Computer Science Fundamentals:
From Logic to Algorithms & Data Structures

Christian J. Rudder

October 2024

Contents

Contents **1**

1 Graphs and Trees **5**
1.1 Paths and Connectivity 5
1.2 Breath-First and Depth-First Search 9
High-Level Overview 9
Edge Classifications – Directed Graphs 15
1.3 Tree Types & Traversals 20
Binary Trees: 2 Children 20
Binary Search Tree 23
2-3 Trees: 3 Children Search 28
1.4 Directed-Acyclic Graphs & Topological Ordering 31

Bibliography **36**

This page is left intentionally blank.

Preface

Big thanks to **Christine Papadakis-Kanaris**
for teaching Intro. to Computer Science II,
Dora Erdos and **Adam Smith**
for teaching BU CS330: Introduction to Analysis of Algorithms
With contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne,
at Boston University

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

— 1 —

Graphs and Trees

1.1 Paths and Connectivity

Graphs are similar to train networks or airline routes. They connect one location to another.

Definition 1.1: Graph

A **graph** is a collection of points, called **vertices** or **nodes**, connected by lines, called **edges**. Similarly to how a polygon has vertices connected by edges.

Definition 1.2: Undirected Graph

An **undirected graph** is a graph where the edges have no particular direction going both ways between nodes. A **degree** of a node is the number of edges connected to it.

Example: Figure (1.1) shows an undirected graph:

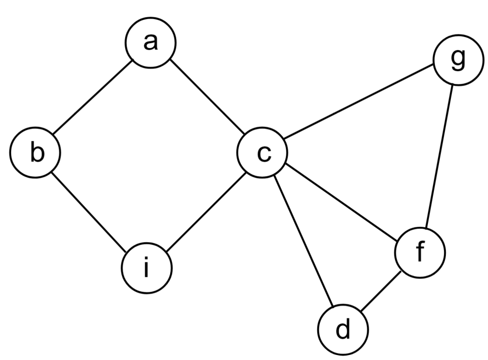


Figure 1.1: An undirected graph with 7 vertices and 9 edges.

Node a has a degree of 2, and node c has a degree of 5.

Definition 1.3: Directed Graph

A **directed graph** is where the edges have a specific direction from one node to another.

- The **indegree** of a node is the number of edges that point to it.
- The **outdegree** of a node is the number of edges that point from it.

Example: Figure (1.2) shows a directed graph:

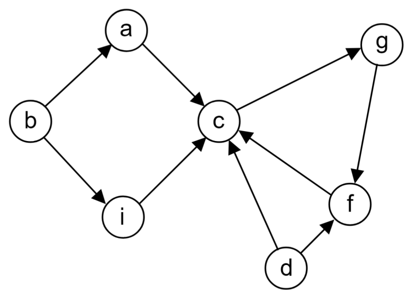


Figure 1.2: A directed graph with 7 vertices and 9 edges.

Node b has an outdegree of 2 and an indegree of 0. c has an indegree of 4 and an outdegree of 1.

Definition 1.4: Weighted Graph

A **weighted graph** is a graph where each edge has a numerical value assigned to it.

Example: Figure (1.3) shows a weighted graph:

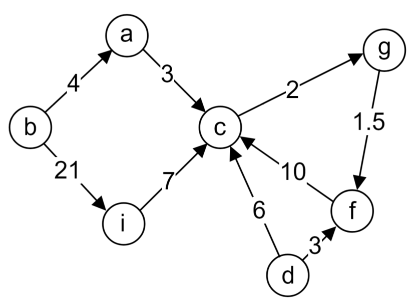


Figure 1.3: A weighted graph with 7 vertices and 9 edges.

Definition 1.5: Path

A **path** is a sequence of edges that connect a sequence of vertices. A path is **simple** if all nodes are distinct.

Example: In Figure (1.4), a simple path $h \leftrightarrow b \leftrightarrow i \leftrightarrow c \leftrightarrow d$ is shown:

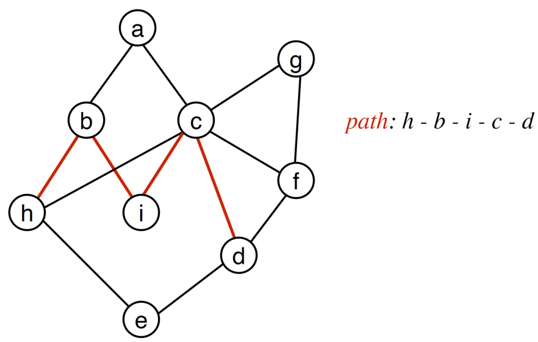


Figure 1.4: A graph with a simple path from h to d.

Definition 1.6: Connectivity

A graph is **connected** if there is a path between every pair of vertices.
A graph is **disconnected** if there are two vertices with no path between them.
Connected graphs of n nodes have at least $n-1$ edges.

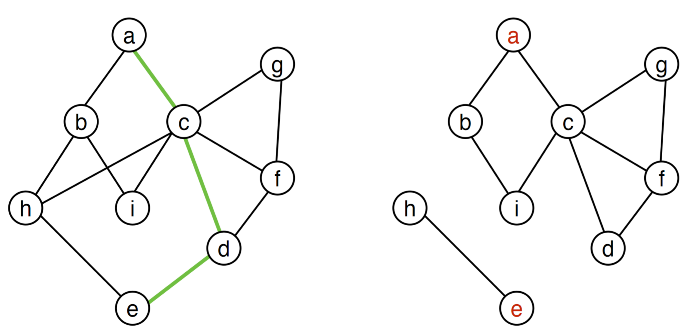
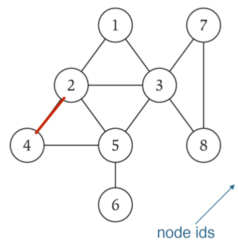


Figure 1.5: A connected graph $a \leftrightarrow c \leftrightarrow d \leftrightarrow e$ and disconnected graph.

Definition 1.7: Adjacency Matrix

An **adjacency matrix** is an $n \times n$ matrix where such that $A[i][j] = 1$ if there is an edge between nodes i and j.



$$\begin{array}{cc} & \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

undirected graph

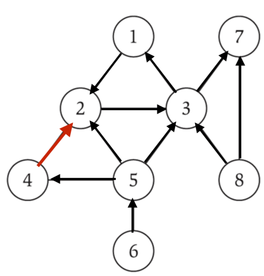
Figure 1.6: An adjacency matrix where the path $4 \leftrightarrow 2$ is highlighted ($A[2][4]$ or $A[4][2]$)

Theorem 1.1: Properties of Adjacency Matrix

The following properties hold for adjacency matrices:

- An undirected graph is symmetric about the diagonal.
- A directed graph is not symmetric about.
- A weighted graph has the weight of the edge instead of binary.

Space Complexity: $\Theta(n^2)$; **Time Complexities:**
Index edge $A[i][j]$**:** $\Theta(1)$; **List neighbors** $A[i]$: $\Theta(n)$; **List all edges:** $\Theta(n^2)$.



$$\begin{array}{cc} & \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Figure 1.7: An adjacency matrix of a directed graph with path $4 \to 2$ highlighted.

Definition 1.8: Adjacency List

An **adjacency list** is a list of keys where each key has a list of neighbors; This often takes the form of a hash-table: **Space Complexity:** $\Theta(n+m)$ for n nodes and m total edges; **Time Complexities: Index key:** $\Theta(1)$; **List key neighbors:** $\Theta(\#$ of outdegrees$)$; **List all edges:** $\Theta(n+m)$; **Insert edge:** $\Theta(1)$. **Note:** $m \leq n^2$ (all nodes connected to all nodes), though typically in practice, $m < n$.

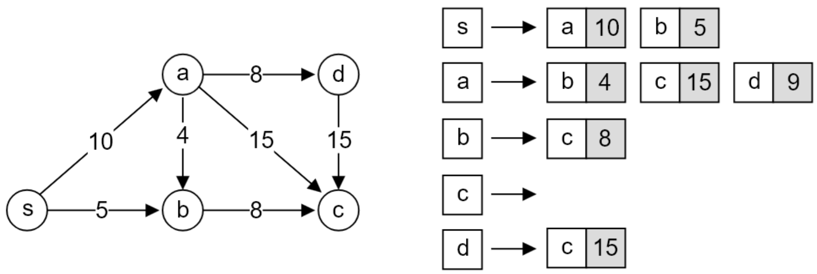


Figure 1.8: An adjacency list of a directed graph, notably c has no outdegrees.

1.2 Breath-First and Depth-First Search

High-Level Overview

Two general methods for traversing a graph are, **breadth-first search** and **depth-first search**.

Definition 2.1: Cycle

A **cycle** is a path that starts and ends at the same node.

Example: In the above Figure (1.7), $1 \to 2 \to 3 \to 1$ form a cycle.

Definition 2.2: Trees

A **tree** is a connected graph with no cycles. A **leaf-nodes** is the outer-most nodes of a tree. A **branch** is a path from the root to a leaf. **Interior nodes** have at least one child.

Tip: Watch this entire section: BFS & DFS (Edge Types, Traversal Orders, & Code).

Theorem 2.1: Tree Identity

Let G be an undirected graph of n nodes. Then any two statements imply the third:

(i.) G is connected. (ii.) G has $n-1$ edges. (iii.) G has no cycles.

Definition 2.3: Rooted Trees

A tree **root** is a designated starting node, which all other nodes follow. The root has no parent/ancestors (proceeding node), and all other nodes are descendants (children).

A **subtree** is a tree formed by picking another node as the root, representing a subset of the original tree.

Definition 2.4: Levels and Heights

The **level** (or **depth**) of a node v in a tree is the number of edges on the path from the root to v, so level(root) $= 0$. The **height** of a node v is the number of edges on the longest downward path from v to any leaf. The **height** of a tree is defined as the height of its root node.

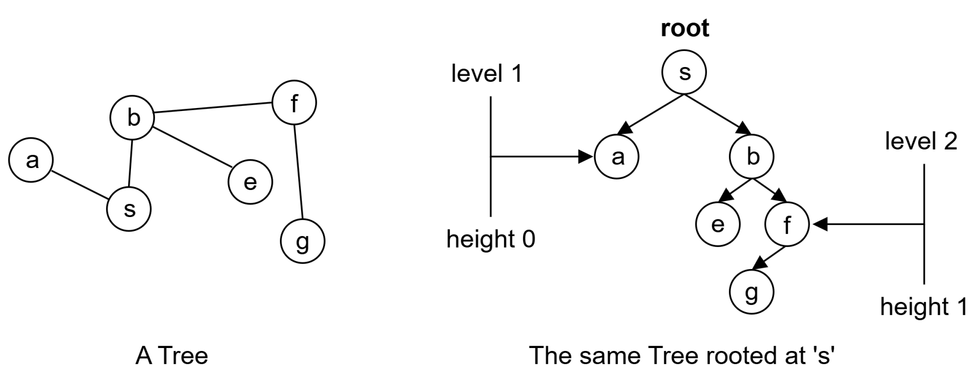


Figure 1.9: The right shows a rooted tree with root s, with a and b as direct children. A possible subtree starts with node b, only consisting of $\{b, e, f, g\}$. **Note:** A tree is not necessarily directed, while a rooted tree is (floods outwards from the root)

Definition 2.5: Breadth-First Search (BFS)

In a **breadth-first search**, we start at a node's children first before moving onto their children's children in level order.

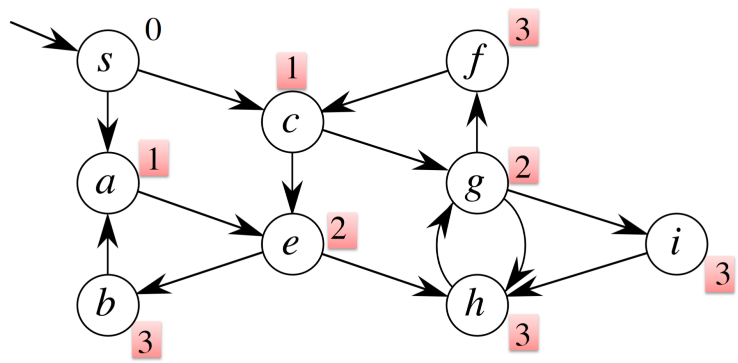


Figure 1.10: A BFS tree traversal preformed on a graph with each level enumerated

Theorem 2.2: Properties of BFS

BFS run on any graph T produces a tree T' with the following properties:

(i.) T' is a tree.

(ii.) T' is a rooted tree with the starting node as the root.

(iii.) The height of T' is the shortest path from the root to any node.

(iv.) Any sub-paths of T' are also shortest paths.

Proof 2.1: Proof of BFS

(i.) and (ii.) follow from the definition of a tree. (iii.) and (iv.) follow that since a tree contains a direct path to any given node in our parent child relationship, that path must be the shortest. ■

Tip: In a family tree, there is only one path from each ancestor to each descendant.

We create a BFS algorithm from what we know, though not the best implementation:

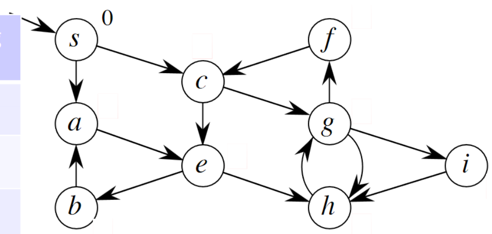
Function 2.1: BFS Algorithm - BFS(s)

Breadth-First Search starting from node s.

Input: Graph $G = (V, E)$ and starting node s.
Output: Levels of each vertex from s.

Function *BFS(s)*:
  for *each* $v \in V$ **do**
    Level[v] $\leftarrow \infty$;
  Level[s] $\leftarrow 0$;
  Add s to Q;
  while Q *not empty* **do**
    $u \leftarrow Q$.Dequeue();
    for *each* $v \in G[u]$ **do**
      if *Level[v]* $= \infty$ **then**
        Add edge (u, v) to tree T (parent[v] $= u$);
        Add v to Q;
        Level[v] $\leftarrow$ Level[u] + 1;

u	Queue contents before exploring u	… after exploring u
s	(empty)	[a,c]
a	[c]	[c,e]
c	[e]	[e,g]
e	[g]	[g,b,h]
g	[b,h]	[b,h,f,i]
b	[h,f,i]	[h,f,i]
h	[f,i]	[f,i]
f	[i]	[i]
i	(empty)	(empty)



Loop invariant: If the first node in the queue has level i, then the queue consists of nodes of level i possibly followed by nodes of level $i + 1$.

Consequence: Nodes are explored in increasing order of level.

Figure 1.11: A table showing the queue at each level of iteration

We analyze the time and space complexity in the below Figure (1.12):

```
BFS(s):
1. Initialize Level[v] to infinity for all vertices
2. Set Level[s]=0
3. Add s to Q                              } O(1) time, run once overall.
4. While (Q not empty)
   a) u ← Q.Dequeue()                      } O(1) time, run once per vertex
   b) For each node v in G[u]
      a) If Level[v]= ∞ then               O(1) time per execution,
         a) Add edge (u, v) to tree T      run at most twice per edge
         b) Add v to Q                     } Total: O(m+n) time
         c) Level[v] := Level[u]+1           (linear in input size)
```

Figure 1.12: An analysis showing $O(m+n)$ for both time and space complexity

Proof 2.2: Claim 1 for BFS

Let s be the root of the BFS tree, then: **Proof:** Induction on the distance from s to u.
Base case $(u = s)$: The code sets Level$[s] = 0$, and there is no path to find since the path has length 0.
Induction hypothesis: For every node u at distance $\leq i$, Claim 1 holds.
Induction step:

- Let v be a node at distance exactly $i+1$ from s. Let u be its parent in the BFS tree.
- The code sets Level$[v]$ = Level$[u] + 1$.
- Let x be the last node before v on a shortest path from s to v. Since v is at distance $i+1$, then x must be at distance i, and so Level$[x] = i$ (by induction hypothesis).
- If $u = x$, we are done!
- If $u \neq x$, then it must be that u was explored before x, since otherwise x would be the parent of u.
- Since we explore nodes in order of level, Level$[u] \leq$ Level$[x] = i$.
- If Level$[u] = i$, then we are done.
- If Level$[u] < i$, then the path $s \sim u \to v$ has length at most i, which contradicts the assumption that the distance of v is $i+1$.

We conclude that Level$[u] = i$, Level$[v] = i+1$, and the path in the BFS tree that goes from s to u to v has length $i+1$. ∎

Definition 2.6: Depth-First Search (DFS)

In a **depth-first search**, we recursively explore each an entire branch before moving onto the next.

Function 2.2: DFS Algorithm - DFS(G)

Depth-First Search on graph G (recursive).

Input: Graph $G = (V, E)$.
Output: Discovery and finishing times for each vertex.

```
Function DFS(G):
    for each u ∈ G do
        u.state ← unvisited;
    time ← 0;
    for each u ∈ G do
        if u.state == unvisited then
            DFS-Visit(u);

Function DFS-Visit(u):
    time ← time + 1;
    u.d ← time ;
    // record discovery time;
    u.state ← discovered;
    for each v ∈ G[u] do
        if v.state == unvisited then
            DFS-Visit(v);
    u.state ← finished;
    time ← time + 1;
    u.f ← time;
    // record finishing time;
```

Time and Space Complexity: $O(n + m)$ where n is the number of vertices and m the number of edges.

Proof 2.3: DFS Correctness

Case 1: When s is discovered, there is a path of unvisited vertices from s to y. We use induction on the length L of the unvisited path from x to y.

- **Base case:** There is an edge (x, y), and y is unvisited.
- **Induction hypothesis:** Assume that the claim is true for all nodes reachable via k unvisited nodes.
- **Induction step:**
  - Consider u reachable via $k+1$ unvisited nodes.
  - Let z be the last node on the path before y, and z is discovered from x (by I.H.).
  - The edge (z, y) will be explored from x, ensuring that y is eventually visited.

Thus, DFS-Visit(x) explores all nodes reachable from x through a path of unvisited nodes, as required. ■

Edge Classifications – Directed Graphs

To build our intuition around edge classifications, we'll follow a walkthrough of DFS & BFS on directed graphs to build our intuition. We'll follow the below format:

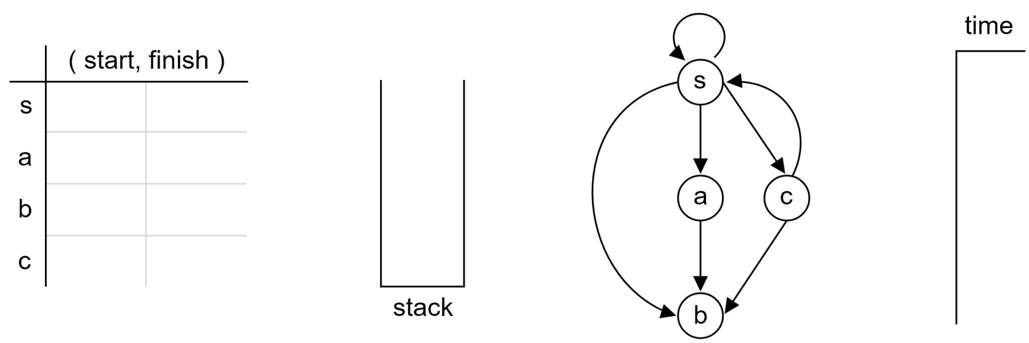


Figure 1.13: A table for start and finish times of each visited nodes, our stack to keep track of traversals, the graph that shall be traversed, and a timer for DFS (BFS will use a queue).

Let's start by filling out a partial DFS traversal:

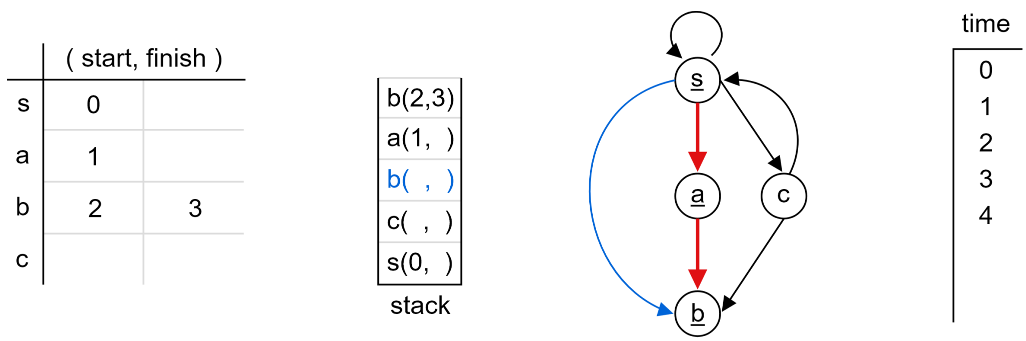


Figure 1.14: So far we have asked s to put all their children on the stack at timestamp 0. We first investigate the first child a's branch at time 1. It gives us b, for which we time stamp (2,3) for start and finish. We highlight the first the alternative b branch in blue for distinction. The edges that are in our (start, finish) table are **Tree Edges**.

Definition 2.7: Forward Edge

A **non-tree edge** that connects an already discovered node to a descendant in the DFS tree is called a **forward edge**. Concretely, (u, v) is a forward edge if given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_1 < s_2 < f_2 < f_1$.
Where the **start time** of u is less than v, and the **finish time** of u is greater than v.

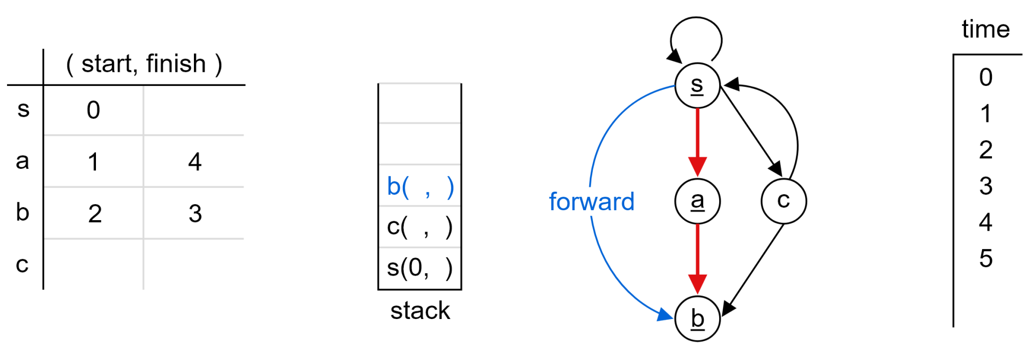


Figure 1.15: Continuing from Figure (1.14), b and a are taken off the stack. The next b on the stack is a **forward edge**, as s has a smaller start time, and a foreseeable finish time than the table's b.

The next two are **cross and back edges**:

Definition 2.8: Cross Edge

A **non-tree edge** that connects branches of a tree is called a **cross edge**. In particular, (u, v) is a cross edge if v is not a descendant or ancestor of u. Concretely, given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_2 < s_1 < f_2 < f_1$.
Where the **start and finish time** of u is greater than v.

Definition 2.9: Back Edge

A **non-tree edge** that connects a node to an ancestor in the tree is called a **back edge**. Concretely, given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_2 < s_1 < f_1 < f_2$.
Where the **start time** of u is greater than v, but the **finish time** of is less than v.

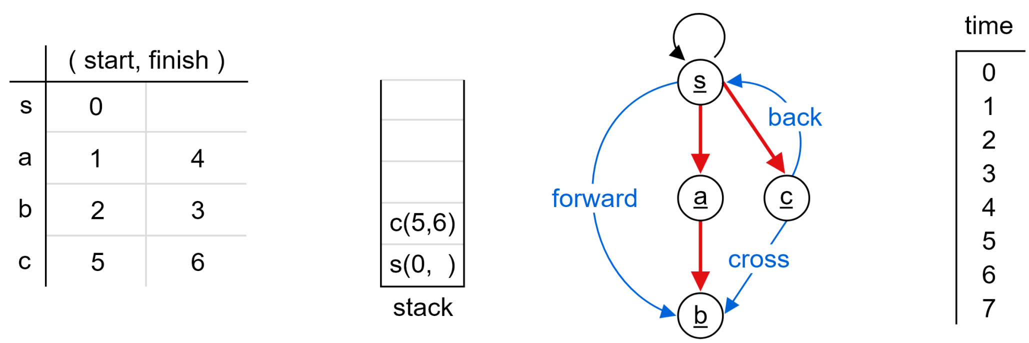


Figure 1.16: Continuing from Figure (1.15), b and a are taken off the stack. Next on the stack is c, which reveals children b and s; Here, b has already been processed (cross edge), and s discovered, but yet to be finished (back edge). There is nothing else to explore, so we finish c. **Note:** There is no classifications for self referential edges.

Next we'll see how our classifications work on BFS.

Starting with the same graph as before we instead run BFS:

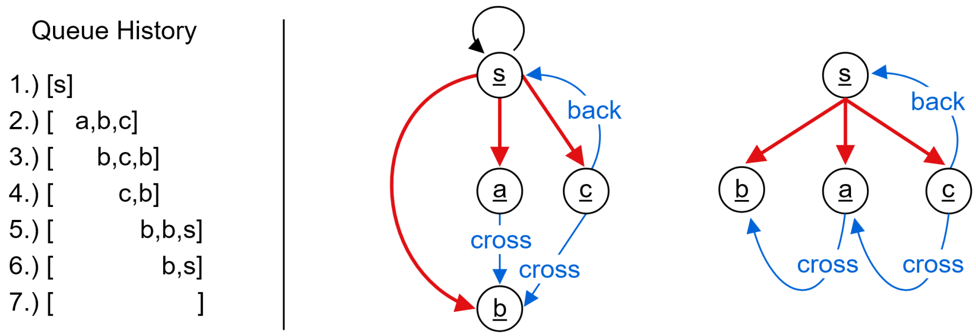


Figure 1.17: We start with s modeling the queue history on the left. The first graph is our original graph finished, and on the right it is redrawn for clarity.

Theorem 2.3: DFS & BFS Directed Graphs

Between DFS and BFS, the following holds:

- **DFS**: Tree, back, forward, and cross edges (all edges).
- **BFS**: Tree, cross, and back edges (no forward edges).

Additionally, one may find the following theorem helpful:

Theorem 2.4: DFS and Cycles

Let DFS run on graph G, then:

$$(G \text{ has a cycle}) \iff (\text{DFS run reveals back edges})$$

Proof 2.4: Proof of Cycles and Back Edges

Proving (G has a cycle) $\Longleftarrow$ (DFS run reveals back edges): Every back edge creates a cycle.
Proving (G has a cycle) $\Longrightarrow$ (DFS run reveals back edges) Suppose G has a cycle: Let u_1 be the first discovered vertex in the cycle, and let u_k be its predecessor in the cycle. u_k will be discovered while exploring u_1. The edge (u_k, u_1) will be a back edge. ■

Next we deal with undirected graphs briefly and give a summary:

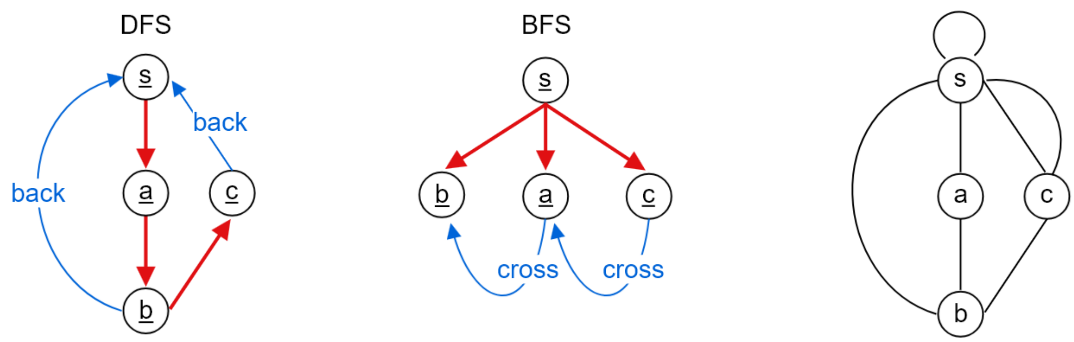


Figure 1.18: Here DFS and BFS are shown, with the original graph on the far right. Here it's quite peculiar to have a double-edge in an undirected graph. In our case it's redundant and may be counted as a single edge.

Summary:
This allows us to summarize the edge classifications for both directed and undirected graphs in the below table:

Graph Type	**DFS Edge Types**	**BFS Edge Types**
Directed Graphs	Tree, Back, Forward, Cross	Tree, Back, Cross
Undirected Graphs	Tree, Back	Tree, Cross

Table 1.1: Edge classifications for DFS and BFS in directed and undirected graphs.

Edge Type	**Start Condition**	**Finish Condition**
Forward Edge	less	greater
Cross Edge	greater	greater
Back Edge	greater	less

Table 1.2: Summary of start and finish conditions for edge types by comparing the starting node with its endpoint. E.g., in a forward edge (u, v), the start time of u is less than v, but the finish time is greater.

1.3 Tree Types & Traversals

Binary Trees: 2 Children

First we'll define a basic binary tree and then all its extensions.

Definition 3.1: Binary Tree

A **binary tree** is a tree where each parent node has at most two children.

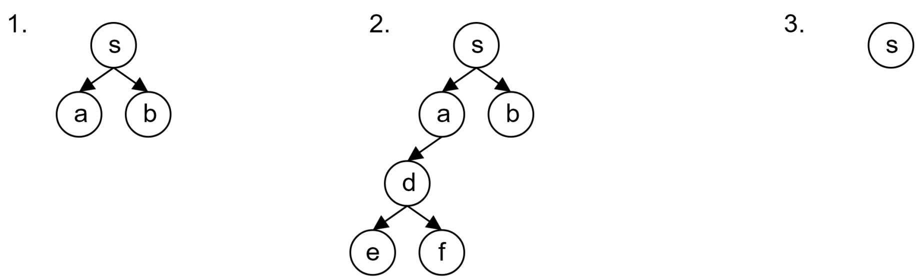


Figure 1.19: These are three examples of binary trees, including a single node (3).

To simply do a binary tree traversal, we may employ the following methods:

Function 3.1: DFS Binary Tree Traversal - DFS(T)

Depth-First Search on binary tree T (recursive).

Input: Binary tree T.
Output: Nodes in pre-order, in-order, and post-order.

```
Function DFS(T):
    if T is not empty then
        Visit node T;
        DFS(T.left);
        DFS(T.right);
```

Time Complexity: Given n nodes, and a force of a tree format, we have $n - 1$ edges. Therefore with m edges, $n + m = n + (n - 1)$, hence $O(n)$ time.
Space Complexity: again, we have $O(n)$ space for the stack.

→ **BFS has the same complexities as per the same reasoning as the above function.**

The order in the above function is called **pre-order traversal**;

Definition 3.2: Order Traversals

The order of traversal refers to the order in which nodes are visited/processed:

- **Pre-order:** Visit the node, then left, then right.
- **In-order:** Visit left, then the node, then right.
- **Post-order:** Visit left, then right, then the node.

In particular, **in-order traversal** is simply a run of **BFS** as it processes the tree level by level starting from the root.

Below is an example of all three:

Example 3.1: Binary Tree Traversals (Part 1)

```
# Pre-order Traversal
def PreOrder(T):
    if T is not None:
        visit(T)
        PreOrder(T.left)
        PreOrder(T.right)

# In-order Traversal
def InOrder(T):
    if T is not None:
        InOrder(T.left)
        visit(T)
        InOrder(T.right)

# Post-order Traversal
def PostOrder(T):
    if T is not None:
        PostOrder(T.left)
        PostOrder(T.right)
        visit(T)

# Level-order Traversal
def LevelOrder(T):
    BFS(T)
```

■

Now to show how this actually looks:

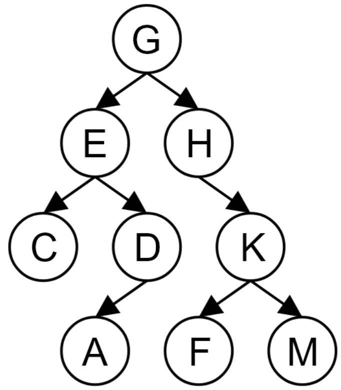


Figure 1.20: The above binary tree has the following traversals:

- **Pre-order:** G, E, C, D, A, H, K, F, M
- **In-order:** C, E, A, D, G, H, F, K, M
- **Post-order:** C, A, D, E, F, M, K, H, G
- **Level-order:** G, E, H, C, D, K, A, F, M

Notably, if there is no left or right child to explore for that particular traversal order, that node then and there is processed. For example, in in-order traversal, since C has no left child, it is processed first. The same goes for Post-order, where D has no right child, so it is processed.

Definition 3.3: Types of Binary Trees

There are several types of binary trees, each with its own properties:

- **Full Binary Tree:** Every node has either 0 or 2 children.
- **Complete Binary Tree:** All levels are fully filled except possibly the last level, which is filled from left to right.
- **Perfect Binary Tree:** All interior nodes have two children and all leaves are at the same level.
- **Balanced Binary Tree:** The height of the left and right subtrees of every node differ by at most one.

Note: A full binary tree is not necessarily complete, and a complete binary tree is not necessarily full.

Observe the following examples of binary trees:

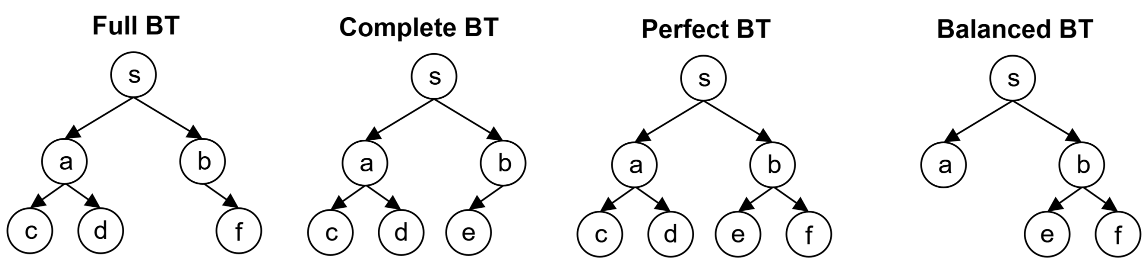


Figure 1.21: These are examples of different types of binary trees (BT).

Binary Search Tree

A binary search tree is a stricter form of a binary tree, which gives us an efficient search algorithm.

Definition 3.4: Binary Search Tree (BST) – Overview

A **binary search tree** is a binary tree where:

$$\text{left child} < \text{parent node} \leq \text{right child}$$

Note: A Binary Tree <u>**is not**</u> a Binary Search Tree, unless it satisfies the above conditions.

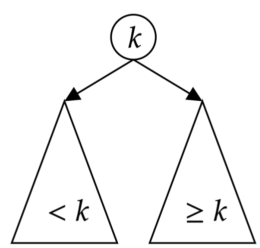


Figure 1.22: Visual Representation of Definition (3.4), where k is the parent node with its respective left (less than) and right (greater than or equal to) nodes.

The next few pages we discuss **Searching, Insertion, and Deletion** in a binary search tree.

Function 3.2: Binary Search Tree (BST) – Searching

Given a BST root node T and a desired value v, we can search for v in T as follows:

Input: BST root node T, value v.
Output: Node containing v or null if not found.

```
Function Search(T, v):
    while T is not null do
        if T.value = v then
            return T;
        if v < T.value then
            T ← T.left;
        else
            T ← T.right;
    return null;
```

Time Complexity: $O(h)$ where h is the height of the tree. In a balanced BST, this is $O(\log n)$, but in the worst case (unbalanced), it can be $O(n)$.

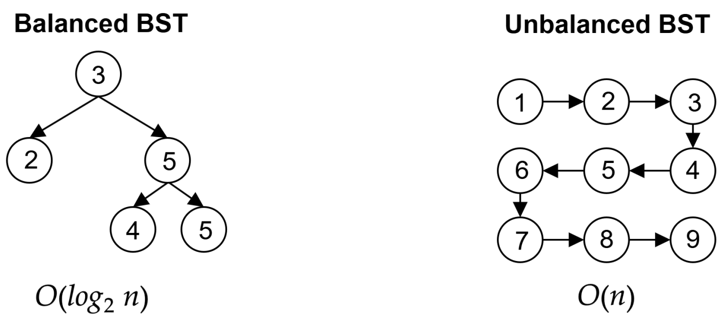


Figure 1.23: Searching for a value in a binary search tree. The worst case is a noodle (all right nodes), forcing n traversals, i.e., the height (within a constant). One could imagine these nodes as complex objects, where the value v is a key, and the node itself contains more data.

Theorem 3.1: Property of Binary Search Trees

In a binary search tree, for any node N:

- All values in the left subtree of N are less than the value of N.
- All values in the right subtree of N are greater than or equal to the value of N.

Insertion is the same an searching, except we insert upon reaching a null node.

Function 3.3: Binary Search Tree (BST) – Insertion

Given a BST root node T and a value v, we can insert v into T as follows:

Input: BST root node T, value v.
Output: Updated binary search tree with v inserted.

```
Function Insert(T, v):
    parent ← null; current ← T;
    while current is not null do
        parent ← current;
        if v < current.value then
            current ← current.left;
        else
            current ← current.right;

    // After the while loop terminates
    if parent is null then
        return v ; // Tree was empty, v becomes root
    if v < parent.value then
        parent.left ← v;
    else
        parent.right ← v;
    return T;
```

Time Complexity: $O(h)$ where h is the height of the tree. In a balanced BST, this is $O(\log n)$, but in the worst case (unbalanced), it can be $O(n)$.

Deletion is a bit more complex, as we have to consider removing intermediate nodes:

Theorem 3.2: Binary Search Tree (BST) – Deletion

When deleting a node from a binary search tree, we must consider three cases:

- v **has no children:** Remove the parent's reference to v.
- v **has one child:** Replace the parent's reference to v's child.
- v **has two children:** Replace the parent's reference to v with either:
    - The **L**argest value in v's **L**eft subtree (in-order predecessor).
    - The **R**unt (smallest) value in v's **R**ight subtree (in-order successor).

This ensures that the binary search tree properties are maintained after deletion.

Function 3.4: Binary Search Tree (BST) – Deletion

Given a BST root node T and a value v, we can delete v from T as follows:

Input: BST root node T, value v.
Output: Updated binary search tree with v deleted.

```
Function Delete(T, v):
    parent ← null; current ← T;
    while current is not null and current.value ≠ v do
        parent ← current;
        if v < current.value then
            current ← current.left;
        else
            current ← current.right;
    if current is null then
        return T ; // Value not found
    // Case 1: Node has at most one child
    if current.left is null or current.right is null then
        if current.left is not null then
            child ← current.left;
        else
            child ← current.right;
        if parent is null then
            return child ; // Deleted root
        if parent.left = current then
            parent.left ← child;
        else
            parent.right ← child;
        return T;
    // Case 2: Node has two children
    succParent ← current;
    succ ← current.right ; // Runt of Right subtree
    while succ.left is not null do
        succParent ← succ;
        succ ← succ.left;
    current.value ← succ.value;
    if succParent.left = succ then
        succParent.left ← succ.right;
    else
        succParent.right ← succ.right;
    return T;
```

Time Complexity: $O(h)$ where h is the height of the tree. In a balanced BST, this is $O(\log n)$, but in the worst case (unbalanced), it can be $O(n)$.

Consider an example as we build a binary search tree and performing the above operations:

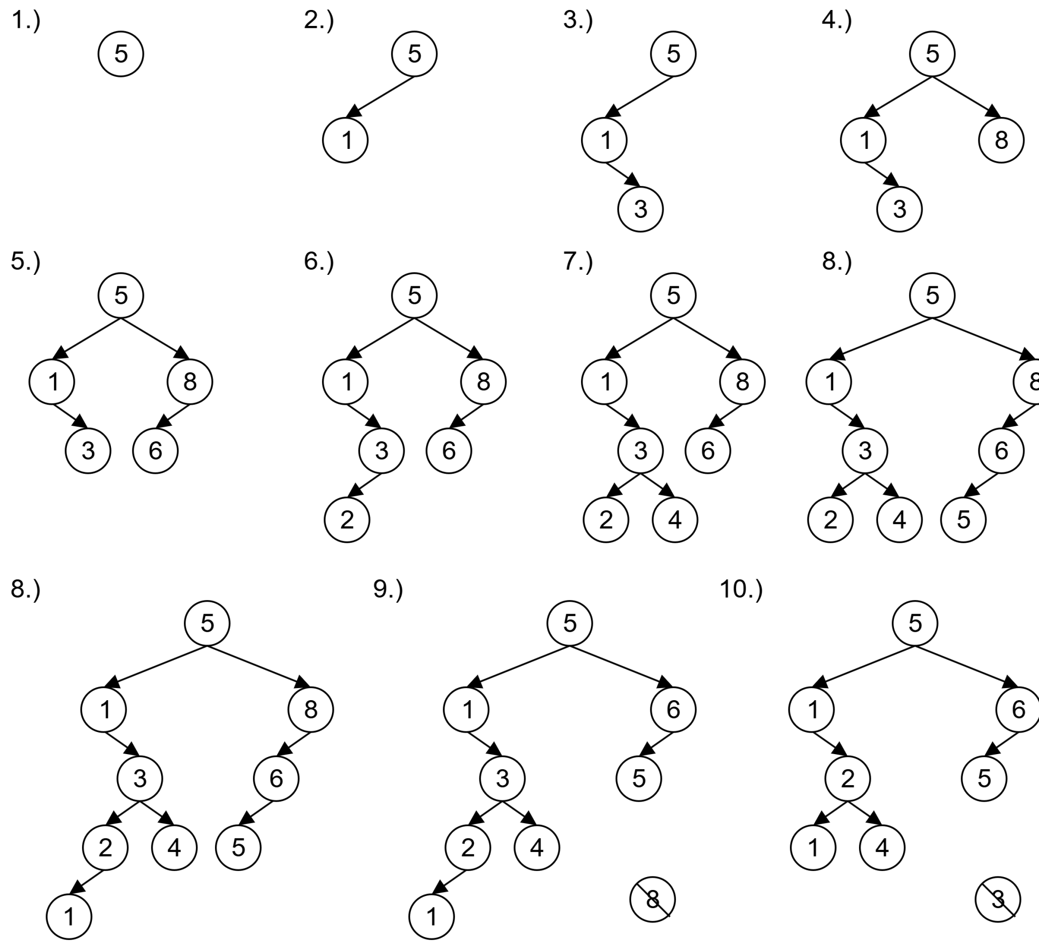


Figure 1.24: **Note:** We use the Largest Left child (in-order predecessor) to replace the 2-children case. From 1–9 we insert values. From 10–11 we delete. In (10) we delete 8 (1-child case), we replace its parent's reference to 8 with its only child 6. In (11) we delete 3 (2-children case), we replace its parent's reference to 3 with the largest left child, which is 2. One could imagine the in-order successor (smallest right child) would have 4 instead of 2, with 2 and 1 on its left subtree.

2-3 Trees: 3 Children Search

We can keep extending the binary tree to allow for more children, which allows us to search faster.

Definition 3.5: Ternary & N-ary Trees

A **ternary tree** is a tree where each parent node has at most three children. An **n-ary tree** is a tree where each parent node can have up to n children.

Definition 3.6: 2-3 Tree – Overview

A **2-3 tree** is a balanced search tree where a parent may contain 1 or 2 keys:

- **1 Key (1K):** This parent has 0 or 2 children.
- **2 Keys (2K):** This parent has 0 or 3 children.

The 1K node abides by the same rules as a BST:

$$\text{left child} < \text{parent node} \leq \text{right child}$$

For 2K let us define the left key as k_1 and the right key as k_2, then the rules are :

$$\text{left child} < k_1 \leq \text{middle child} < k_2 \leq \text{right child}$$

Time Complexity: Searching, inserting, and deleting in a 2-3 tree is $O(\log n)$, where n is the number of keys in the tree. The performance gain comes from 3 children possibility: if the tree has only 3 children, it would be $O(\log_3 n)$, which is faster than $O(\log_2 n)$; However, the tree has have a mix of 1K and 2K nodes. So that's why in the worst case, we have $O(\log_2 n)$. I.e., $\lfloor \log_3 n \rfloor \leq h \leq \lfloor \log_2 n \rfloor$, where h is the height of the tree.

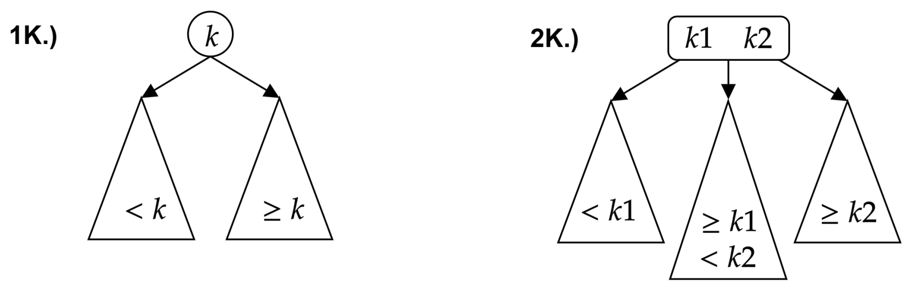


Figure 1.25: Visual Representation of Definition (3.4), where k_1 and k_2 are the keys of the parent node with its respective left, middle, and right nodes.

Definition 3.7: 2-3 Tree – Search & Insert

Searching is similar to a BST (3.2), we compare the desired value v to the keys (If $v < k_1$, we search in the left child. If $k_1 \leq v < k_2$, we search in the middle child. If $v \geq k_2$, we search in the right child. If the child is null, it does not exist.)

Inserting a value v into a 2-3 tree involves the following steps after a search:

- **Null:** If no nodes exists, create a new $1k$ node of v.
- **1K Node:** Insert v in sorted order, becoming a 2K node.
- **2K Node:** Insert v in sorted order (3 keys), then split:
  - **No Parent:** Push the middle key up taking the remaining left (LK) and right (RK) keys and create two new children satisfying 1K node rules (3.6).
  - **Parent is 1K:** The parent becomes a 2K node. The remaining LK and RK keys are again placed respectively satisfying the 2K node rules (3.6).
  - **Parent is 2K:** The parent momentarily becomes a 3K node with 4 children. These children are split are split between k_1 and k_2 respectively after the middle key is pushed up.

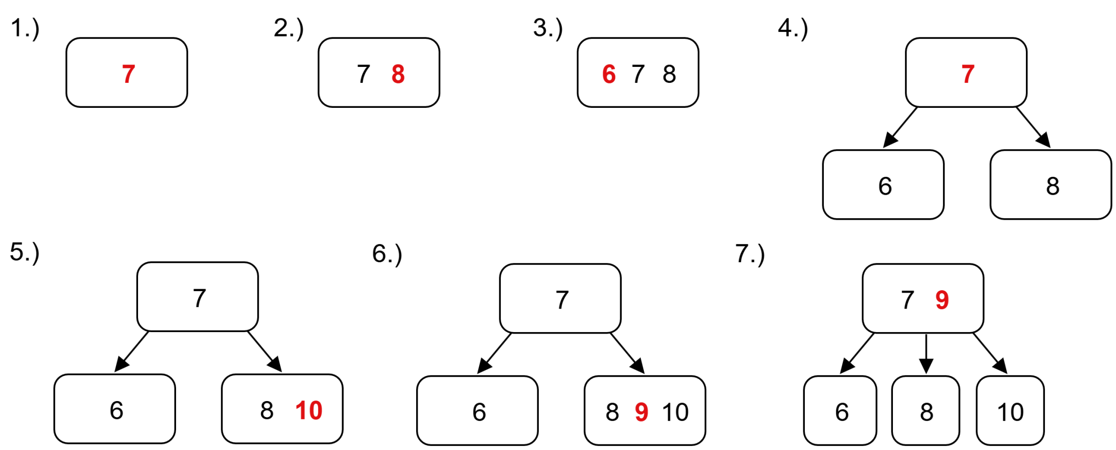


Figure 1.26: Visual Representation of Definition (3.7). In (3), there are three keys, initiating a split, pushing up 7. In particular, 1 & 4 are 1K nodes, and 2 & 7 are 2k nodes.

The next page provides a more complex example demonstrating the "Parent is 2K" case.

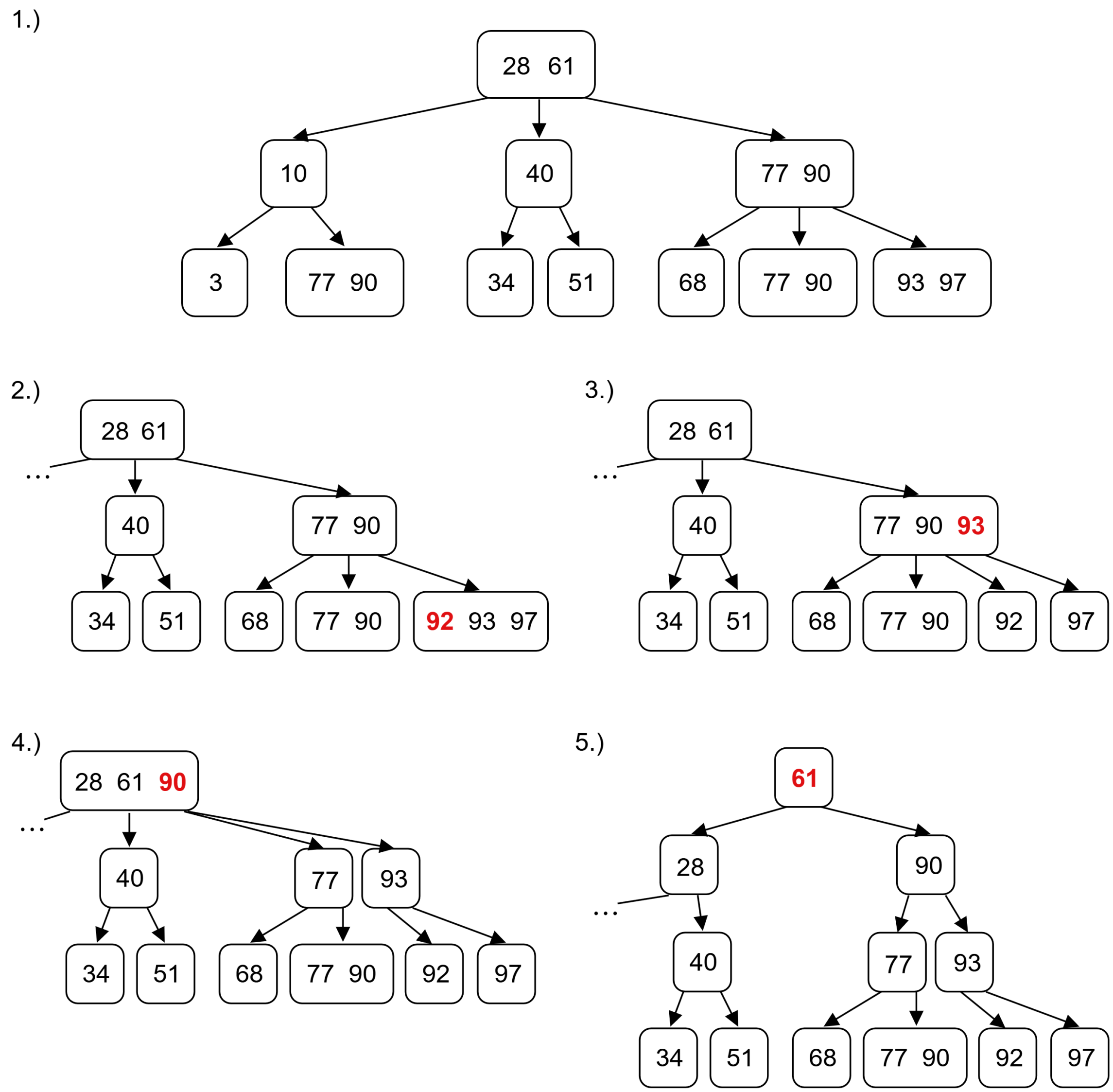


Figure 1.27: Visual Representation of Definition (3.7) of the "Parent is 2K" case: In (2) we start out inserting 92, overflowing the current node, (3) 93 is pushed up, splitting the previous node $\{92, 97\}$ into total 4 children, (4) 90 is pushed up, splitting node $\{77, 93\}$, each taking their respective two children from the previous children-overflow, (5) and finally, 61 is pushed up in the same manner.

1.4 Directed-Acyclic Graphs & Topological Ordering

Graphs may represent a variety of relationships, such as dependencies between tasks or the flow of information.

Definition 4.1: Directed-Acyclic Graph (DAG)

A **directed-acyclic graph** is a directed graph with no cycles.

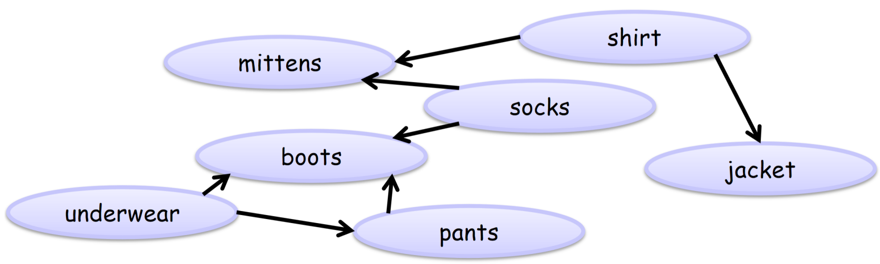


Figure 1.28: A DAG depicted by getting dressed for winter. For each node to be processed its **dependencies** or parents must be processed first.

Definition 4.2: Topological Ordering

Given a graph, a **topological ordering** is a linear ordering of nodes such that for every edge (u, v), u comes before v.

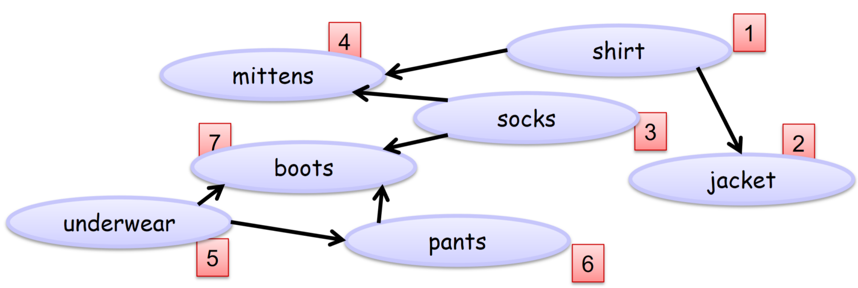


Figure 1.29: A topological ordering of the DAG in Figure (1.28) enumerated in red. Another possible ordering of [1, 2, 3, 4, 5, 6, 7] is [5, 6, 1, 2, 3, 4, 7], as $5 \rightarrow 6$ is independent.

Theorem 4.1: Topological Sort

Given a DAG, a topological ordering can be found.

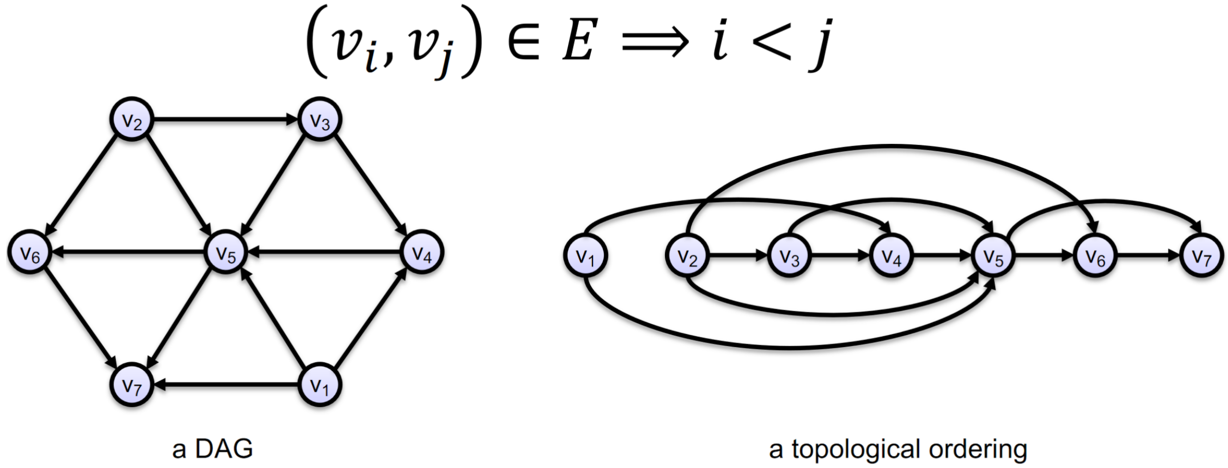


Figure 1.30: A topological sorting of a DAG E and v nodes

Proof 4.1: Topological Sort via DFS

Lemma: In a directed graph G, if (note necessarily acyclic):

- (u, v) is an edge, and
- v is not reachable from u,

Then in every run of DFS, $u.f > v.f$.
Proof:

- If v is started before u, then the DFS-Visit(v) will terminate without reaching u (because there is no path to u).
- If u is started before v, then the edge (u, v) will be explored before u is finished.

Therefore, in all cases, $u.f > v.f$. ■

Definition 4.3: Strongly Connected Components

A **strongly connected component** is a subgraph where every node is reachable from every other node. Then we say $u \rightsquigarrow v$ and $v \rightsquigarrow u$ are **mutually reachable**.

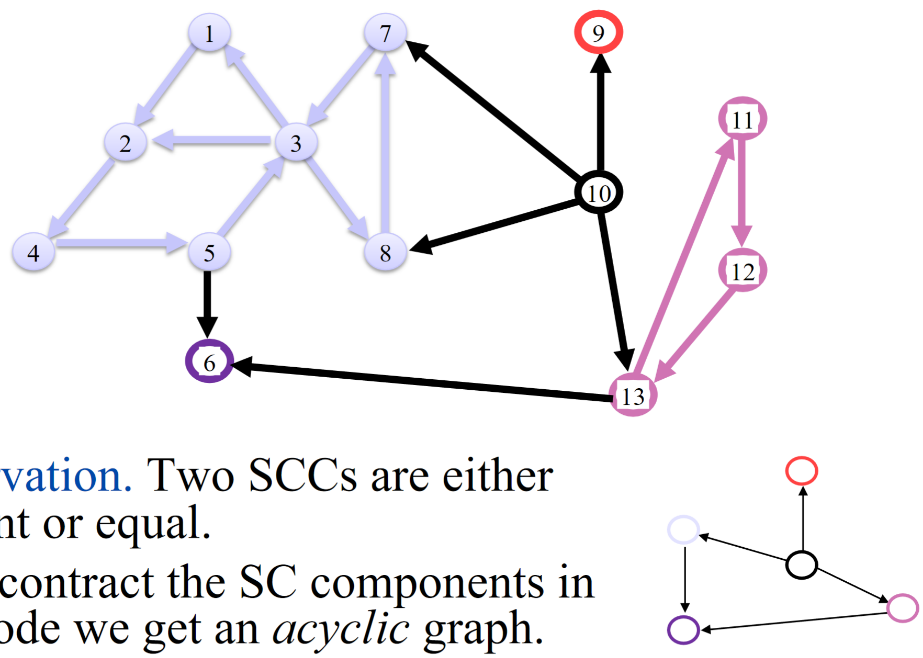
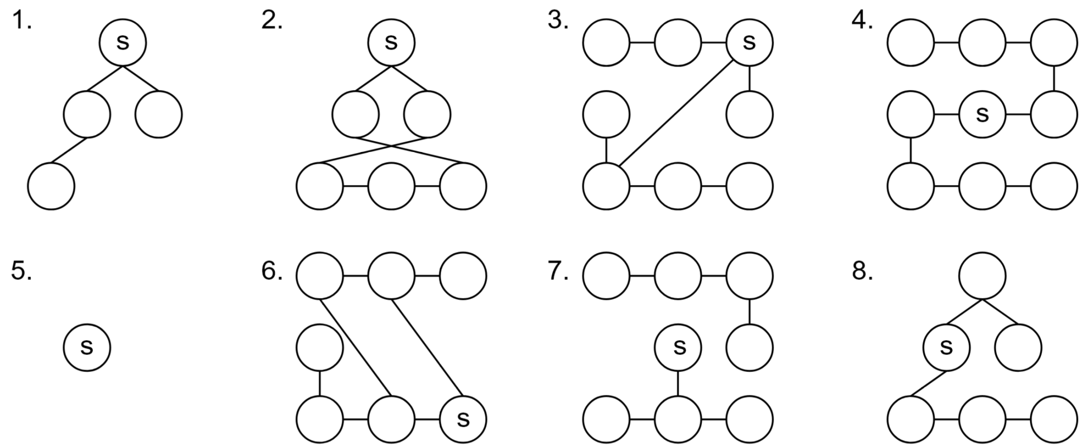


Figure 1.31: A graph with 5 strongly connected components.

Exercise 4.1: Based on the images which are trees, and if not, why?



Exercise 4.2: Based on Exercise (4.1) above, starting with the s nodes, what are possible results of BFS and DFS?

Exercise 4.3: What type of edges do BFS and DFS traversals produce on directed and undirected graphs?

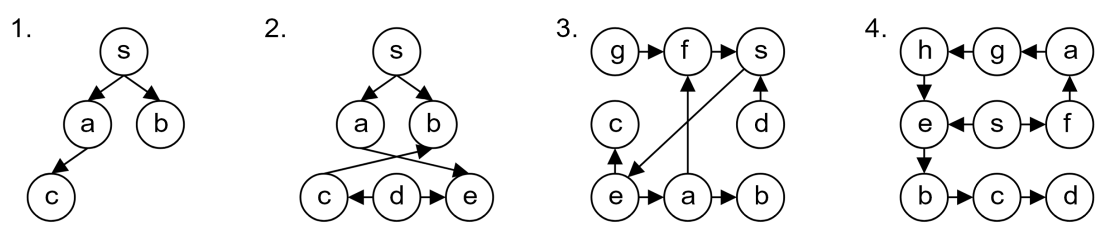
Exercise 4.4: BFS can find shortest paths, does DFS find longest paths (True/False)?

Exercise 4.5: To find a path and return such path from node s to node t in a graph, should we use BFS or DFS?

Exercise 4.6: To detect cycles in a graph, should we use BFS or DFS, or both?

Exercise 4.7: Say we wanted to find the shortest round-trip path, that is, the minimal path starting from origin node s visiting a set of nodes exactly once returning to s. How should we use BFS or DFS?

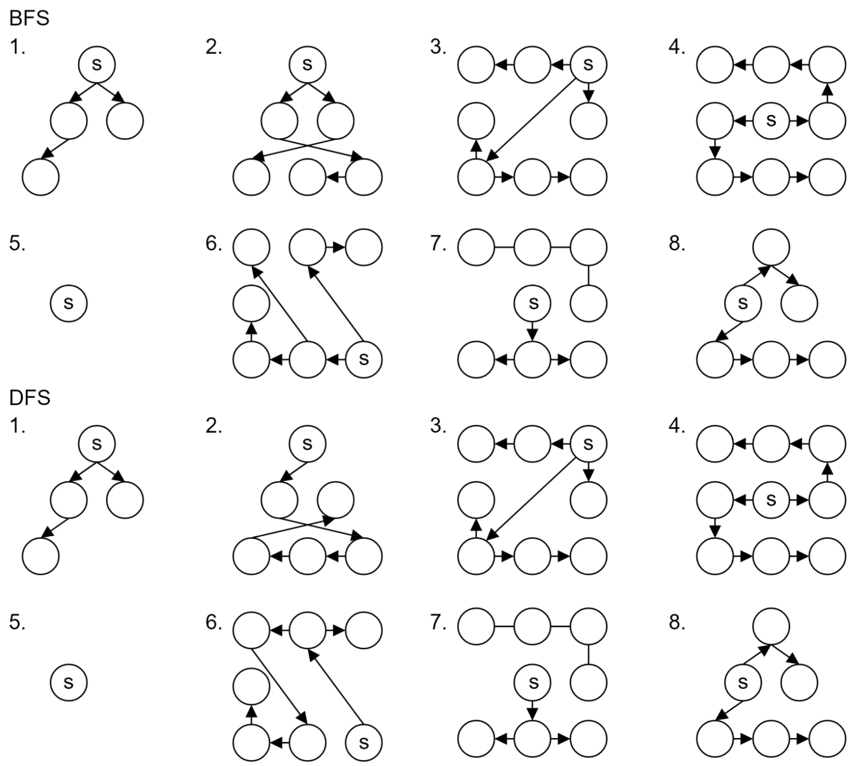
Exercise 4.8: Which of the below images are DAGs.



Exercise 4.9: Based on Exercise (4.8) above, what are possible topological orderings?

Answer 4.1: 1, 3, 4, 5, 8 are trees. 2 has a cycle, 6 has a cycle, 7 is disconnected.

Answer 4.2: There are other possible solutions depending on order of traversal. For 7, BFS and DFS don't alone handle disconnected graphs. Though modifications could be made to handle such.



Answer 4.3: Undirected - BFS: Tree edges, DFS: Tree edges, Back edges. Directed - BFS: Tree edges, Back edges, Cross edges, DFS: Tree edges, Back edges, Forward edges, Cross edges [1, 3, 2]

Answer 4.4: False, as of today, there is no known algorithm to find the longest path in a graph.

Answer 4.5: BFS, it finds shortest child-parent connections, giving us a direct path from s to t.

Answer 4.6: Both, as DFS when it encounters a node that has already been visited, it has found a cycle. BFS when two branches meet, a cycle is found.

Answer 4.7: BFS, one implementation is to run BFS from s to all nodes, the first cycle found is the shortest round-trip path.

Answer 4.8: 1, 2, and 4 are DAGs. 3 has a cycle (f,s,e,a).

Answer 4.9:

1. $[s, a, b, c]$, $[s, b, a, c]$, $[s, a, c, b]$.
2. $[s, a, d, c, b, e]$, $[d, c, s, a, e, b]$ or $[d, c, s, a, b, e]$
4. $[s, f, a, g, h, e, b, c, d]$

Bibliography

[1] Dfs, graph modeling. https://courses.cs.washington.edu/courses/cse417/21wi/lecture/06-DFS-model.pdf, 2021. Lecture 6.

[2] P.-Y. Chen. Chapter 22.1: Breadth-first search – clrs solutions. https://walkccc.me/CLRS/Chap22/Problems/22-1/, 2017–2024. Built by P.-Y. Chen, made with Material for MkDocs. Accessed: 2024-06-15.

[3] Rong Ge and Will Long (Scribe). Graph algorithms: Types of edges. https://courses.cs.duke.edu/fall17/compsci330/lecture12note.pdf, 2017. Lecture 12.