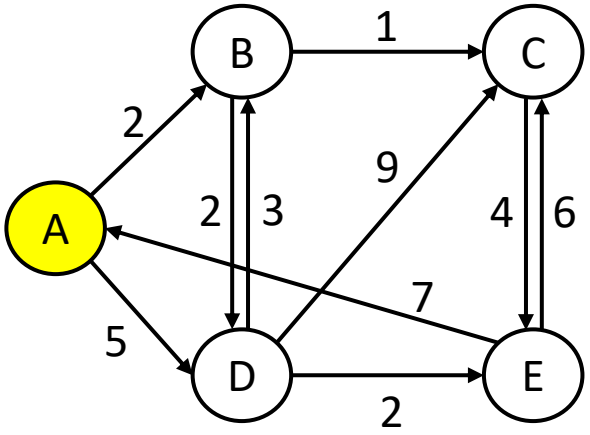


weighted graph

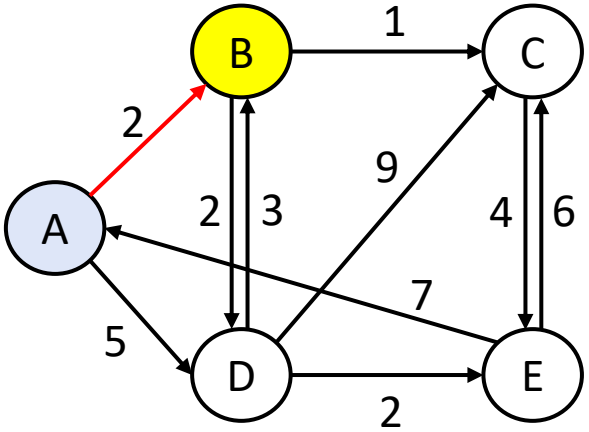
shortest path: A-B-D-E (cost: 6)

Dijkstra's Algorithm

- Each time select a node with the small distance from visited nodes.
- Maximize the number of nodes that can be visited given a budget.

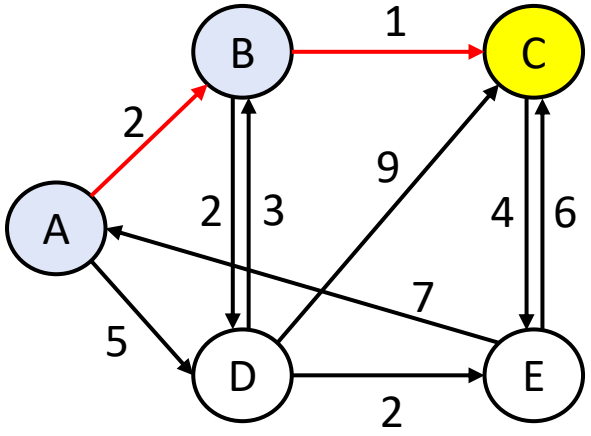


	A	B	C	D	E
Visited?	Y	N	N	N	N
Distance	0	2	∞	5	∞

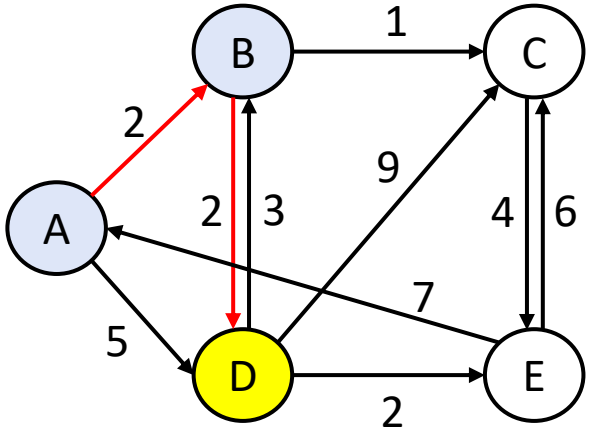


	A	B	C	D	E
Visited?	Y	Y	N	N	N
Distance	0	2	3	4	∞
Previous		A	B		

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	Y	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:** $O(n^2)$

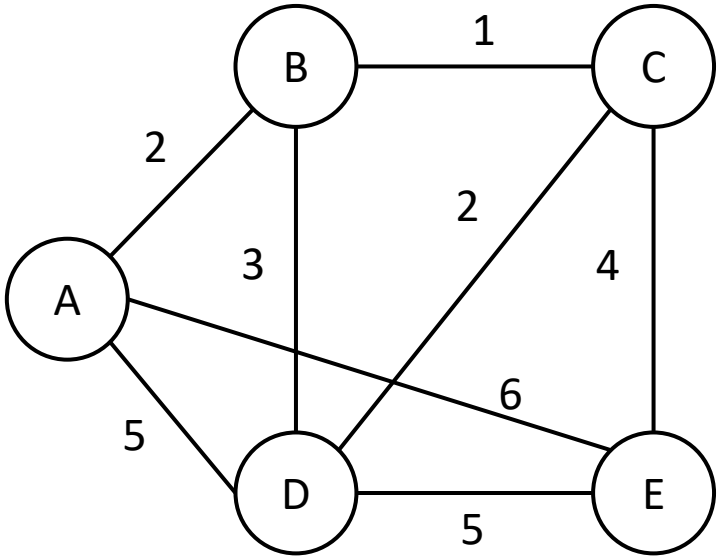
What if the Graph Have Negative Weights?

- **Update the distance of visited nodes.**
  - Bellman–Ford algorithm

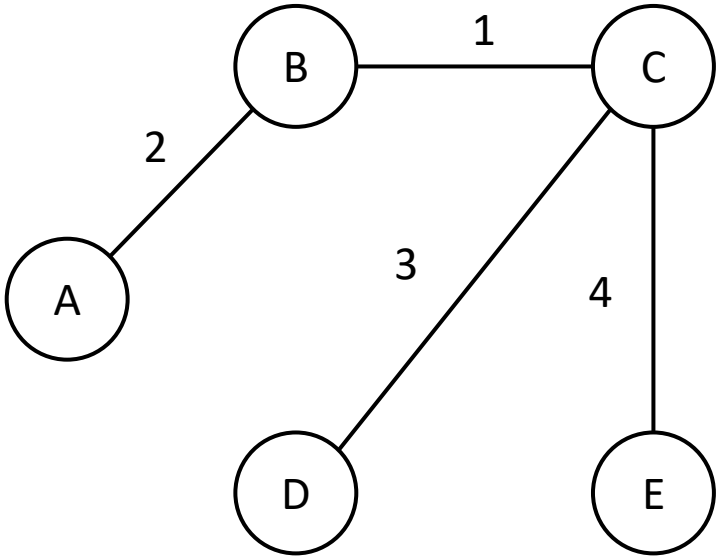
Minimal Spanning Tree

- **A spanning tree of a connected graph is a subset of the edges that:**
  - Connects all the vertices of the graph
  - Contains no cycles (so it's a tree)
- **A minimum spanning tree is a spanning tree whose sum of edge weights is minimal among all possible spanning trees of the graph.**

Example



Undirected Weighted Graph

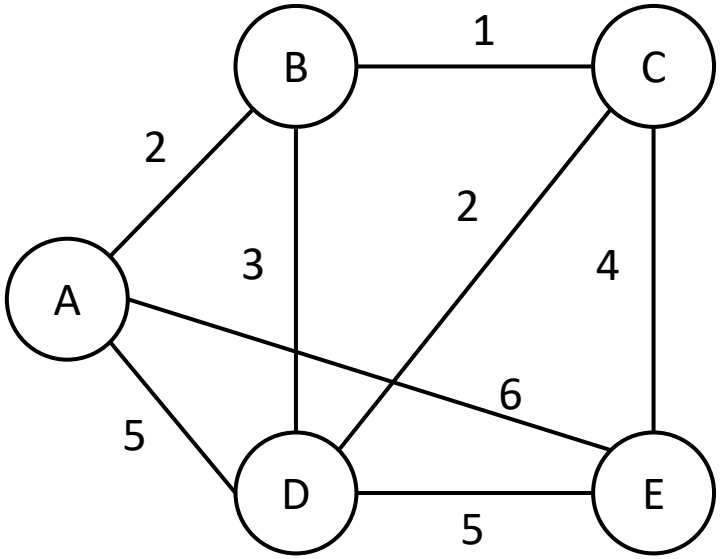


Minimal Spanning Tree

Kruskal's Algorithm (greedy)

- **Steps:**
  - Sort all edges by weight (from smallest to largest).
  - Start with an empty set of edges.
  - Add edges one by one, skipping any that would form a cycle.
  - Stop when you have n−1 edges.
- **Efficient for sparse graphs.**
- **Uses Union-Find (Disjoint Set Union) to detect cycles.**

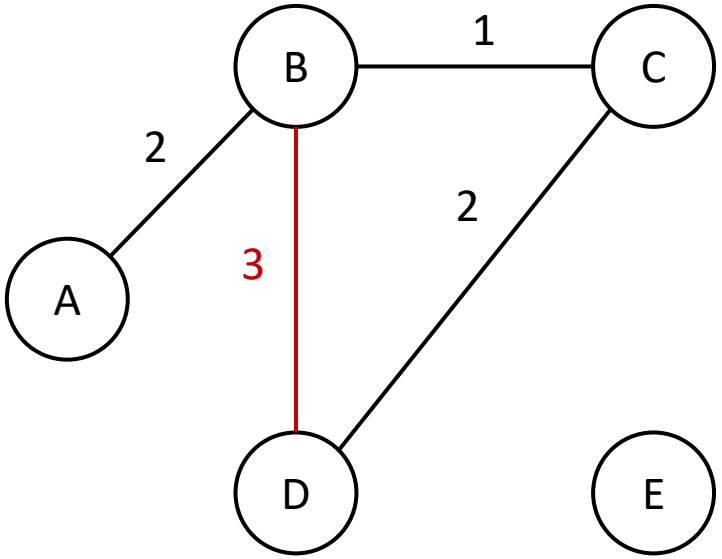
Sort Edges by Weight



Undirected Weighted Graph

Weight	Edges
1	B-C
2	A-B, C-D
3	B-D
4	C-E
5	A-D, D-E
6	A-E

Add Edges One by One and Avoid Cycle



Weight	Edges
1	B-C
2	A-B, C-D
3	B-D
4	C-E
5	A-D, D-E
6	A-E

How to Detect Cycles?

- **Recognize connected components**
  - Union-find algorithm
- **Detect cycles for each connected component via DFS-based method:**
  - Start a Depth-First Search (DFS).
  - Keep track of the parent node (the vertex you came from).
  - If you encounter a neighbor that has already been visited and it is not the parent, then a cycle exists.

Union-Find Algorithm

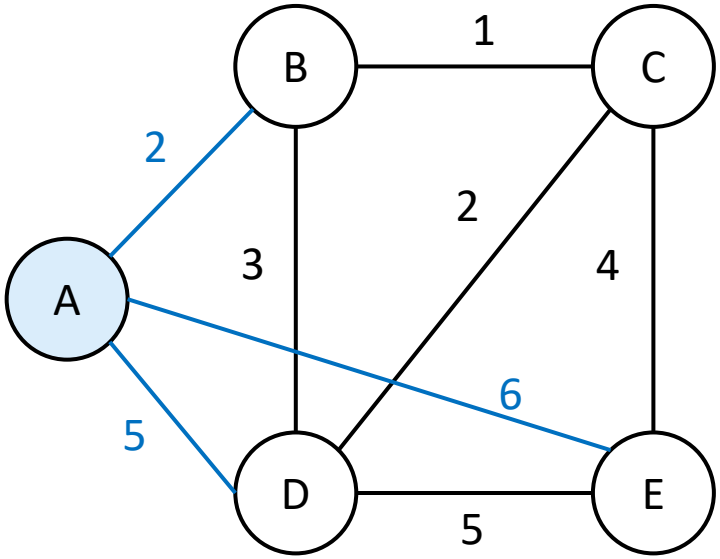
- Find: Determine which set or connected component an element belongs to.
- Union: Merge two sets into a single set.

Edge	Connected Component
{A, B}	{A, B}
{A, C}	{A, B, C}
{B, C}	{A, B, C}
{B, H}	{A, B, C, H}
{D, E}	{A, B, C, H} {D, E}
{D, F}	{A, B, C, H} {D, E, F}
{E, F}	{A, B, C, H} {D, E, F}
{G, H}	{A, B, C, H, G} {D, E, F}

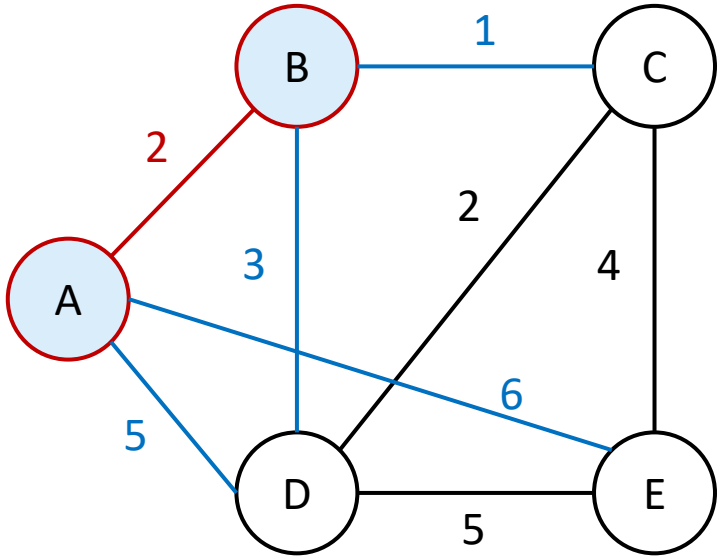
Prim's Algorithm (greedy)

- **Steps:**
  - Start from any vertex.
  - Repeatedly add the smallest edge that connects a visited vertex to an unvisited vertex.
  - Continue until all vertices are included.
- **Efficient for dense graphs.**
- **Use a priority queue (min-heap) to pick edges quickly.**

Example

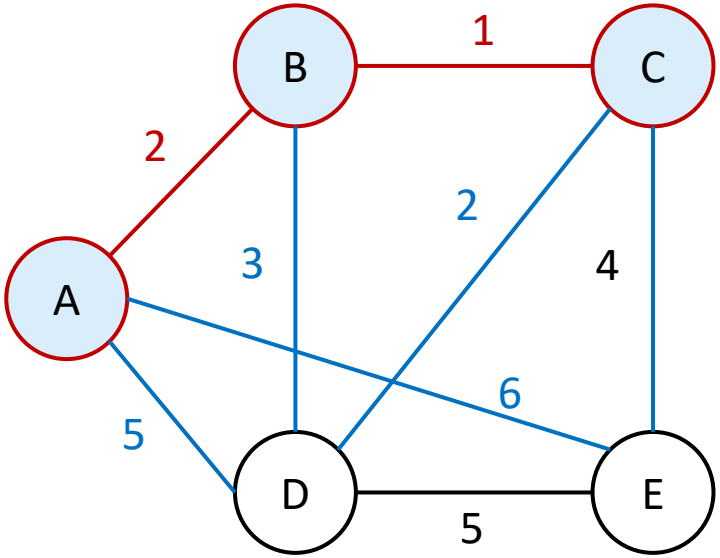


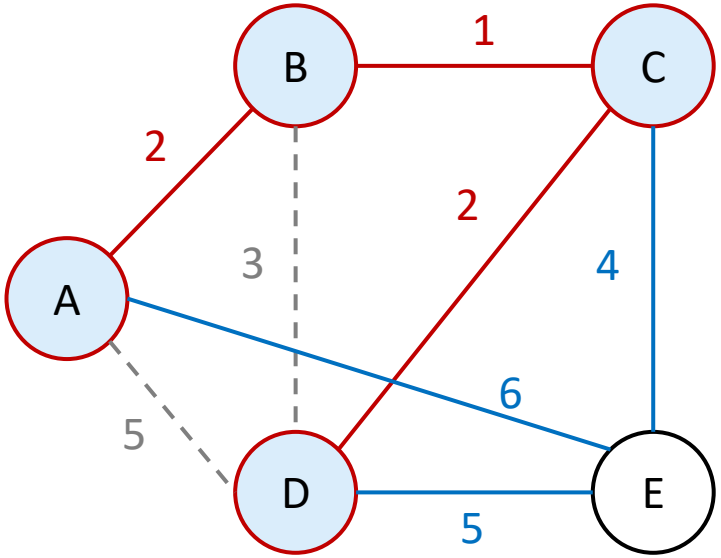
Start from Vertex A

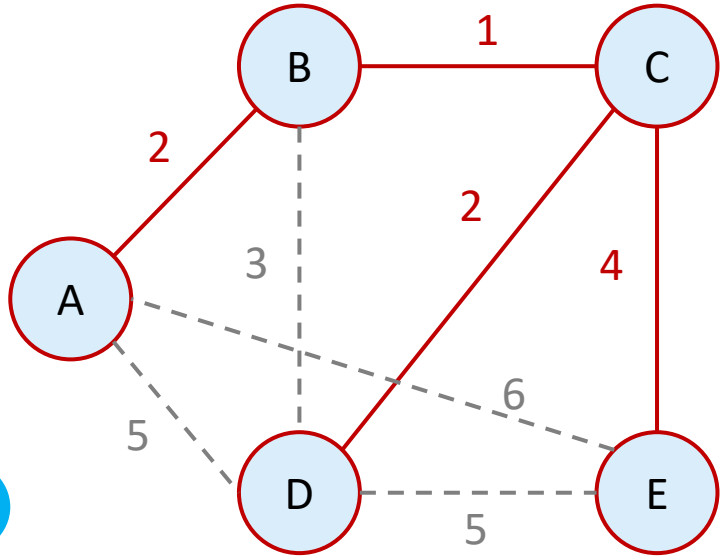


Add the smallest edge
from a visited vertex to an unvisited vertex

Example







Exercise: Vibe Coding

- Design experiments to benchmark the performance of naïve BST, AVL tree, and read-black tree.
- Design experiments to benchmark the performance of Kruskal's algorithm and Prim's algorithm.