4_recursive_ds

June 24, 2025

1 Recursive Data Structures

1.1 Outline

- Trees
- Anatomy, tree traversal methods
- Binary Search Trees
- Graphs
- Nearest Neighbor Problem

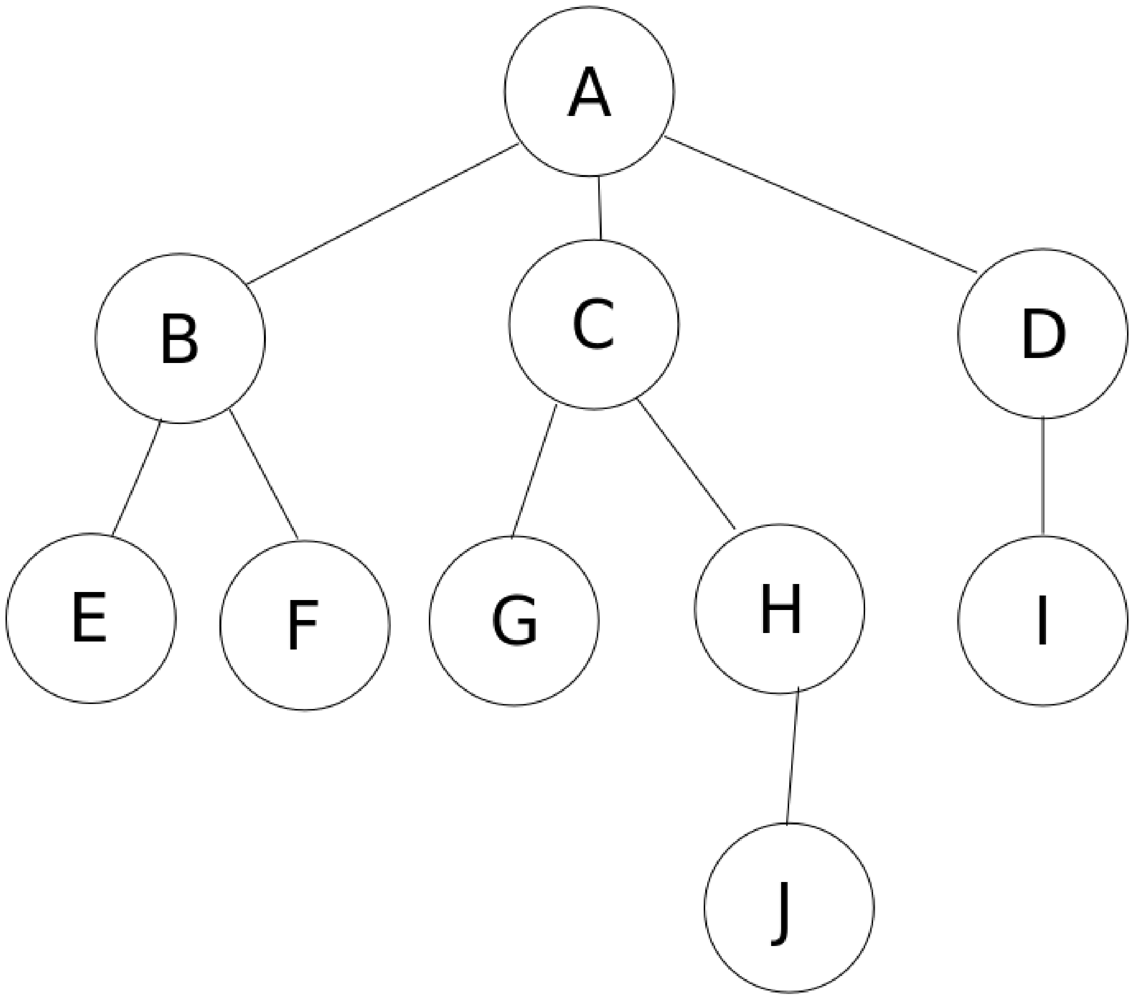
2 Trees

2.1 Introduction to Trees

- Not all data has a natural linear order. Organization charts and file storage systems have a *hierarchical structure*, in which each entity is linked to multiple entities below it
- This type of data is represented using a *tree*. A tree is either
  - Empty
  - Has a *root value* connected to any number of other trees, called *subtrees*
- We draw the root at the top of the tree

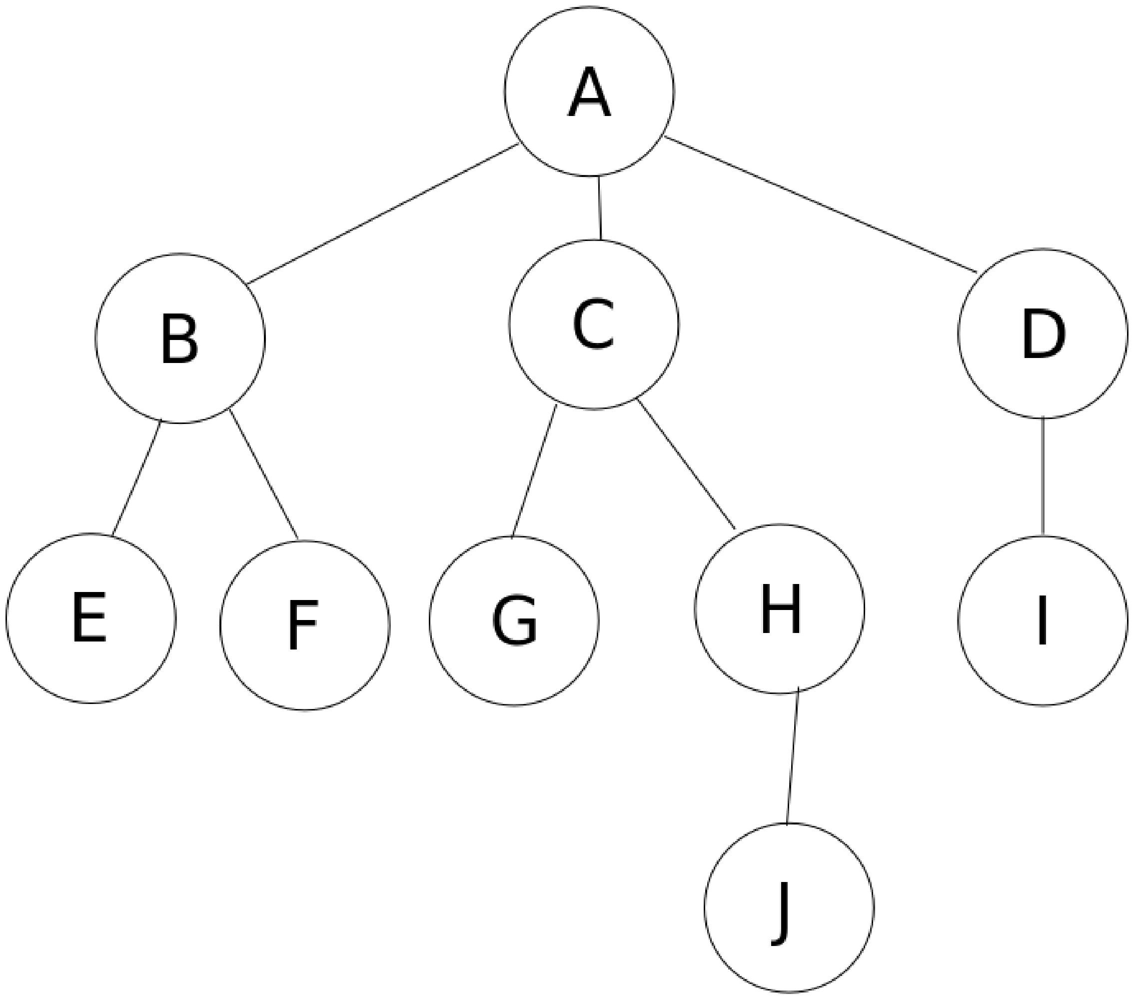
2.2 Anatomy of a Tree

- The *size* of a tree is the number of values in the tree
- A *leaf* is a value with no subtrees. The leaves of this tree are labeled E, F, G, J, and I
- The *height* of a tree is the longest path from its root to its leaves. The height of this tree is 4



2.3 Anatomy of a Tree

- The *children* of a value are all values directly connected underneath that value. The children of A are B, C, and D
- The *descendants* of a value are it's children, the children of its children, etc. This can be defined recursively
- The *parent* of a value is the value immediately above and connected to it. The parent of H is C
- The *ancestors* of a value are its parent, the parent of its parent, etc.



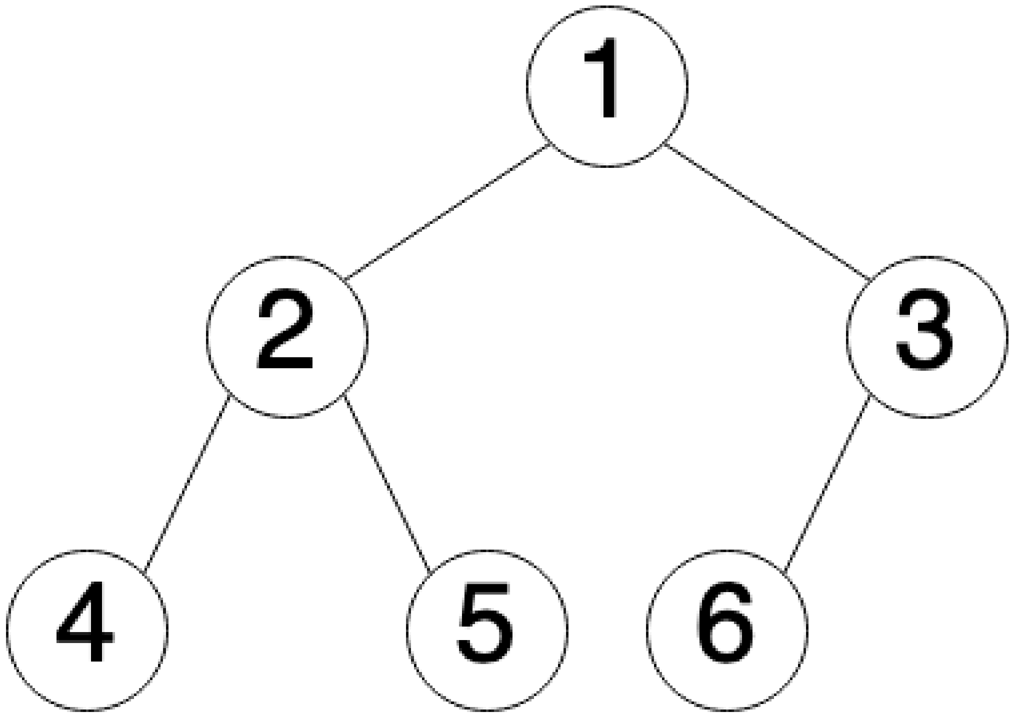
2.4 Tree Traversal Methods

- Linear data structures only have one logical way to traverse them. Trees can be traversed in different ways
- We'll look at the following methods of tree traversal and their applications
  - *Depth First Search* (DFS): Inorder, Preorder, and Postorder traversal
  - *Breadth First Search* (BFS)
- Note there are other methods not covered

2.5 DFS: Inorder Traversal

1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree

Result: 4 2 5 1 6 3



2.6 DFS: Inorder Traversal Code

Let's look at the code to do this

[1]:
```
class Node:
  """Tree class
  """
  def __init__(self, key):
    self.left = None
    self.right = None
    self.val = key

def print_inorder(root):
  if root:
    print_inorder(root.left)
    print(root.val, end = " ")
    print_inorder(root.right)
```

2.7 DFS: Inorder Traversal Code

[2]:
```
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
print_inorder(root)
```

```
4 2 5 1 6 3
```

In binary search trees (next section), inorder traversal gives the nodes in a non-decreasing order.

2.8 DFS: Inorder Traversal Complexity

Time complexity

- Each node is visited exactly once. The work done at each node is constant. $O(n)$

Space complexity

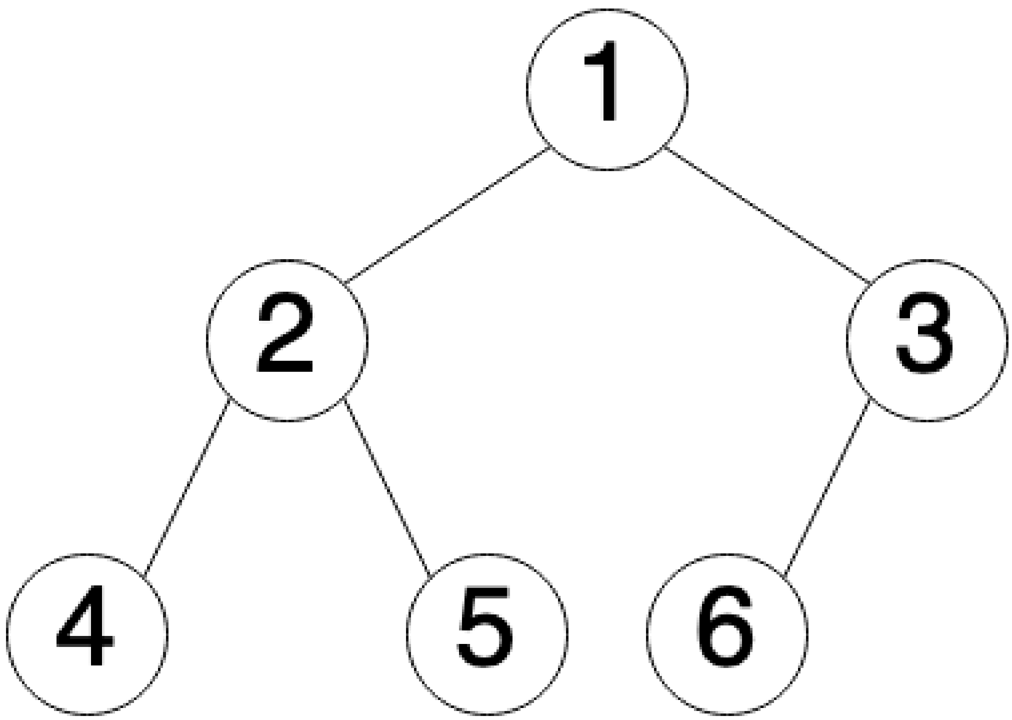
- Dependent on the maximum depth of the recursion, which is the height of the tree. $O(h)$

2.9 DFS: Preorder Traversal

1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree

Result: 1 2 4 5 3 6

Preorder traversal is used to create a copy of the tree

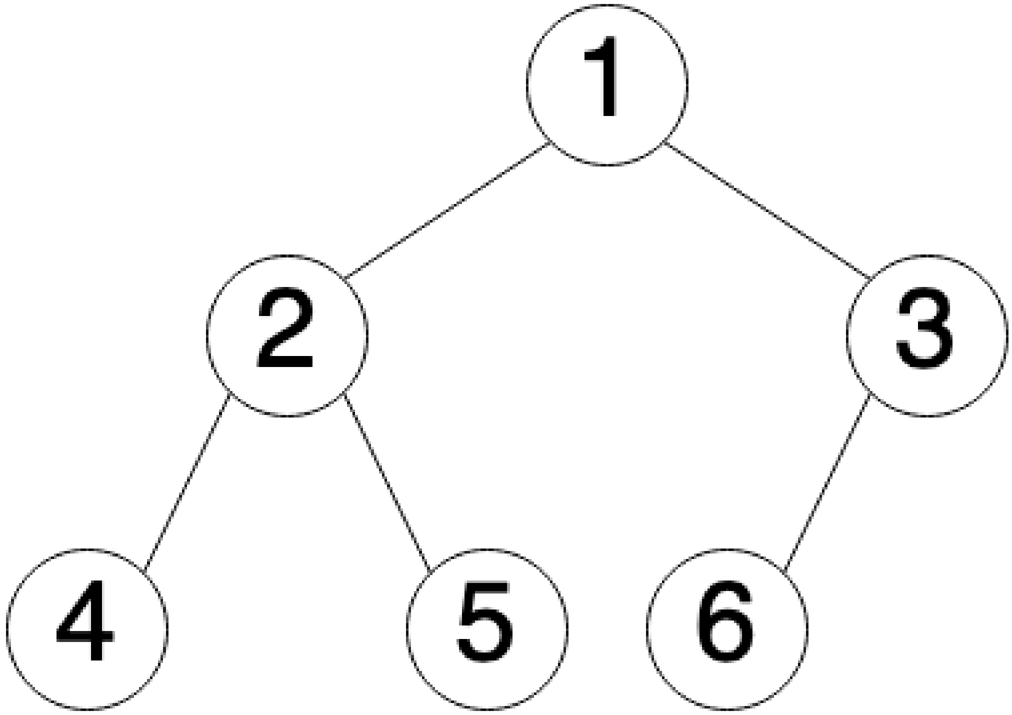


2.10 DFS: Postorder Traversal

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root

Result: 4 5 2 6 3 1

Preorder traversal is used to delete subtrees. (why?)



2.11 BFS

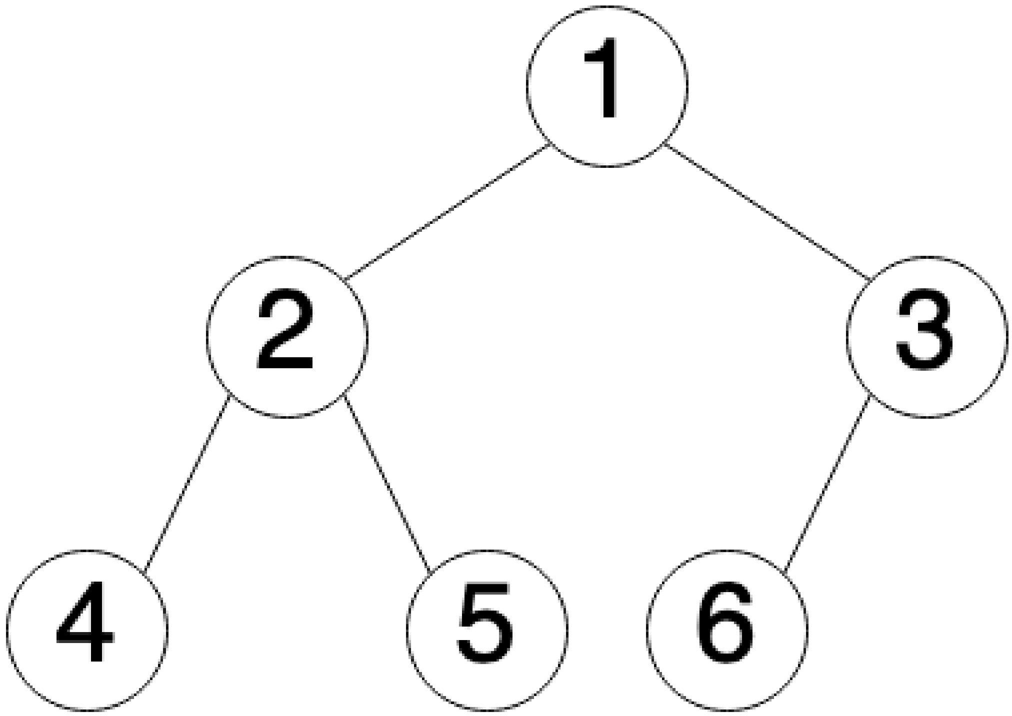
BFS (or Level Order Traversal) traverses nodes present in the same level before traversing the next level

1. For each node

- The node is visited
- The child nodes are enqueued in a FIFO queue

1. First node is dequeued
2. Child nodes are enqueued
3. Repeat until the queue is empty

Result: 1 2 3 4 5 6



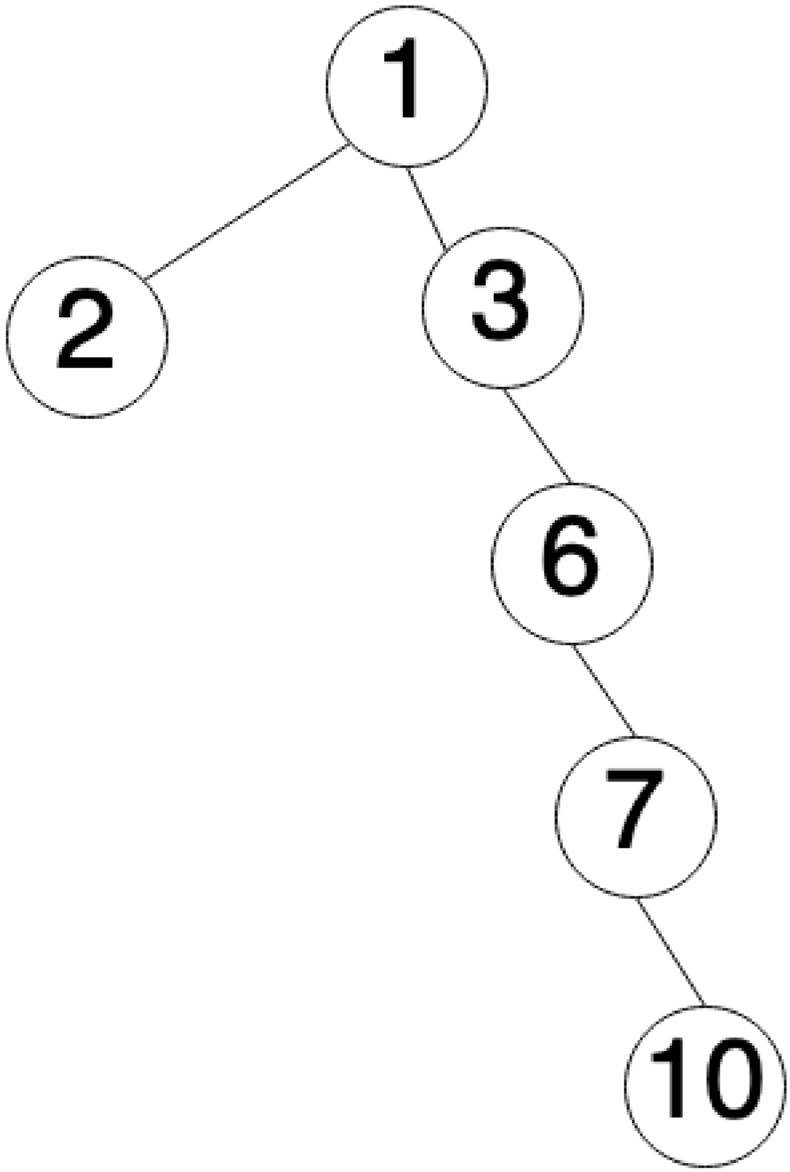
3 Binary Search Trees

3.1 BST Definitions

- You can think of a BST as a sorted tree
- A *binary tree* is a tree in which every item has at most two subtrees
  - The tree used in illustrating DFS and BFS methods is a binary tree
- A binary tree is a *binary search tree property* if its value is greater than or equal to all items in the left subtree
- A binary tree is a *binary search tree* if every item in the tree satisfies the binary search tree property

3.2 BST Efficiency

- Consider the BST on the right. Verify that it is a BST.
- The worst-case run time is $O(h)$, h being the height of the tree
  - So the tree on the right is $O(n)$
- A tree of height h can have at most $2^h - 1$ nodes. So we need at least $\log n$ height to store all of them.
  - So if the tree was balanced, then it would be $O(\log n)$

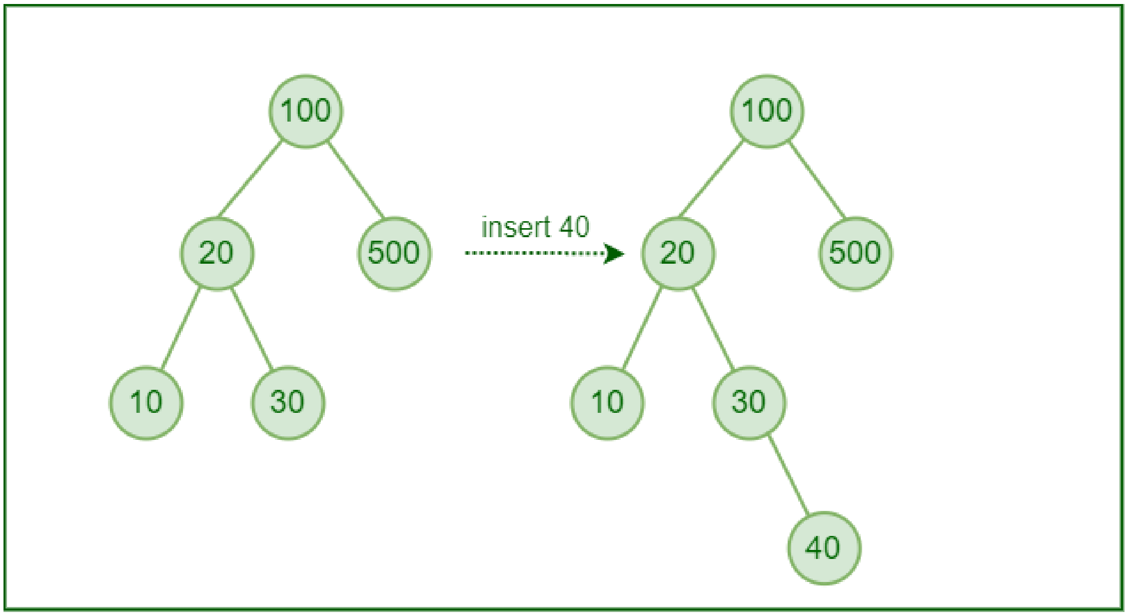


3.3 BST Efficiency

- Convince yourself that for a balanced BST the search, insert, and delete Big-O is all $O(\log n)$
- Ensuring that a tree is balanced is important
  - Red-Black trees (not covered) are trees that balance themselves
  - You may also be interested in B-trees, which are used in databases

3.4 Live Coding

Given a BST, insert a new node in this BST.

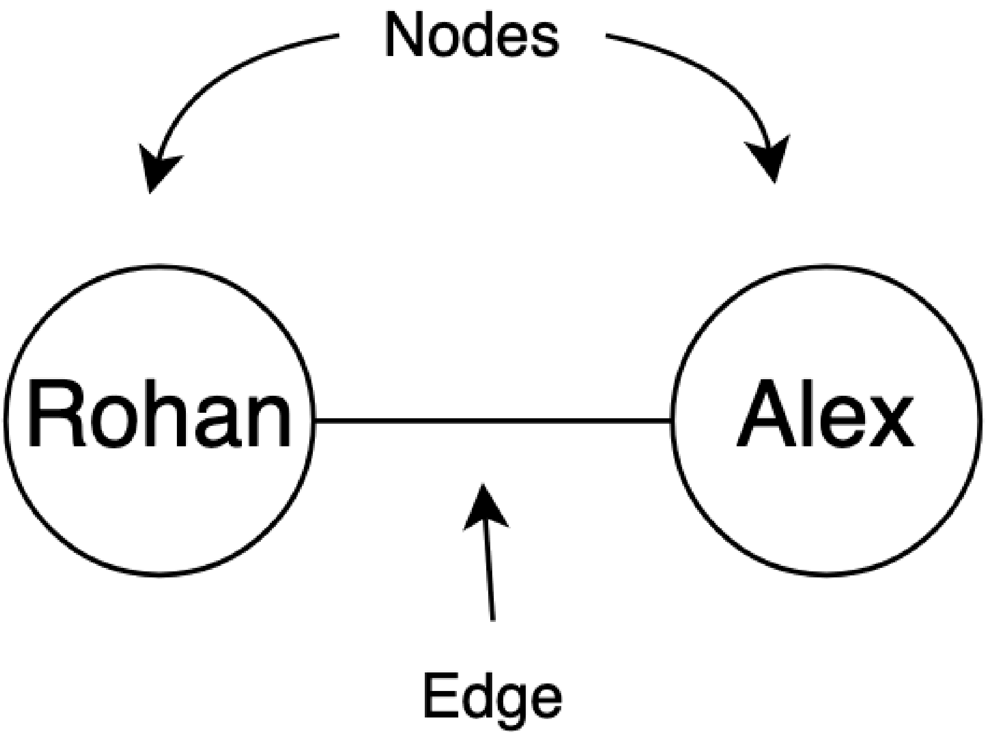


[ ]:
```
# Your code here
```

4 Graphs

4.1 Introduction

- We looked at lists and trees, which represent linear and hierarchical relationships respectively
  - But many relationships are neither
  - Friend networks, internet connections, flight connections
- Graphs consist of two parts, *nodes* and *edges*
  - A node connected to another is a *neighbor*



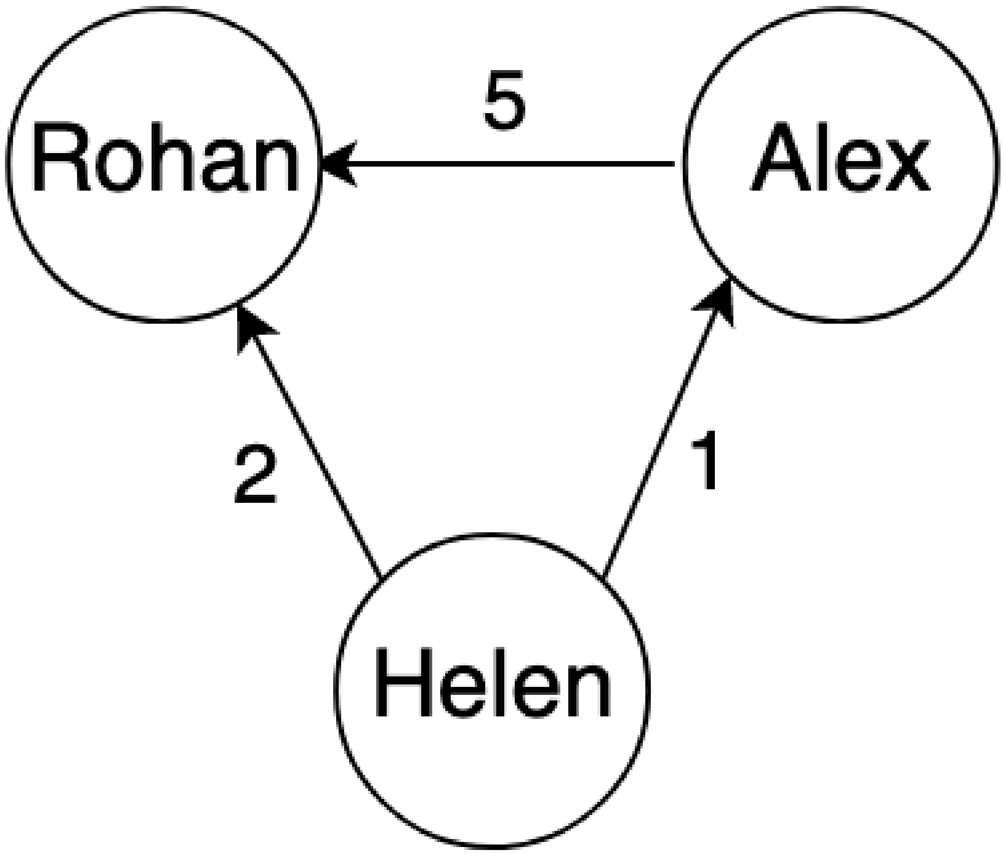
4.2 Types of Graphs

There are directed and undirected graphs to represent different situations

- Friendships: undirected
- Twitter followers: directed
- Who owes who money: directed
- Note that trees are special cases of directed graphs

Graphs can also be weighted, to differentiate strengths between nodes

There are two questions we ask about graphs: Is there a path from node A to B? What is the shortest path from node A to B? BFS answers both!



4.3 BFS of Graphs

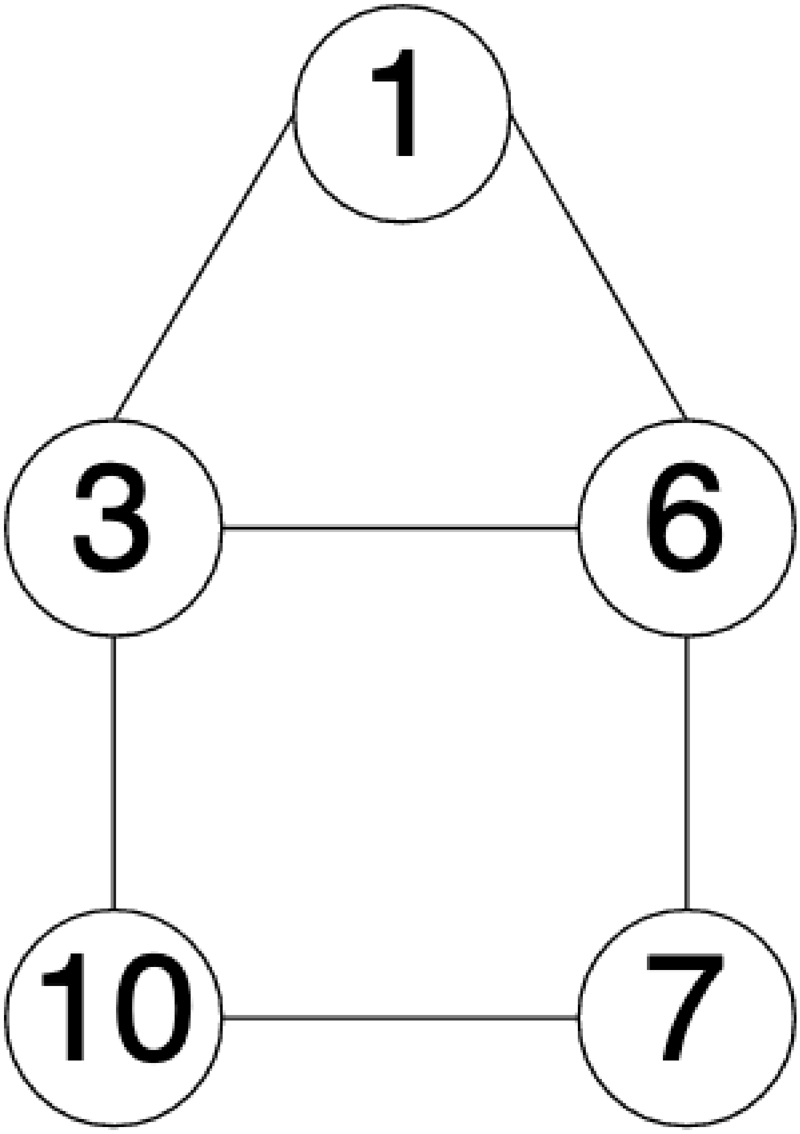
- *Breadth First Search* (BFS) searches graph for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level
  - If there are multiple nodes meeting the criteria, then BFS will also find the nearest node!
- The issue is that graphs contain *cycles*, so we may visit the same node more than once
  - Let's split edges into visited and not visited
- We use a list to keep track of visited nodes
- All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue
- Is BFS a recursive or iterative graph search method?

4.4 BFS Example

- Let's traverse a graph with BFS starting at node "1"
- Visited list and queue start as empty

Visited: [, , , ,]

Queue: [, , , ,]

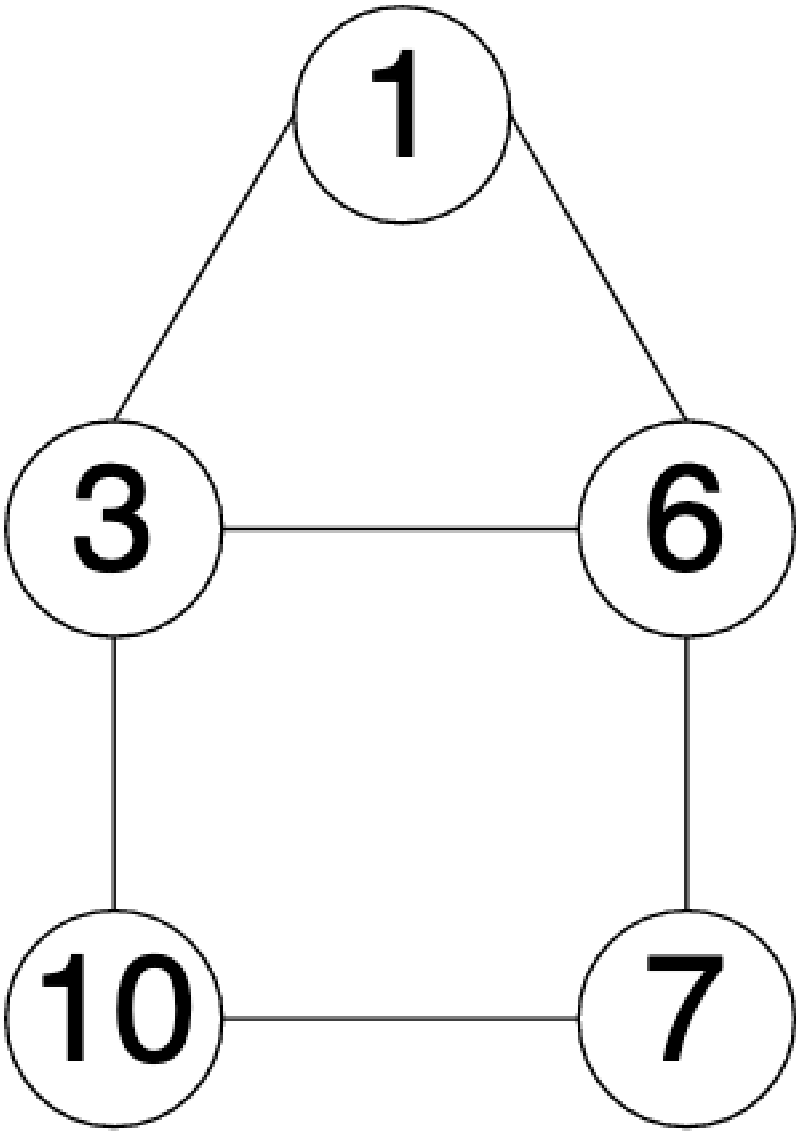


4.5 BFS Example

- We're at node 1, so we push it onto the visited list and push it onto the queue

Visited: [1, , , ,]

Queue: [1, , , ,]

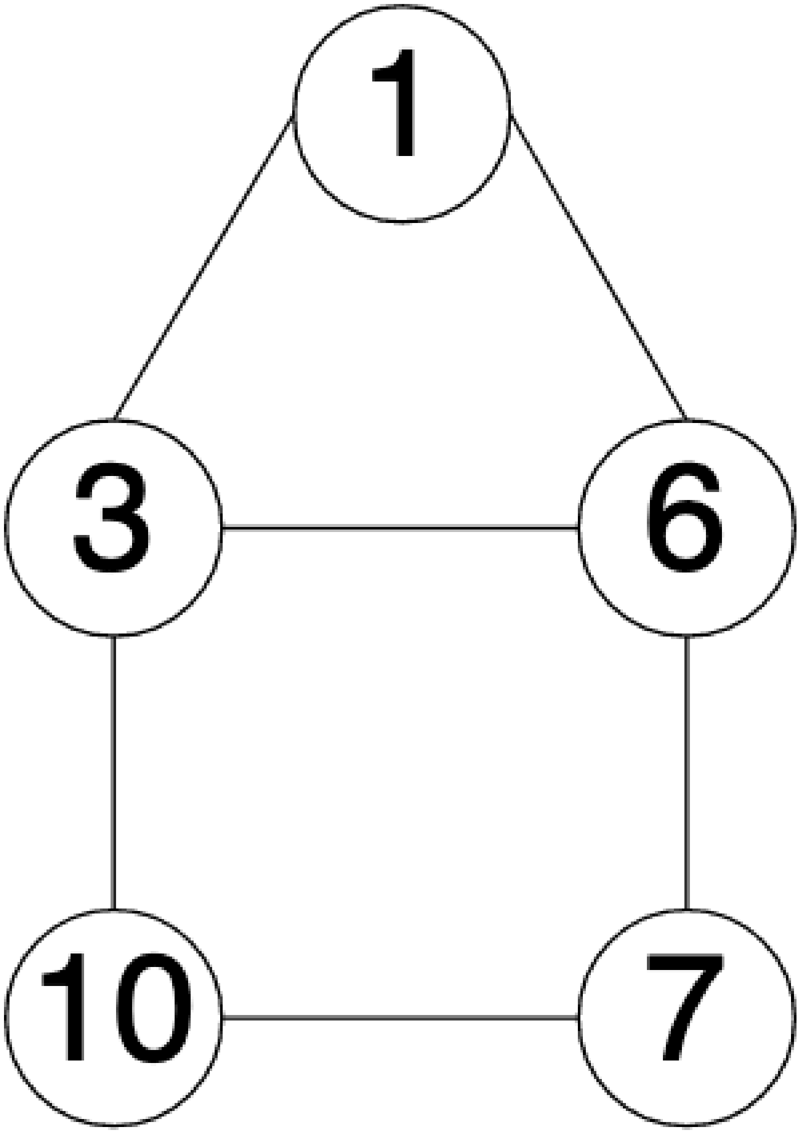


4.6 BFS Example

- Now we visited 1, so it is dequeued.
- At the first level away from node 1, there is 3 and 6.
- We visit 3 and 6, but we have not visited any of it's neighbors (other than 1), so 3 and 6 are enqueued.

Visited: [1, 3, 6, ,]

Queue: [3, 6, , ,]

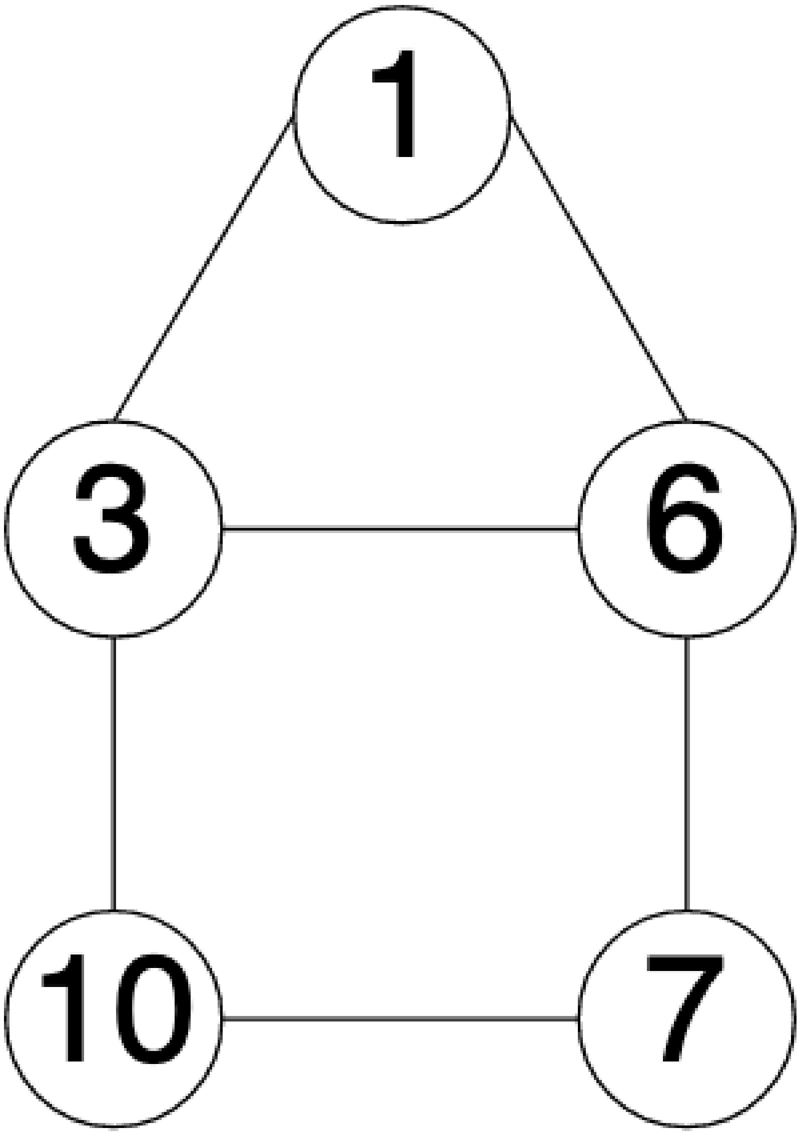


4.7 BFS Example

- Visit the neighbors of node 3, so we dequeue it
- But we need to enqueue 10, because we haven't visited its neighbors

Visited: [1, 3, 6, 10,]

Queue: [6, 10, ,]

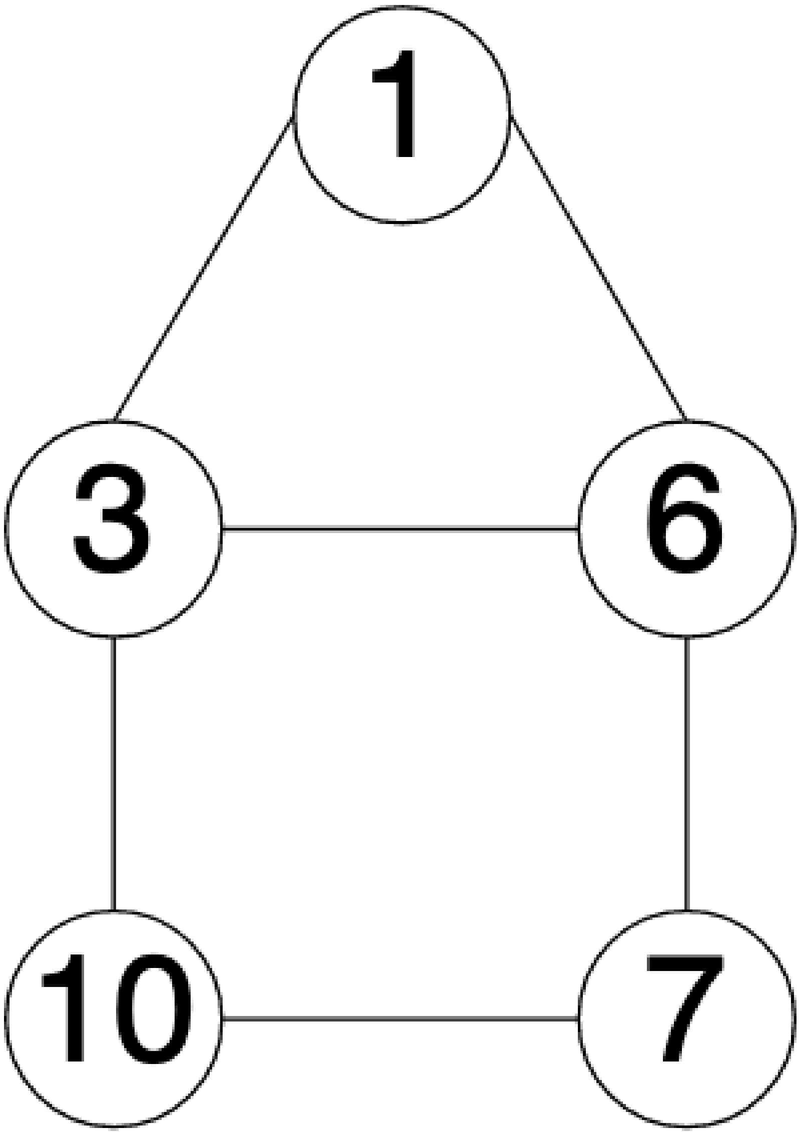


4.8 BFS Example

- Visit the neighbors of node 6, which is just 7, so we dequeue it
- But we need to enqueue 7

Visited: [1, 3, 6, 10, 7]

Queue: [10, 7, ,]

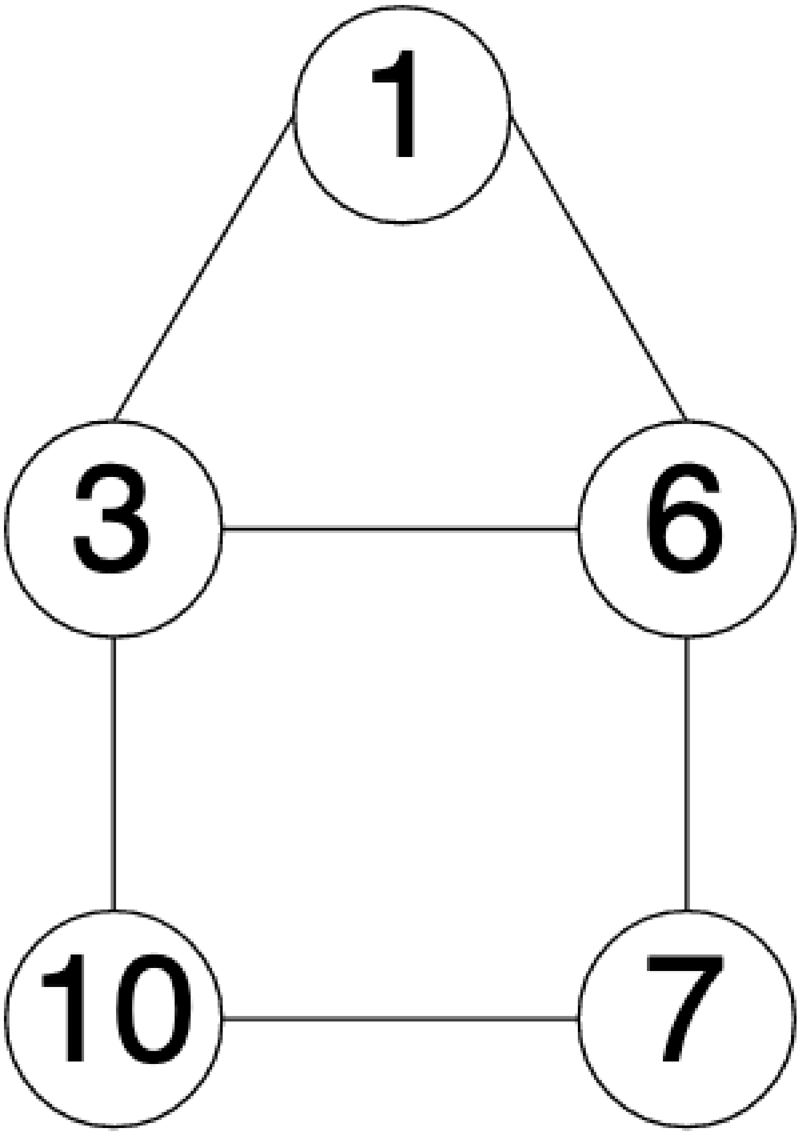


4.9 BFS Example

- Visit the neighbors of node 10, and dequeue 10
- But we already visited those nodes, so the visited list does not change

Visited: [1, 3, 6, 10, 7]

Queue: [7, , , ,]

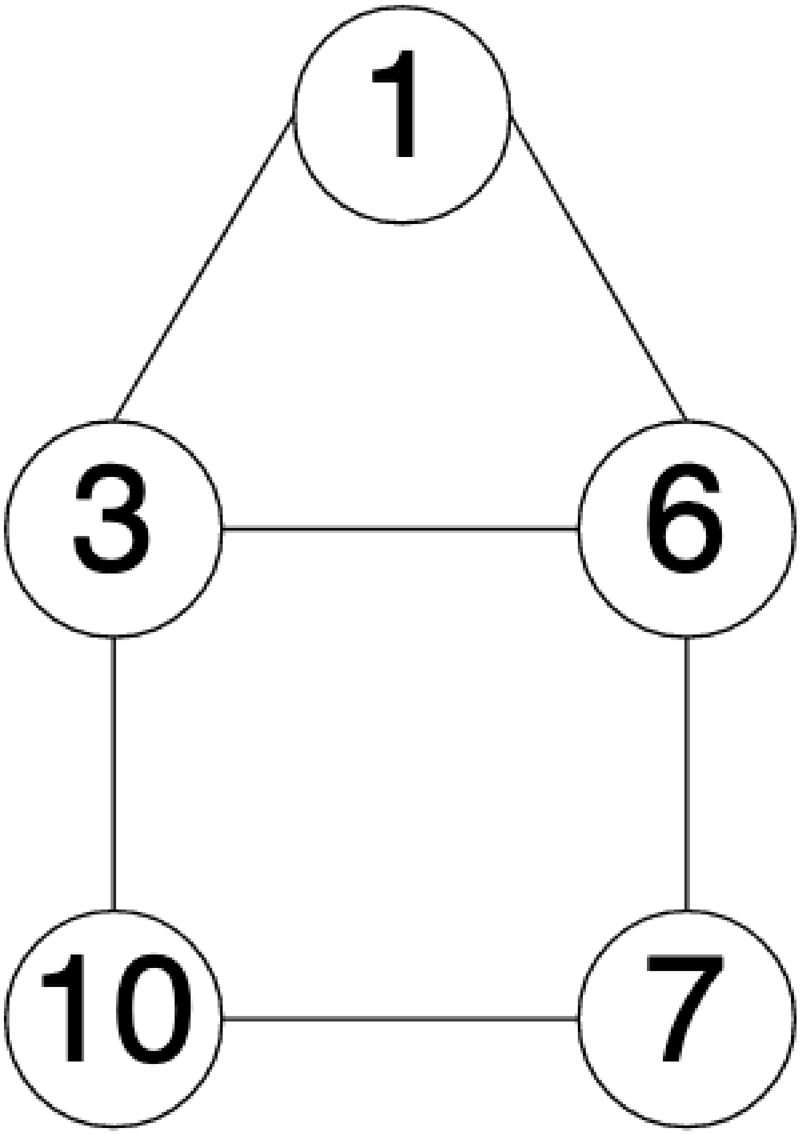


4.10 BFS Example

- Visit neighbors of 7, which are also all visited
- The queue is empty, so the algorithm ends

Visited: [1, 3, 6, 10, 7]

Queue: [, , , ,]



4.11 Time and Space Complexity of BFS

- Each edge and each node must be visited once, so the time complexity is $O(n + e)$
- Since we need to store each node of the graph by the end of the algorithm, the space complexity is $O(n)$

4.12 Implementing Graphs and BFS

We can represent graphs using the *adjacency list* representation

- Other options include adjacency matrix or using a Python library

[3]:
```
from collections import deque
```

```python
class Graph:
  def __init__(self):
    self.graph = {}

  def add_edge(self, vertex, neighbors):
    self.graph[vertex] = neighbors
```

4.13 Implementing Graphs and BFS

[4]:
```python
def bfs(graph, start):
  visited = set()
  queue = deque([start])

  while queue:
    current_vertex = queue.popleft()

    if current_vertex not in visited:
      # Process the current vertex
      print(current_vertex, end=' ')
      visited.add(current_vertex)

      # Enqueue unvisited neighbors
      for neighbor in graph.graph.get(current_vertex, []):
        if neighbor not in visited:
          queue.append(neighbor)
```

4.14 Implementing Graphs and BFS

[5]:
```python
# Represent graph from above
ex_graph = Graph()
ex_graph.add_edge(1, [3, 6])
ex_graph.add_edge(3, [10, 6])
ex_graph.add_edge(6, [3, 7])
ex_graph.add_edge(10, [3, 7])
ex_graph.add_edge(7, [10, 6])

# Perform BFS starting from vertex 1
bfs(ex_graph, 1)
```

```
1 3 6 10 7
```

4.15 Recursive Graph Search: Preorder Traversal

- Using the same `Graph` class, let's implement preorder traversal

[6]:
```python
def recursive_preorder_traversal(graph, start, visited=None):
  if visited is None:
```

```
    visited = set()

  # Process the current vertex
  print(start, end=' ')
  visited.add(start)

    # Recursive traversal of neighbors
  for neighbor in graph.graph.get(start, []):
    if neighbor not in visited:
      recursive_preorder_traversal(graph, neighbor, visited)
```

4.16 Recursive Graph Search: Preorder Traversal

[7]:
```
bfs(ex_graph, 1)
```

```
1 3 6 10 7
```

5 Nearest Neighbour Problem

5.1 Nearest Neighbour Problem

- As you may have encountered already, machine learning and statistical methods often depend on finding the nearest neighbor to a data point
  - K-nearest neighbors regression, propensity score matching
- In a k dimensional space, if we conduct a linear search for points, the running time will be $O(kn)$ for n data points.
- Can we do better?

5.2 k-d Trees

- k-d trees is short for k dimensional tree (notation is a bit unfortunate, different K than KNN)
  - It is useful for multidimensional searches
- Let's discuss the properties of k-d trees and why they work
- Binary tree where each node represents an axis-aligned hyperrectangle in the k-dimensional space
  - hyperrectangle: rectangle in higher dimensions
- Nodes in the left subtree have coordinates less than the splitting dimension of the current node, while nodes in the right subtree have coordinates greater than the splitting dimension.
- At each level of the tree, a specific dimension is chosen to split the data. The choice of dimension alternates as we traverse down the tree.
- Each leaf represents a single point in the k-dimensional space

5.3 k-d Trees Animation

https://commons.wikimedia.org/wiki/File:KDTree-animation.gif

5.4 Applications and Issues

- Notice k-d trees can also find values within a certain range very quickly, not just a specific point
- GIS (geographic information systems) queries
- KNN algorithm
- Computer graphics, such as ray tracing to facilitate efficient space partitioning
- Issues occur in high-dimensional spaces and trees can become imbalanced

6 Recommended Problems and References

6.1 Recommended Readings

- Bhargava: Chapter 6
- Bhargava: Chapter 11, pages 203 to 206 about Trees

6.2 Recommended Problems

- Cormen: Chapter 10 exercises
  - 10.3-1, 10.3-2, 10.3-3
- Bhargava: Chapter 6 exercises
  - 6.1 to 6.5

6.3 Recommended Problems

- Implement preorder, postorder, and level order traversal. Determine the time and space complexity in each case
- Implement a function that find an element in a BST and deletes it. The descendants of the deleted node are given to the deleted node's parent.
- Using the `graph` class from the slides, implement BST search such that it stops and tell you the distance the node is from the starting point.
  - For instance, if we searched for 7 in the graph given in the slides, it would return `"Found! Distance 2"`.
  - If we searched for 100 in the graph, it would return `"Not found!"`
- Implement postorder graph traversal using the `graph` class from the slides.
- Implement a function using recursion to find the sum heterogeneous nested lists such as [[1, [2]], [[[3]]], 4, [[5, 6], [[[7]]]]].

6.4 References

- Bhargava, A. Y. (2016). *Grokking algorithms: An illustrated guide for programmers and other curious people.* Manning. Chapter 6, 10,

  11.

- Cormen, T. H. (Ed.). (2009). *Introduction to algorithms* (3rd ed). MIT Press. Chapter 12 and 20.
- Horton, D., & Liu, D. (2023, November 19). *CSC148 Lecture Notes.* https://www.teach.cs.toronto.edu/~csc148h/winter/notes/