BFS and DFS: Time and Space complexity

	Best	Average	Worst
BFS	O(V+E)	O(V+E)	O(V+E)
DFS	O(V+E)	O(V+E)	O(V+E)

Space is O(V) for both.

Topological Sort

- In top sort, vertices are sorted on their indegrees
- How can we topologically sort a graph given its edge list / adjacency list?
  - Kahn's Algorithm
    - count and decrement indegrees
  - DFS / BFS

Kahn's Algorithm

- Add all nodes with in-degree 0 to a queue
- while queue is not empty
  - remove a node from queue
  - For each outgoing edge from the node, decrement in-degree of destination node by 1
  - If in-degree of a destination node becomes 0, add to queue
- Cannot topological sort on undirected graphs
- Cannot top-sort on directed graphs with cycles

Topological sort DFS and BFS intuition

- Traverse graph to find two types of nodes
  - a) Nodes with no outgoing edges go last
  - b) Nodes with no incoming edges go first
- Nodes "in the middle" are added to solution after their outedge vertices have been added to the solution

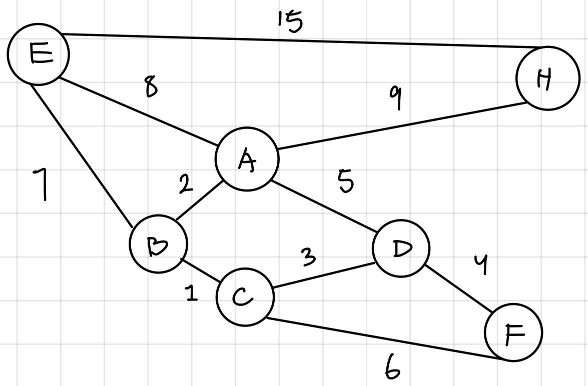
Topological Sort Time Complexity

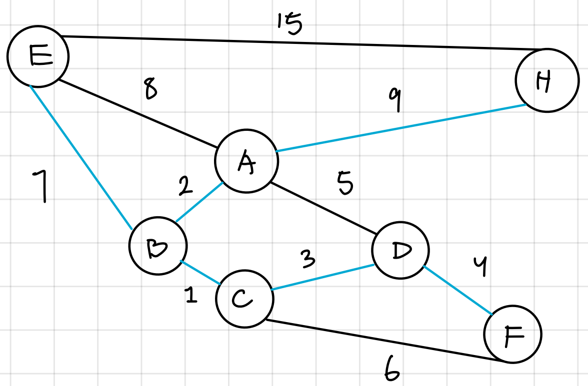
- O(V+E) time complexity
- O(V) space for queue / stack

Minimum Spanning Tree

MST Intuition

- ex. City planner assigned to upgrade roads
  - Every landmark in the city should be connected by upgraded roads
  - Total cost of upgrade should be minimized





This is the MST!
Total Weight: 28

Note:
1. No cycles
2. Heaviest edge excluded
3. Count (E) = Nodes − 1
4. There is no other possible solution (because of unique weights)

3 well-known MST algorithms

- 1. Prim's
- 2. Kruskal's
- 3. Boruvka's

Prim's Algorithm

- Pick a random vertex to start from, and add to visited list
- From visited nodes, pick edge with minimum weight, and visit its destination
- For multiple valid choices, pick any minimum weight edges
- Used to implement Heaps and Lists
- Time Complexity: $O(V^2)$ for Adj. Matrix

  $O(V\log V + E\log V)$ for Adj. Lists

Kruskal's Algorithm

- To start, pick minimum weight edge
  - Tiebreak if multiple valid choices
- Then repeat picking minimum weight edge
- If minimum weight edge is connecting two nodes in the same tree, ignore
- Used to implement Disjoint Set
  - List of disjoint sets
  - Uses union-find
- Time complexity: $O(E \log E)$

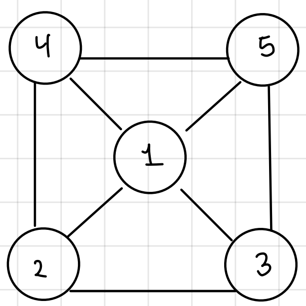
MST Applications

- Constructing trees for broadcasting in computer networks
- Curvilinear feature extraction in computer vision
- Cluster analysis
  - a. clustering points in the plane
  - b. graph-theoretic clustering
  - c. clustering gene expression data

Graph Representation

- Internally represent the edge list
  - Arrays for each vertex
  - Linked Lists
  - Trees
- Second common way to represent a graph is an adjacency matrix

Graph



Adj. List

1: 2, 3, 4, 5
2: 1, 3, 4
3: 1, 2, 5
4: 1, 2, 5
5: 1, 3, 4

Adj. Matrix

	1	2	3	4	5
1	F	T	T	T	T
2	T	F	T	T	F
3	T	T	F	F	T
4	T	T	F	F	T
5	T	F	T	T	F

- undirected graphs is symmetrical around the diagonal
- can get rid of the top or bottom half
- If the edges are weighted, we can store the weight as a double for adj. lists
  - ex. 1: (2, 1), (3, 1), (4, 1), (5, 1)
    2: (1, 1), (3, 0.5), (4, 0.5)
- For missing edges, we can use ∞ / $-\infty$ or simply nil.

- For now, we don't consider self-loops
  - edges that start and terminate at the same vertex

Graph Representation: When to use what?

Adjacency Matrices

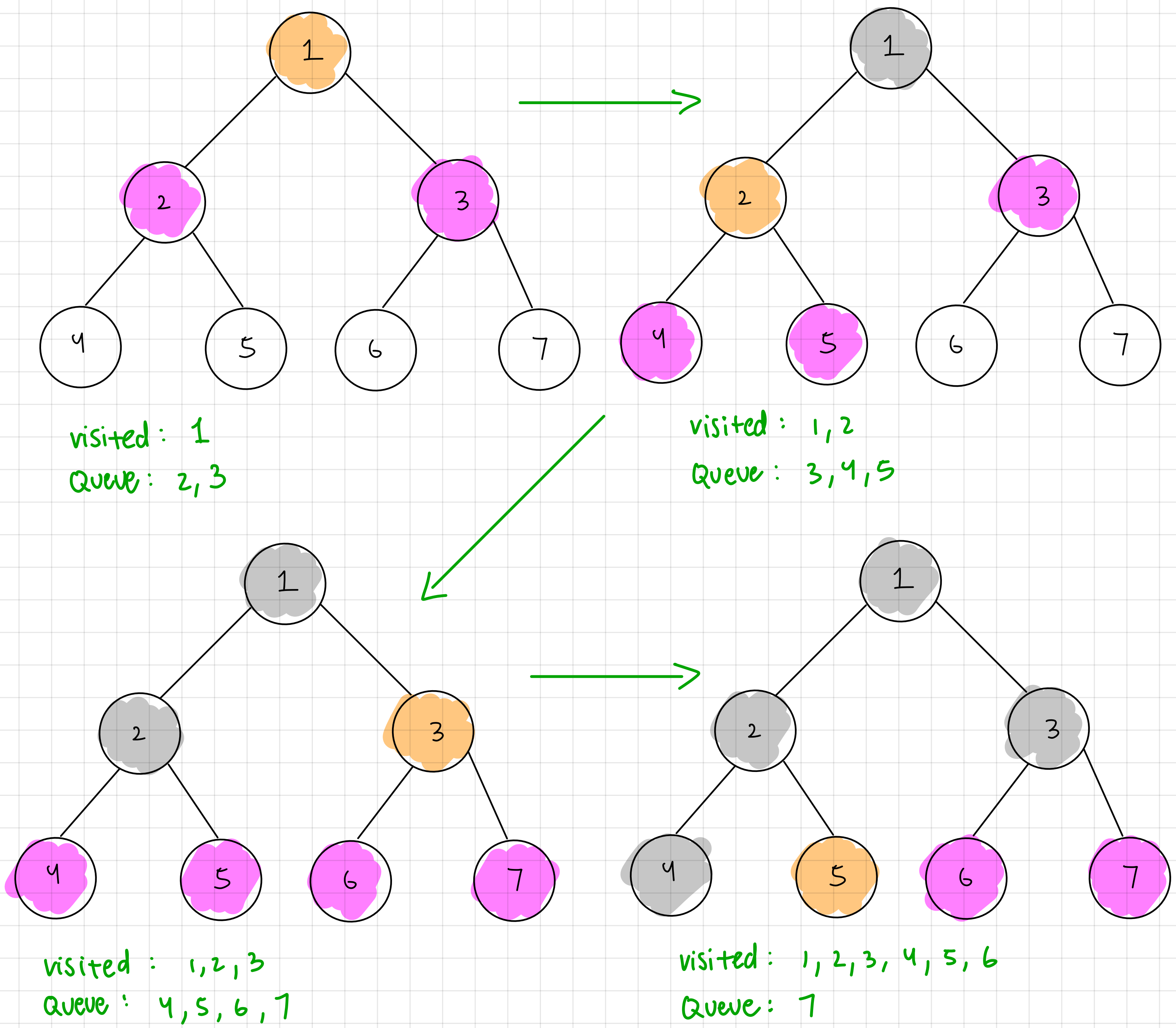
- Use more memory $O(n^2)$
- Fast lookup and checks for presence of edges $O(1)$
- Slow to iterate over all edges
- Slow to add/delete a node $O(n^2)$
- Fast to add a new edge $O(1)$

Adjacency List

- Memory usage depends more on number of edges, not nodes
- helps when the graph is sparse
- slow lookup and checks for presence of edges $O(k)$
- Faster to iterate over all edges
- Fast to add/delete a node
- Fast to add a new edge $O(1)$
- If density (edge/nodes^2) goes over $\frac{1}{64}$ (for 32 bit computers), use an adjacency matrix

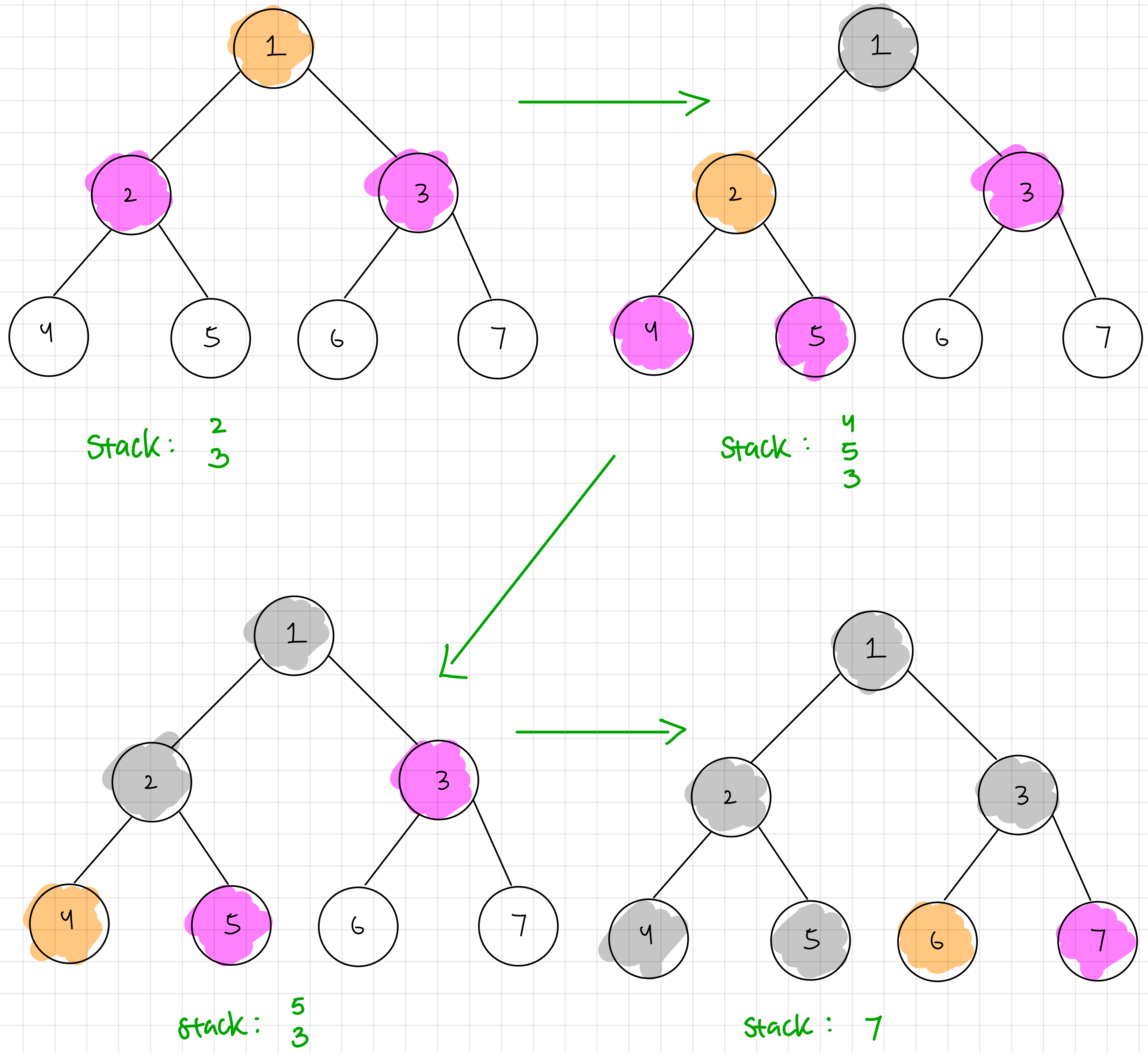
BFS Intuition

- Basic Idea : Visit each node , one level at a time
  - Can use a queue to track which node to visit next
- First , visit the starting node , then mark as visited
- Second , discover all neighbors of this node , then add to queue to visit them next
- While the queue is not empty , dequeue the next element and repeat step 2
- Discover all neighbors of this node , then add them to the queue to visit them next



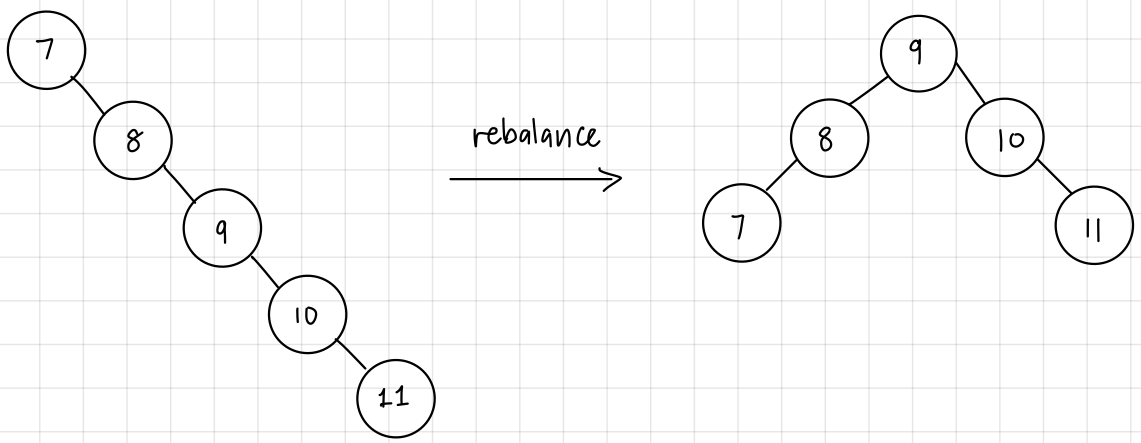
DFS Intuition

- First , visit the starting node , then mark as visited
- Second , discover all neighbors of this node , then add to stack to visit them next
- Pop the stack , and repeat step 2.



BST Problems

- Worst case time complexity for the following in a BST:
  - Traversal
  - Insertion
  - Deletion
  - $\Theta(n)$
    - In the case of skewed BSTs
- We can avoid our BSTs becoming skewed by rebalancing/rotations



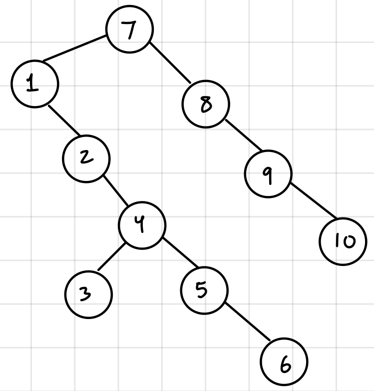
Can be done via a $\Theta(n)$ time algorithm, without rotations

- Height = order of N for skewed trees for N nodes
- Height = order of logn for balanced trees for n nodes
  - We want balanced trees to have $\Theta(\log n)$ time complexity

Red-Black Tree properties

- Every node is either red or black
- The root is black
- All NIL nodes are considered black
- A red node does not have a red child
- Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes

BST Review

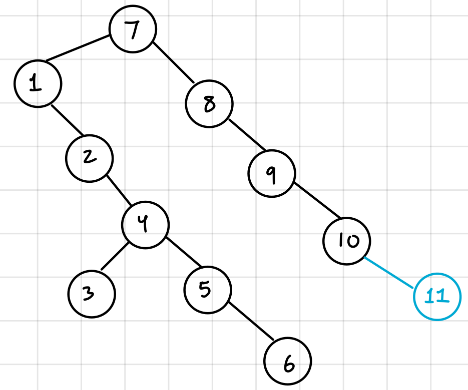


- Traversal / Searching
- Insertion
- Deletion

BST Traversal

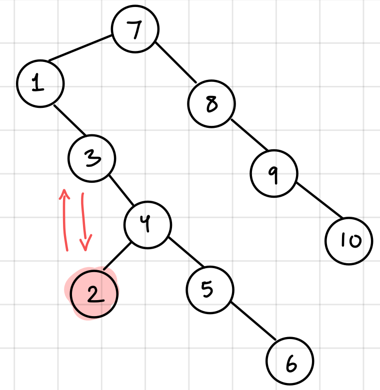
- Left = lesser than parent
- Right = Greater than parent
- If does not exist in tree, end search after traversing

BST Insertion



- We can have duplicate values
  - Approach 1: Add duplicate values as right / left child
    - Use right if you want "stability" during in-order traversals
  - Approach 2: Augment nodes with counts

BST Deletion



- Deleting leaf nodes is simple, similar to heaps
- Deleting within the tree is more difficult
- If we want to delete 2
  - We want to find the smallest value greater than 2 to minimize work
    - We swap 2 with that value and delete 2 as a leaf node

BST Time Complexity

Op	Best	Average	Worst
Search	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
Insert	$\Theta(1)^*$	$\Theta(\log n)$	$\Theta(n)$
Delete	$\Theta(1)^*$	$\Theta(\log n)$	$\Theta(n)$

Trees

- Trees in general have no special constraints