Lecture 2

Writing algorithms and graph traversal

Chinmay Nirkhe | CSE 421 Winter 2026

Algorithmic complexity

Measuring algorithmic efficiency

The RAM model

- RAM Model = “Random Access Machine” Model
- Each simple operation (arithmetic, evaluating if loop criteria, call, increment counter, etc.) takes one time step
- Accessing any one arithmetic number in memory takes one time step
- Measuring algorithm efficiency
  - Let input be $(x_1, \ldots, x_n)$ with each x_i representing one arithmetic number
  - Runtime of algorithm is the number of “simple operations” taken to compute algorithm in RAM model.

Complexity analysis

- Input $(x_1, \ldots, x_n)$ of length n.
- Multiple measures of complexity.
  - Worst-case: **maximum** # of steps taken on *any* input of length n
  - Best-case: **minimum** # of steps taken on *any* input of length n
  - Average-case: **average** # of steps taken over *all* input of length n

Complexity analysis

- The complexity of an alg. is a function $T(n)$ for each input size $n \in \mathbb{N}$.
- i.e. $T_{\text{worst}}(n)$ or $T_{\text{avg}}(n)$ could be two different functions.
- $T : \mathbb{N} \to \mathbb{N}$
- We are interested in understanding the overall behavior/shape of T, not the exact function.
- Sometimes there is more than one size parameter. $T(n, m)$ for a n vertex and m edge graph.

Polynomial time

A notion of efficiency

- A function $T(n)$ is **polynomial time** if $T(n) \leq cn^k + d$ for some constants $c, k, d > 0$.
  - Let k be the minimal such value. This is the degree of the *dominating* polynomial.
  - Polynomial time is known as "efficient" in theoretical CS.

Polynomial time

A notion of efficiency

- A function $T(n)$ is **polynomial time** if $T(n) \leq cn^k + d$.
- Why **polynomial time**?
  - Scaling the instance by a constant factor also scales $T(n)$ by a constant.
  - If $T(n) = cn^k + d$ then $T(2n) = c(2n)^k + d \leq 2^k(cn^k + d) = 2^kT(n)$.
  - **Church-Turing thesis**: Any function computable in polynomial time by a physically realizable model of computation can also be computed in polynomial time a *different* physically realizable model.
    - I.e. polynomial-time is a notion independent of model of computation.
    - Ideal for theoretical study of what problems are efficient and which are not.
  - Problem size grows by constant, then running time also grows by constant.
  - Typically, polynomials for common algorithms are small polynomials cn, cn^2, cn^3, cn^4. Rarely anything higher.

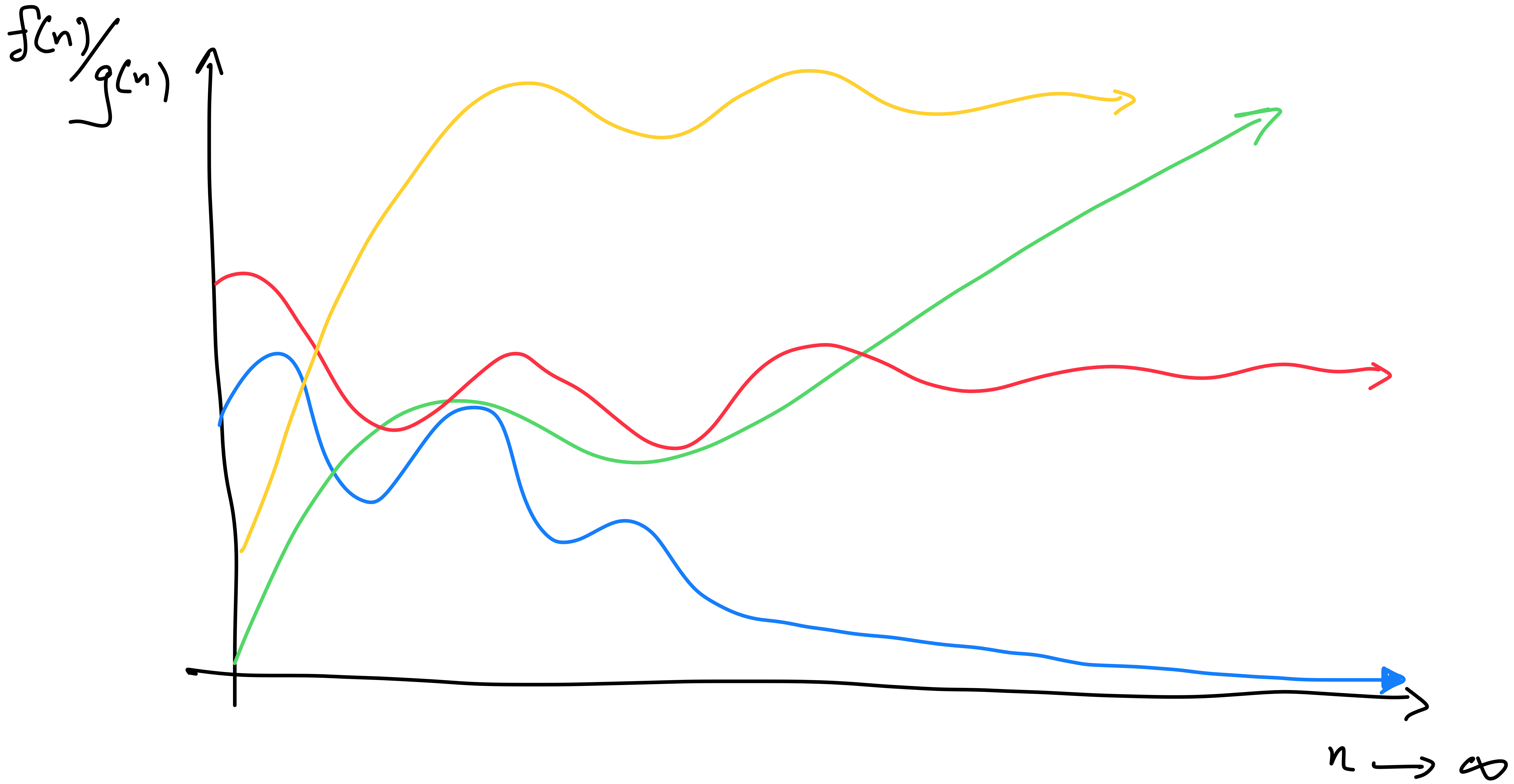
Big-O notation

Let $T, g : \mathbb{N} \to \mathbb{N}$. Then

- $T(n)$ is $O(g(n))$ if $\exists\ c, n_0 > 0$ such that $T(n) \le cg(n)$ when $n \ge n_0$.
- $T(n)$ is $o(g(n))$ if $\lim_{n\to\infty} \frac{T(n)}{g(n)} = 0$.
- $T(n)$ is $\Omega(g(n))$ if $\exists\ \epsilon, n_0 > 0$ such that $T(n) \ge \epsilon g(n)$ when $n \ge n_0$.
- $T(n)$ is $\Theta(g(n))$ if $T(n)$ is $O(g(n))$ **and** $T(n)$ is $\Omega(g(n))$.

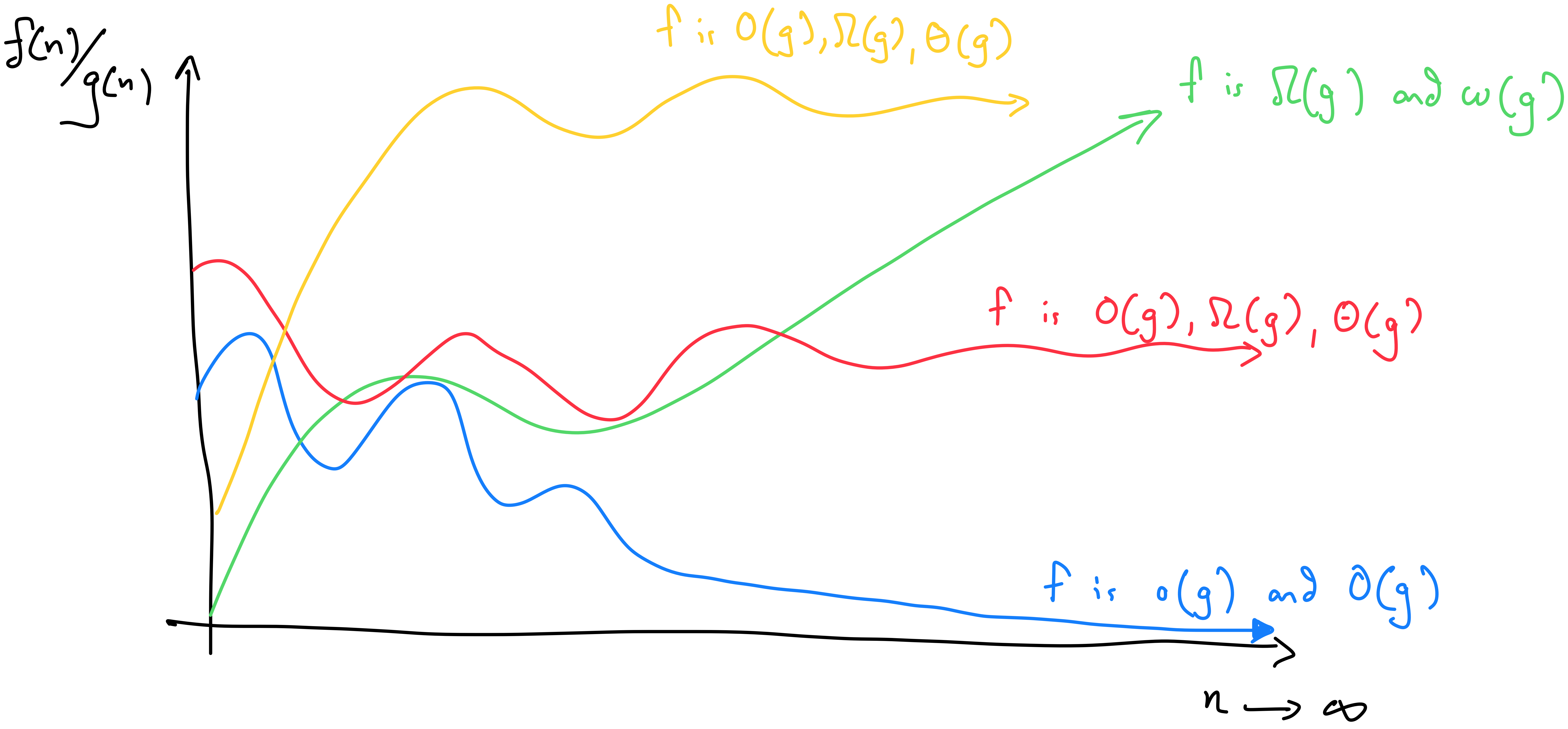
Big-O notation

Cartoon



Big-O notation

Cartoon



Measuring algorithmic efficiency

The RAM model

- RAM Model = “Random Access Machine” Model
- Each simple operation (arithmetic, evaluating if loop criteria, call, increment counter, etc.) takes one time step
- Accessing any bit of memory takes one time step

Measuring algorithmic efficiency

The RAM model, Examples

- Sorting a list of integers $L = (x_1, \ldots, x_n)$
  - You probably know that sorting can be solved in $\Theta(n \log n)$ time by algorithms such as merge sort.
  - This is measuring the number of comparisons $x_i < x_j$ that we are making. RAM model makes this rigorous.
- All-pairs shortest path problem: Given a weighted graph $G = (V, E)$ output $d_{uv} = \min_{p:u \rightsquigarrow v} \sum_{(a,b) \in p} w_{ab}$ for every pair of vertices $u, v \in V$.
  - Floyd-Warshall alg. Makes $O(n^3)$ arithmetic comparisons where $n = |V|, m = |E|$.
  - Requires adjacency matrix access to the graph. Meaning, unit cost to compute w_{ab} for any $a, b \in V$.

Graph traversal

Graph search and traversal

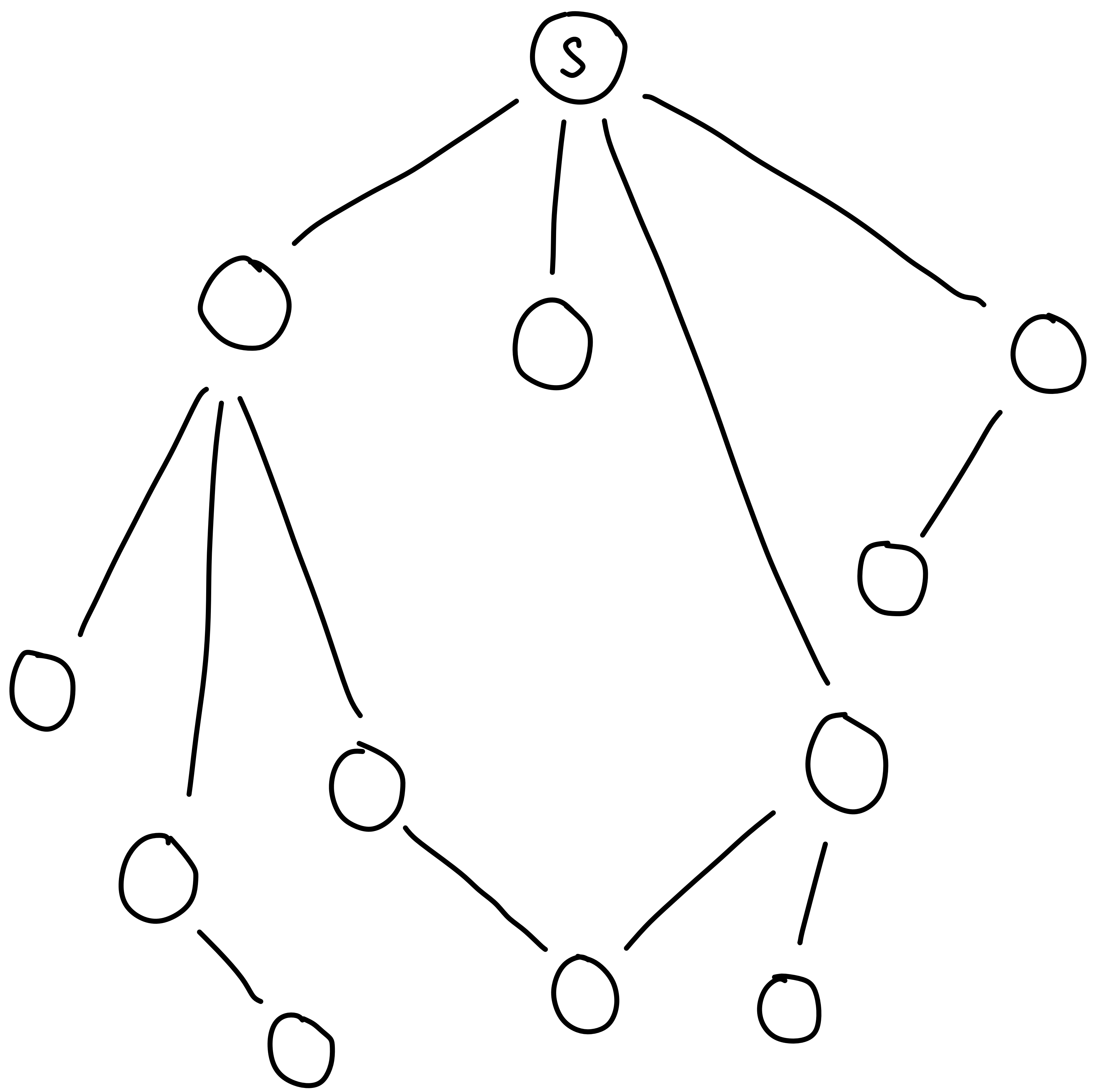
- Used to discover the structure of a graph
- "Walk" from a fixed starting vertex s ("the source") to find all the vertices reachable from s
- **Generic traversal algorithm.**
  - **Input:** Graph G and vertex $s \in V$
  - **Find:** set $R \subseteq V$ reachable from s

Reachable(s):

$R \leftarrow \{s\}$
While there exists a $(u, v) \in R \times (V \backslash R)$
  Add v to R: $R \leftarrow R \cup \{v\}$.
return R

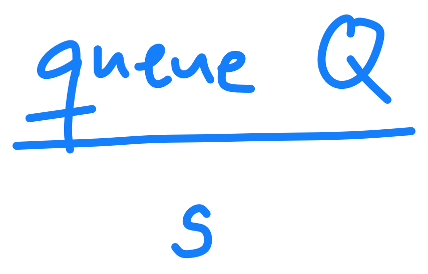
Breadth-first search (BFS)

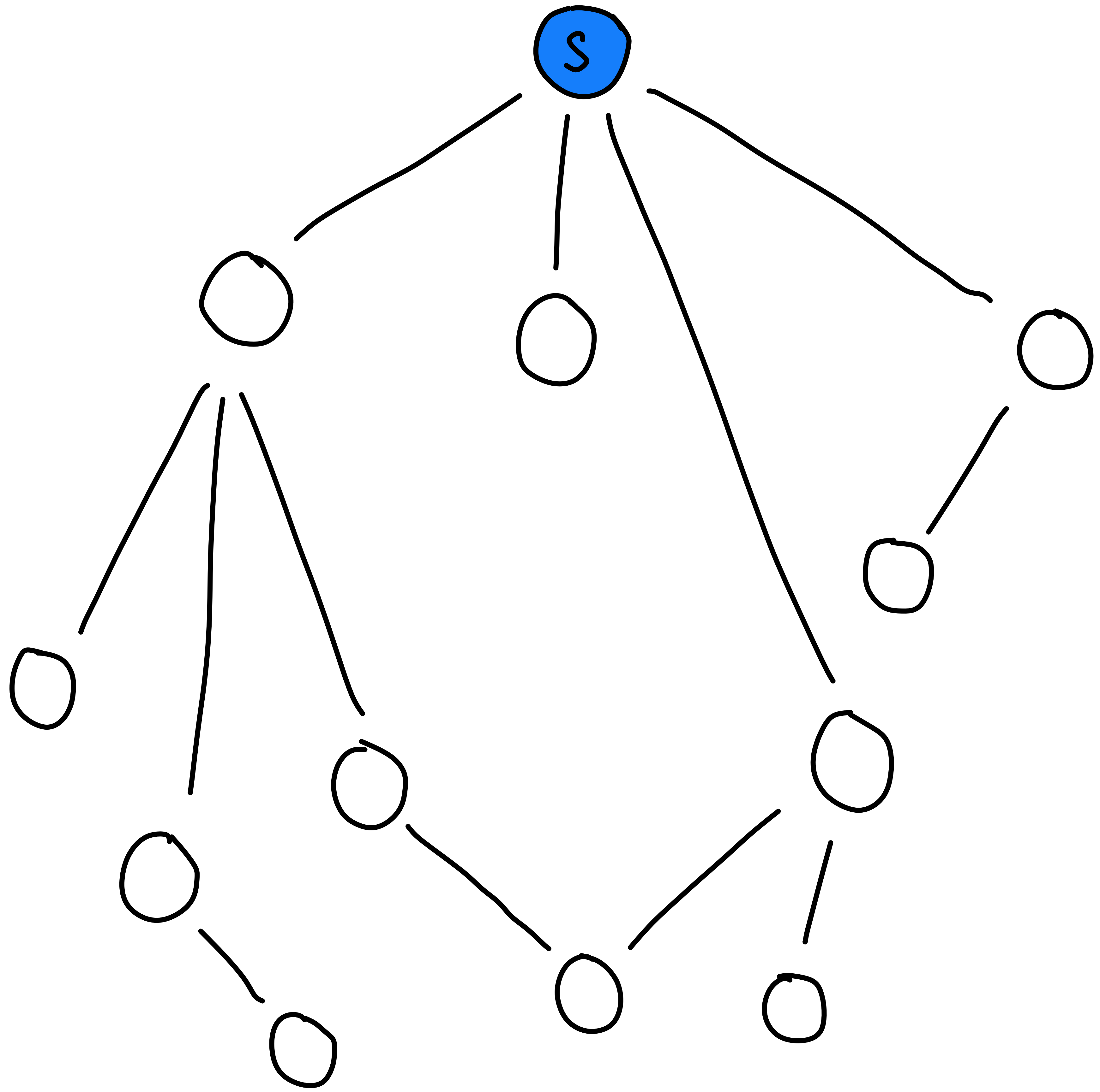
- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.



Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.



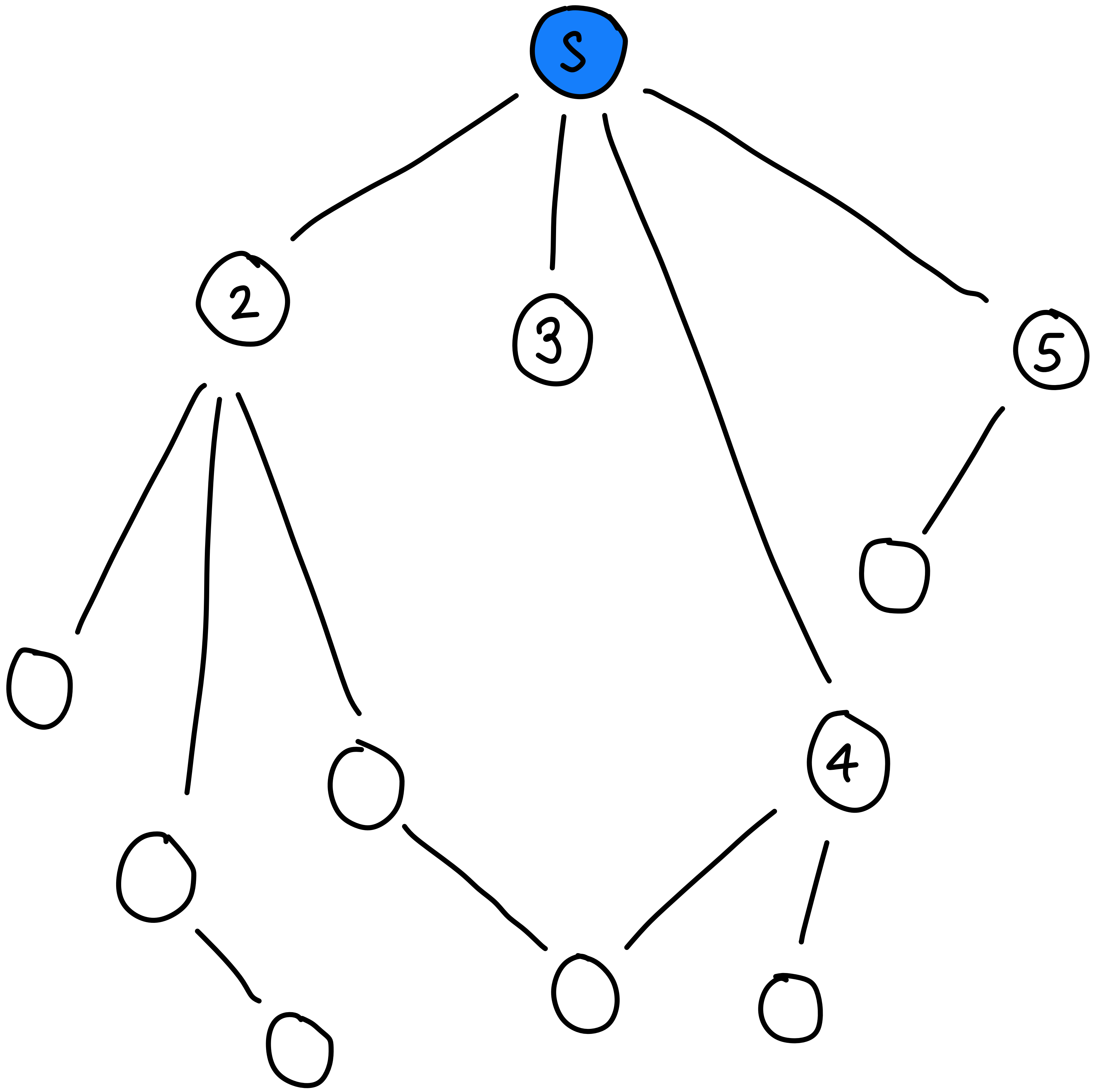


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

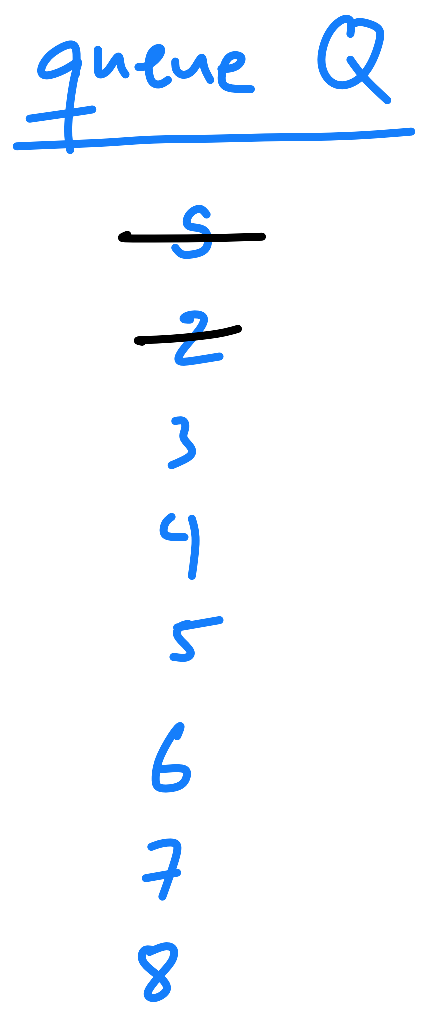
queue Q

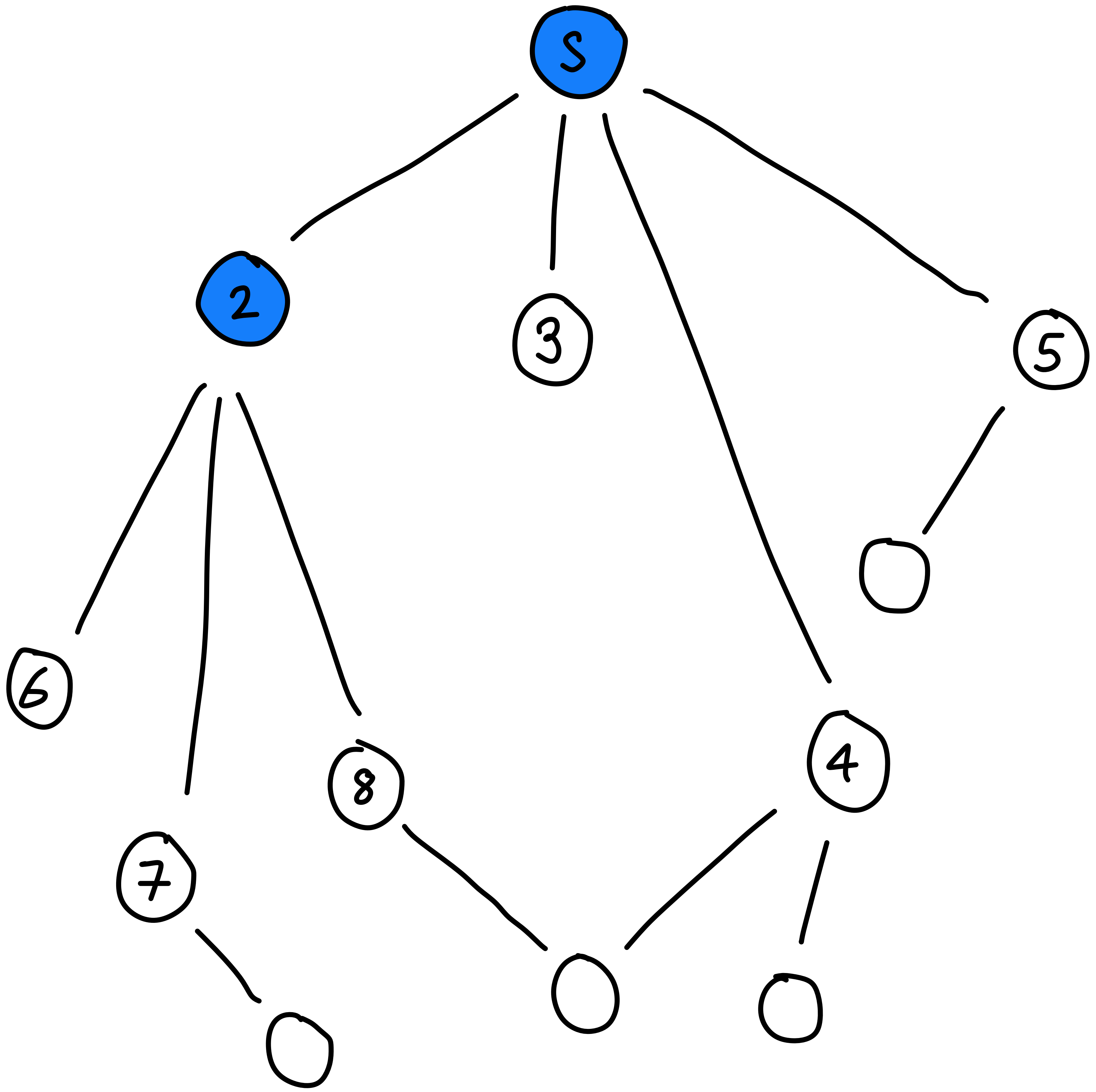
~~s~~
2
3
4
5



Breadth-first search (BFS)

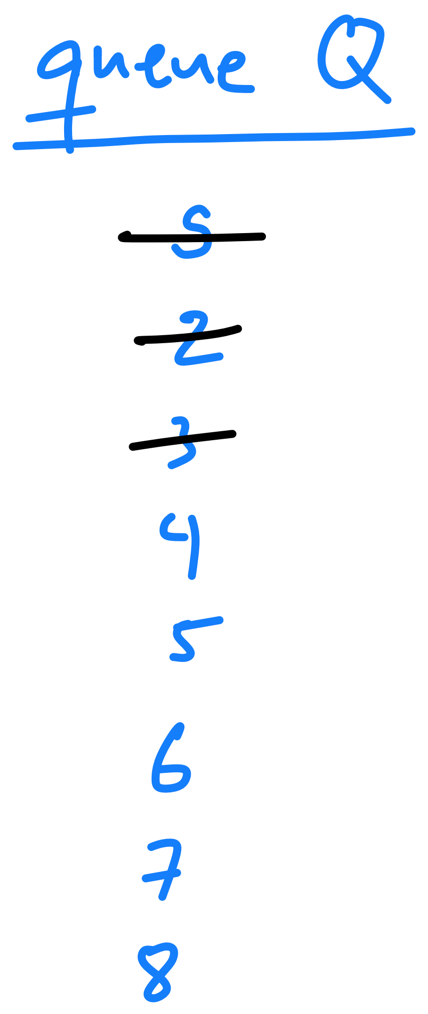
- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.

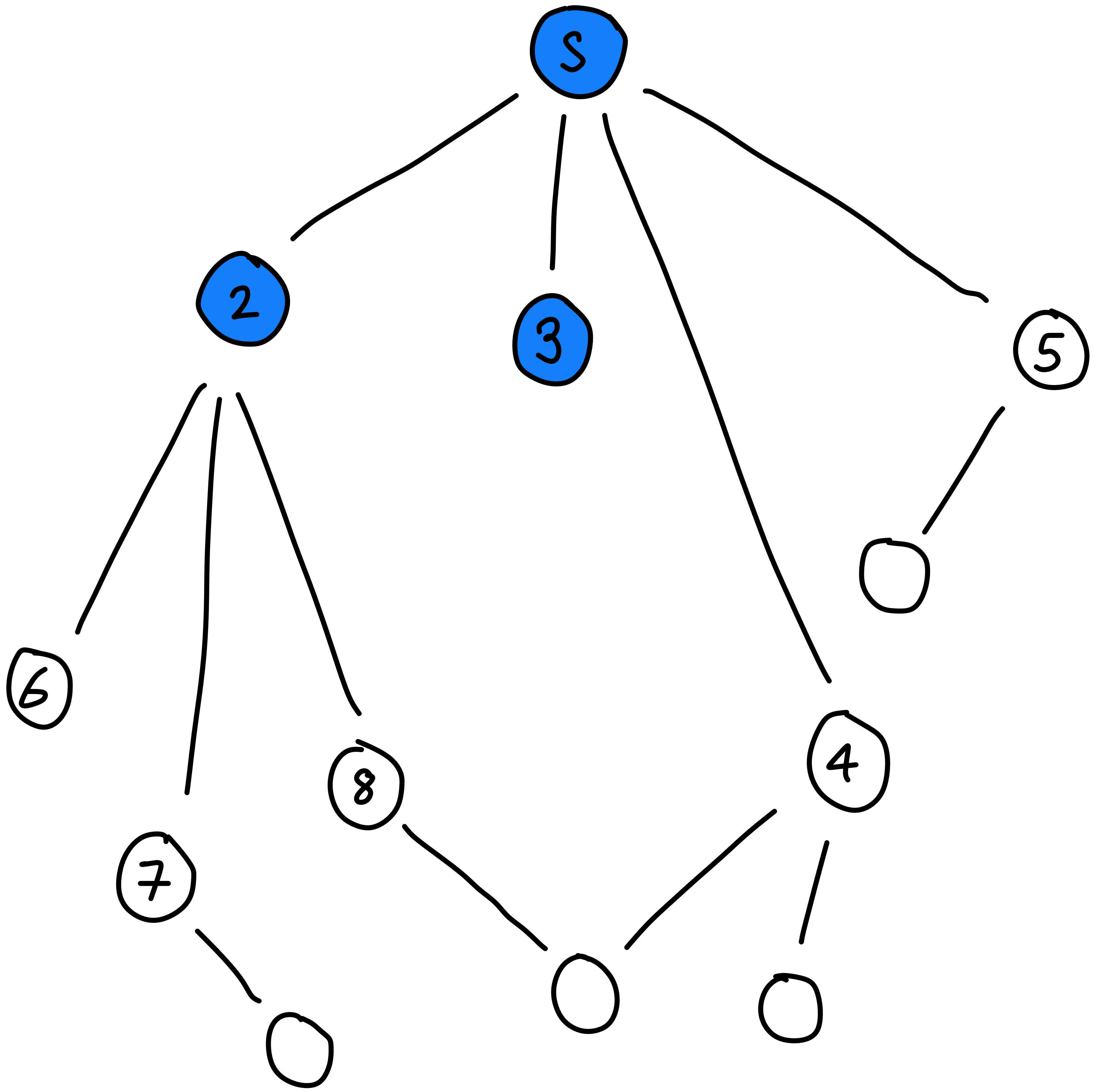




Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.



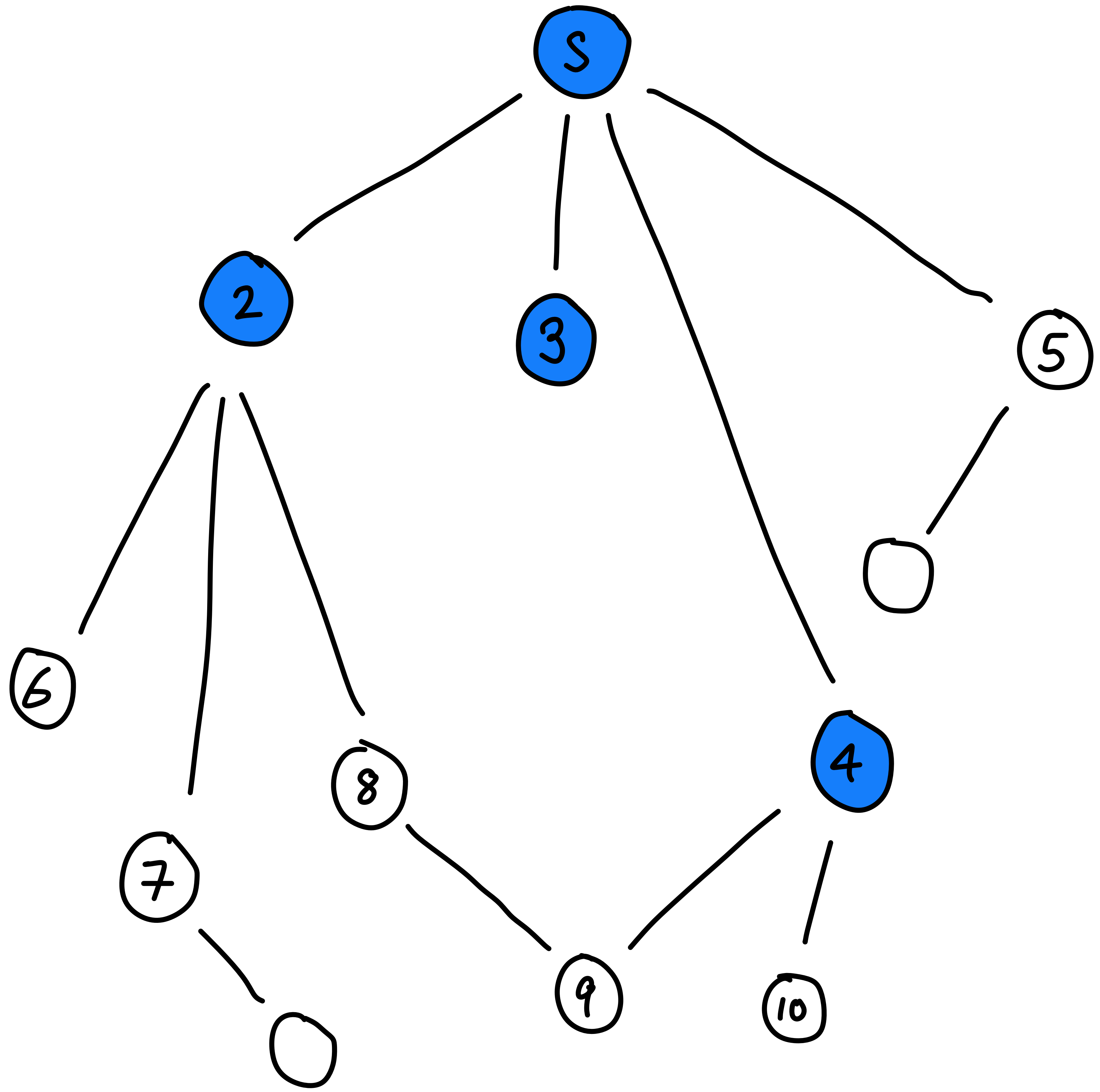


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

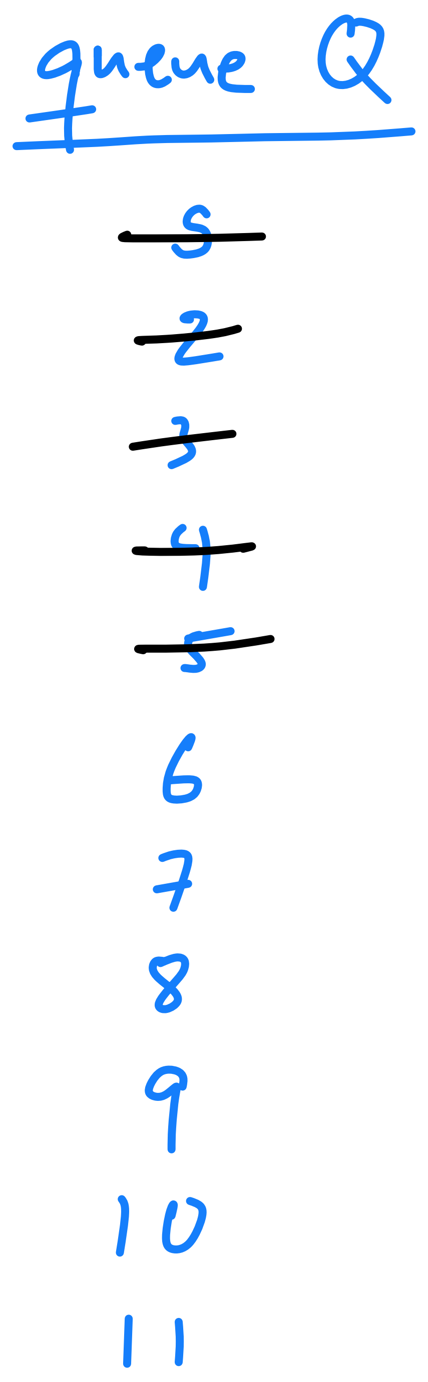
queue Q

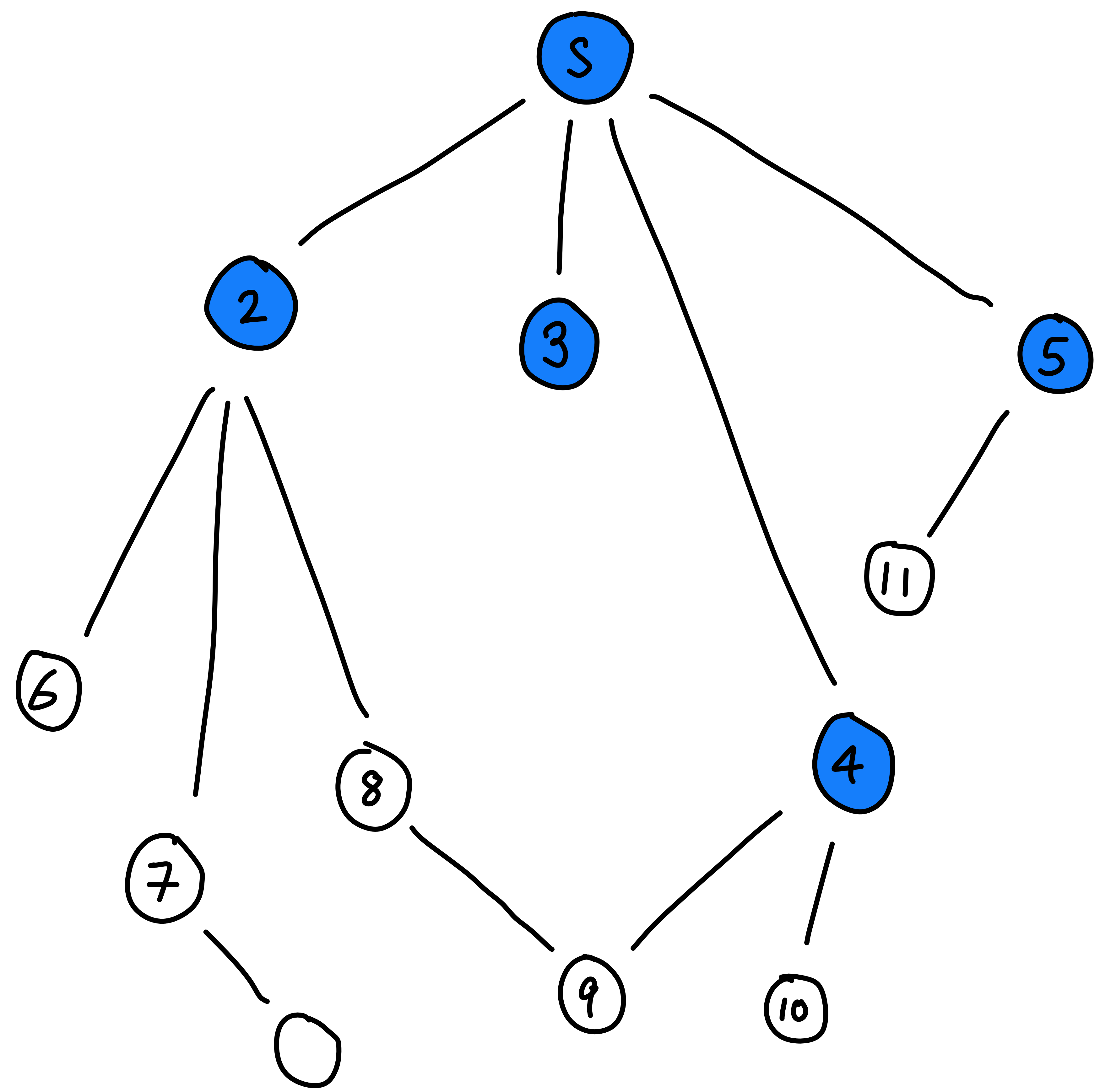
s
2
3
4
5
6
7
8
9
10



Breadth-first search (BFS)

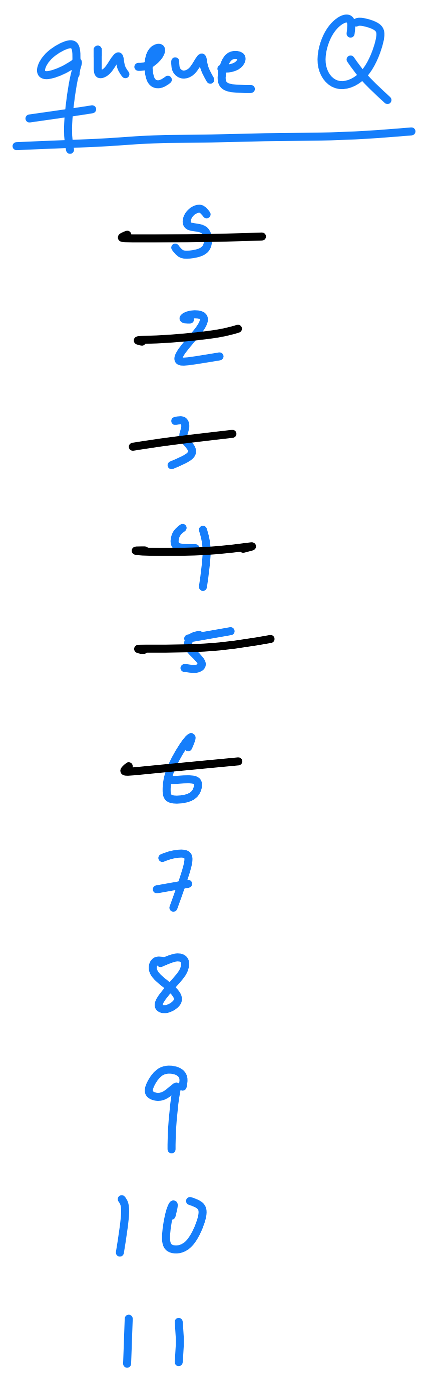
- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

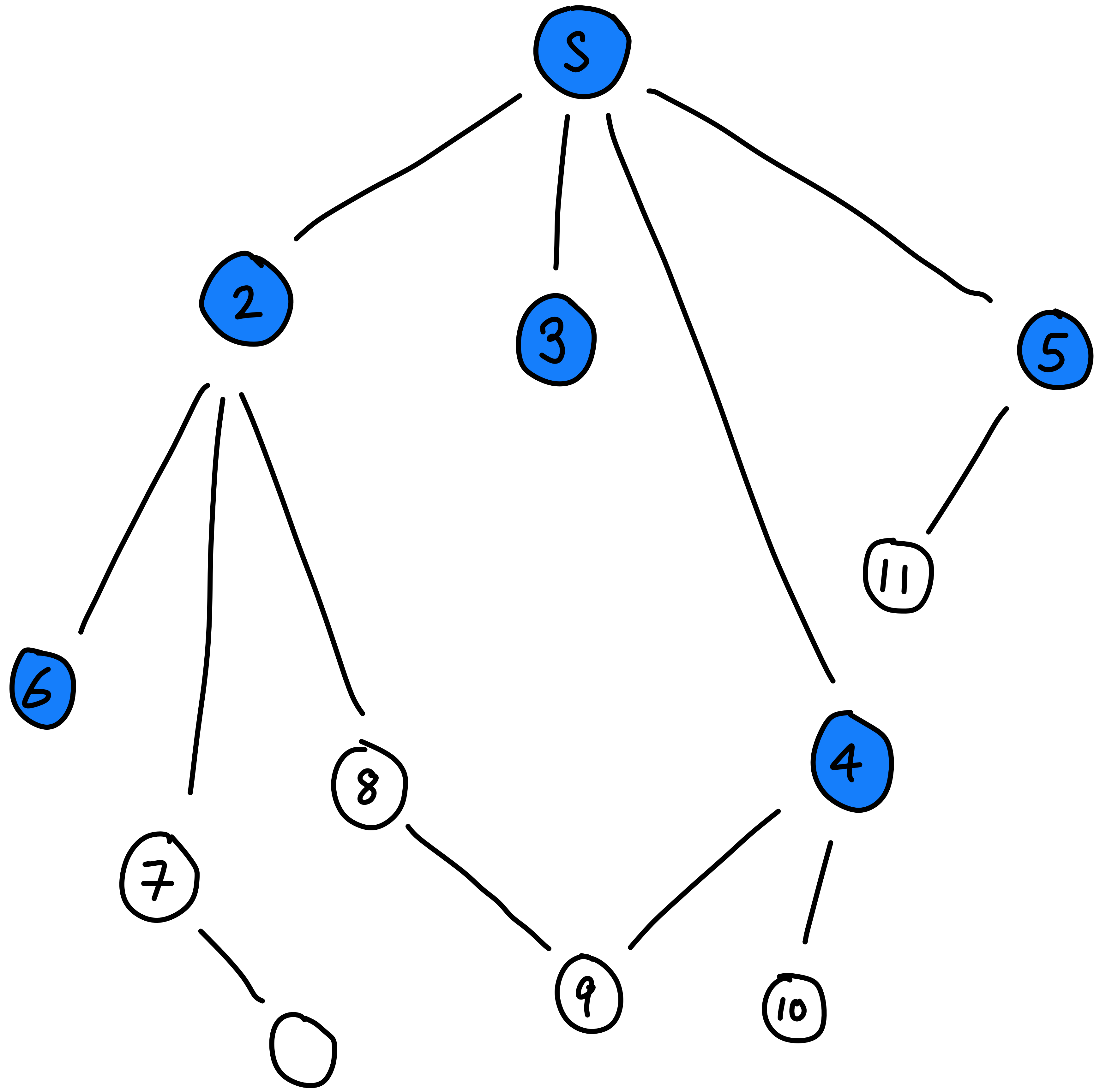




Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.



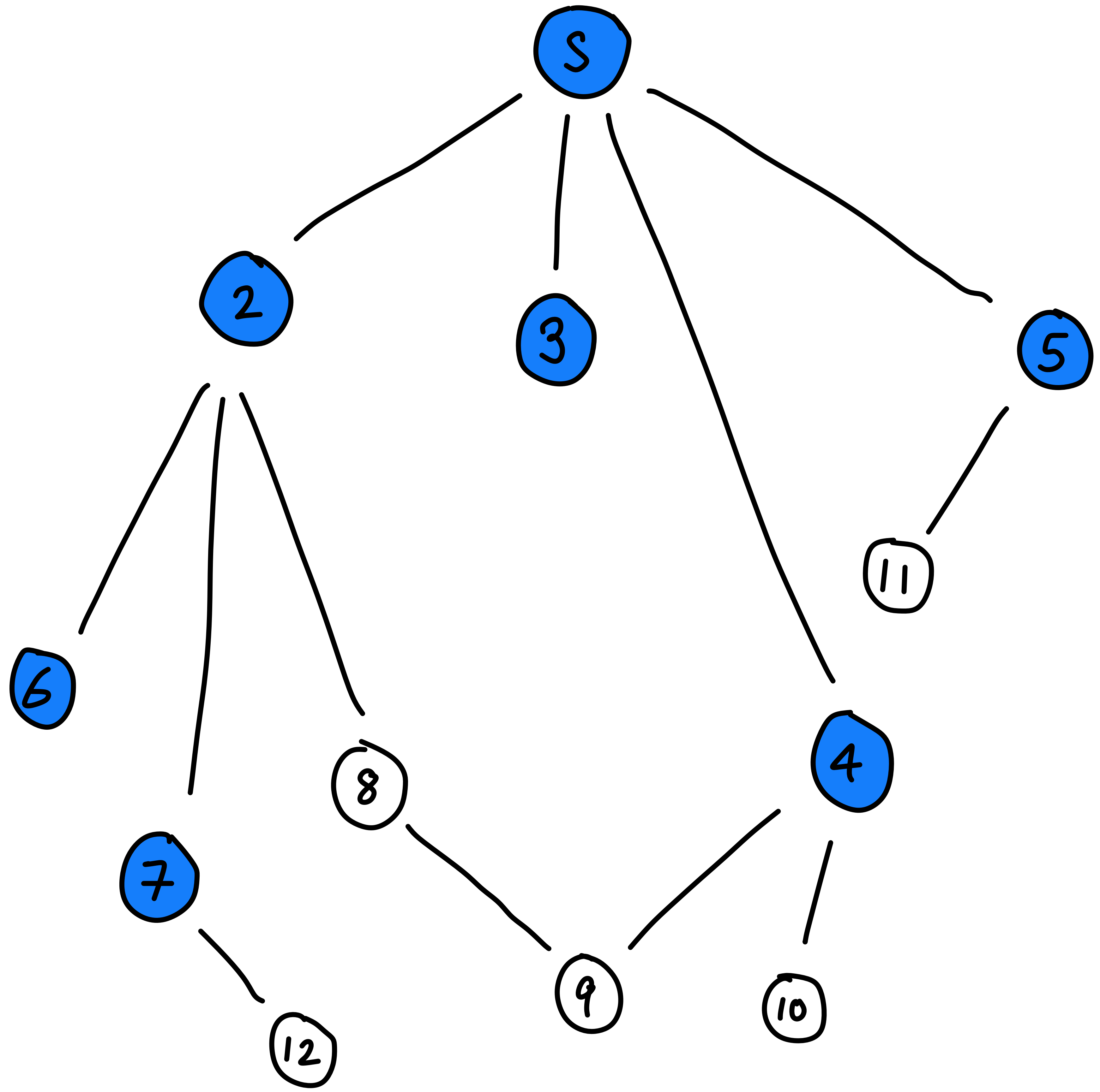


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

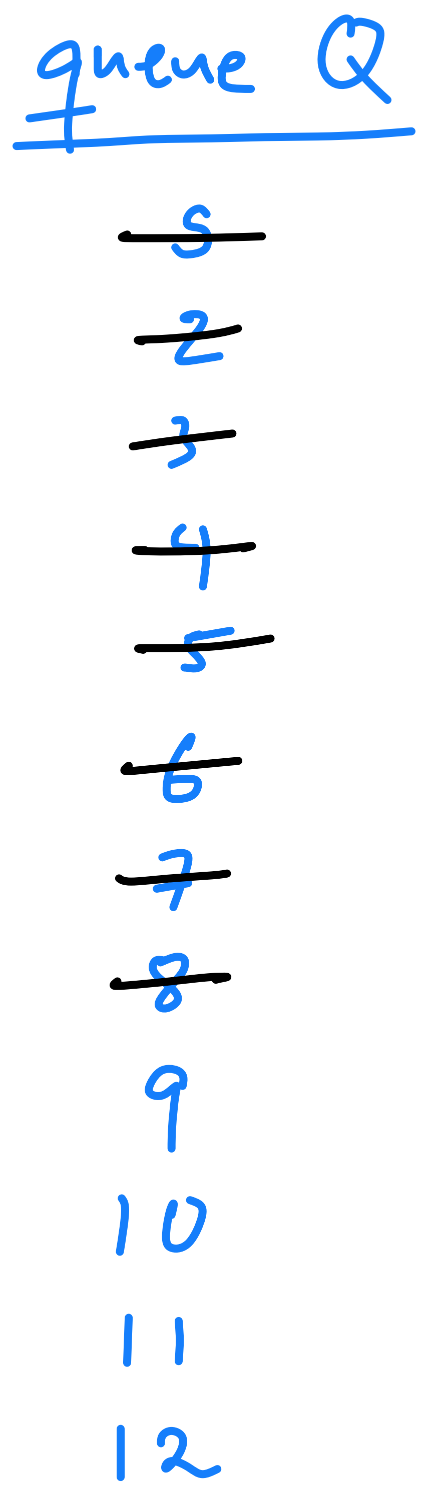
queue Q

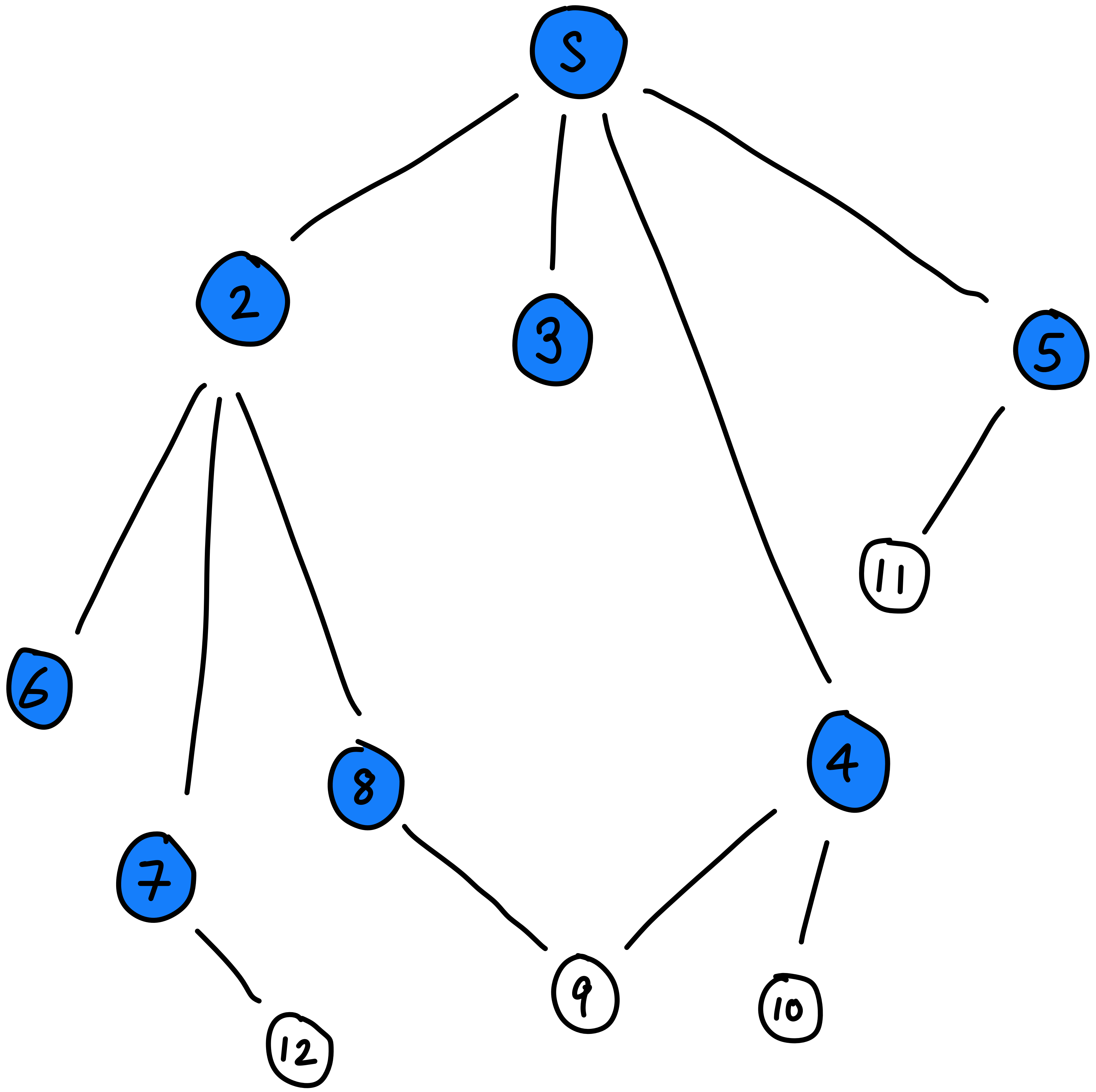
~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
8
9
10
11
12



Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.



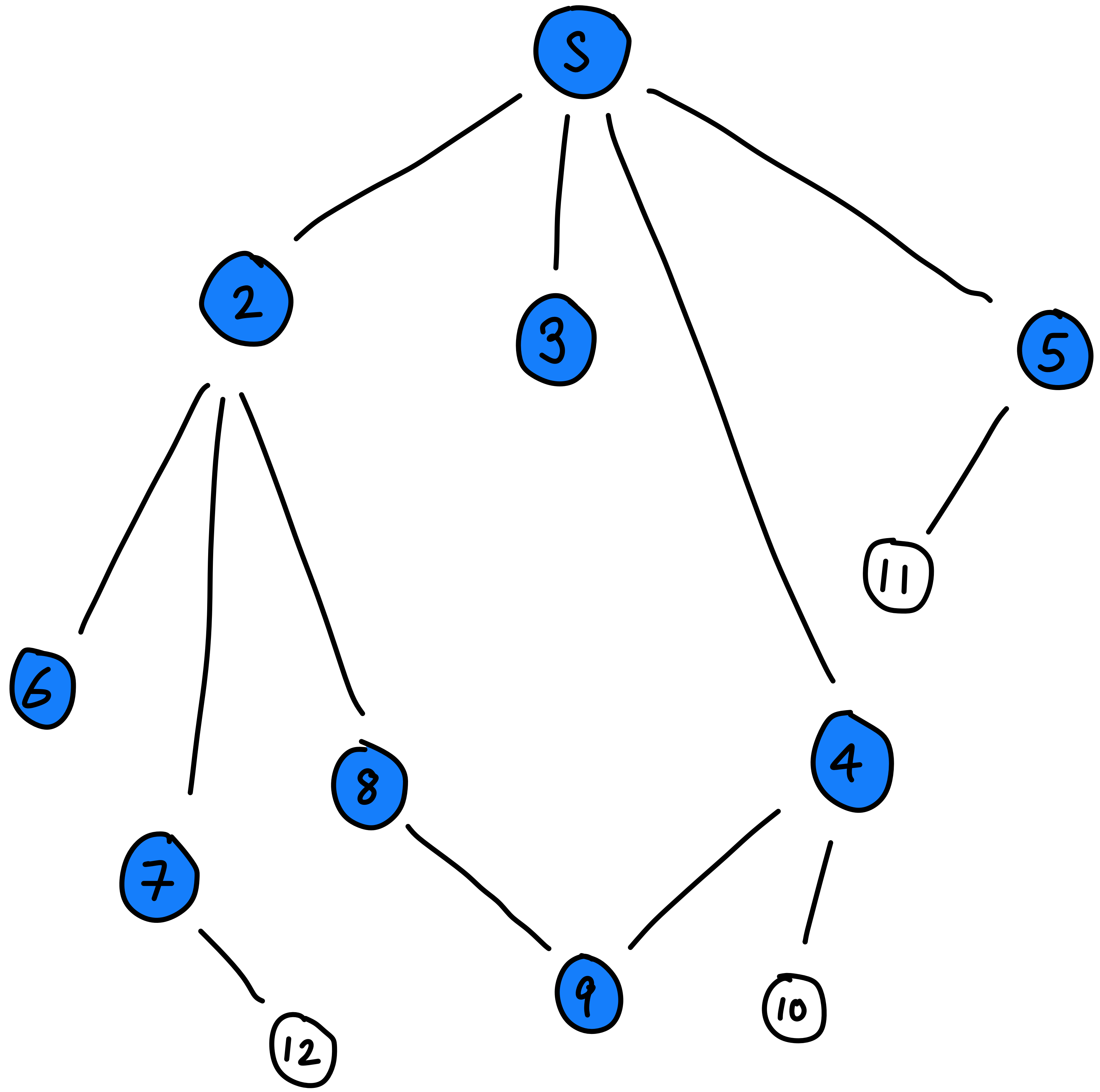


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

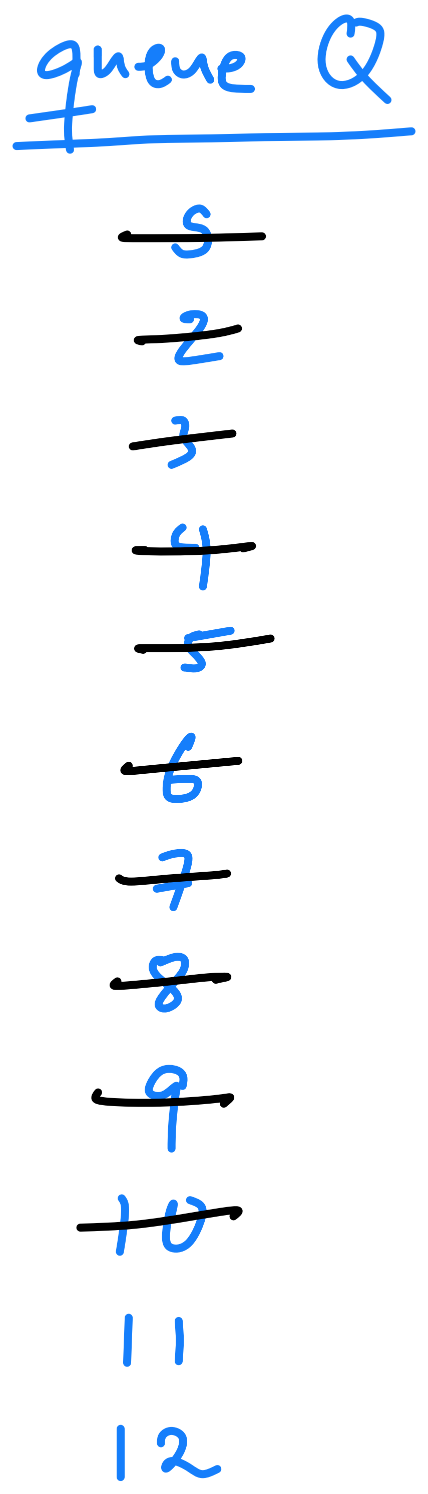
queue Q

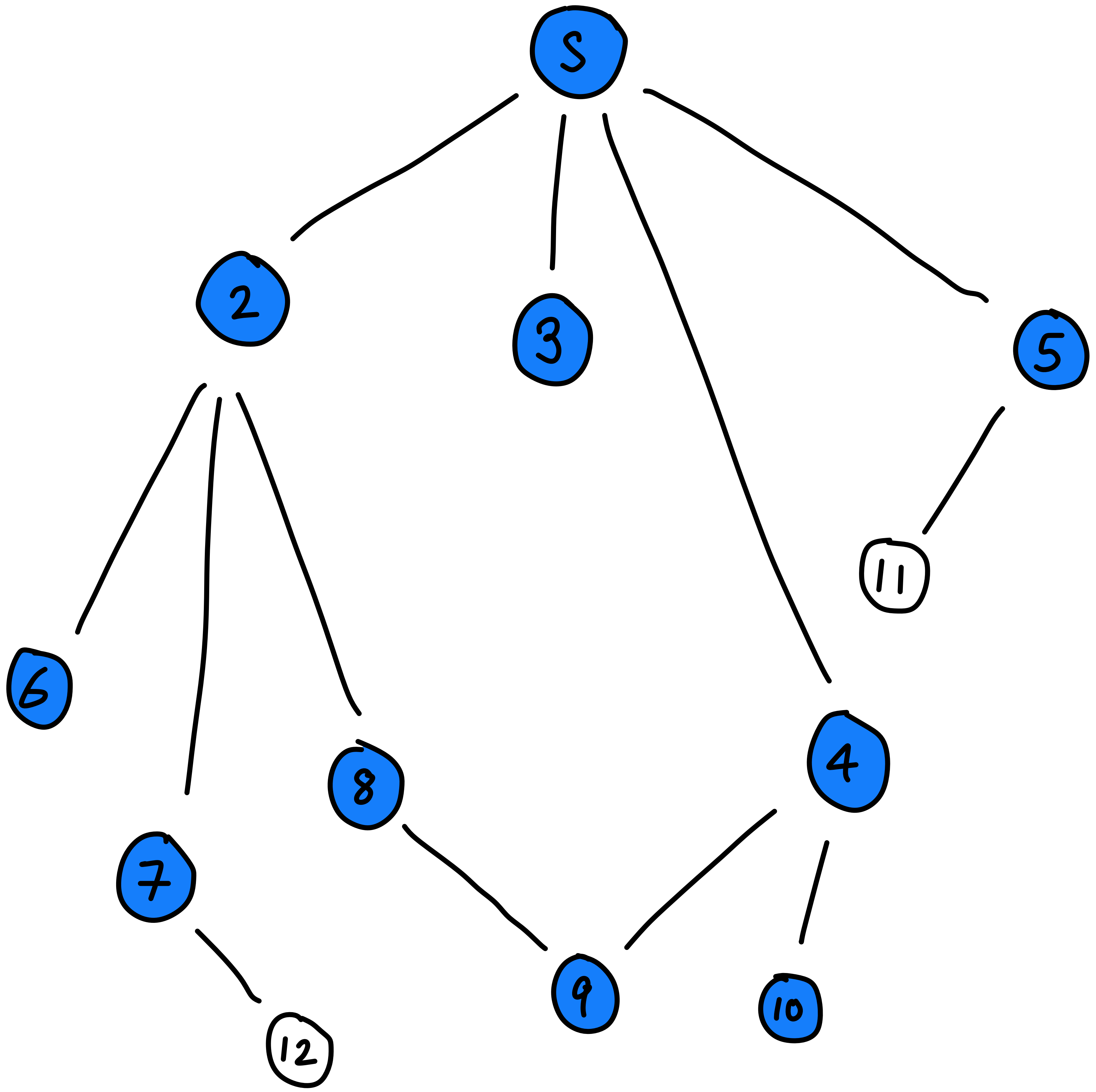
~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
10
11
12



Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.



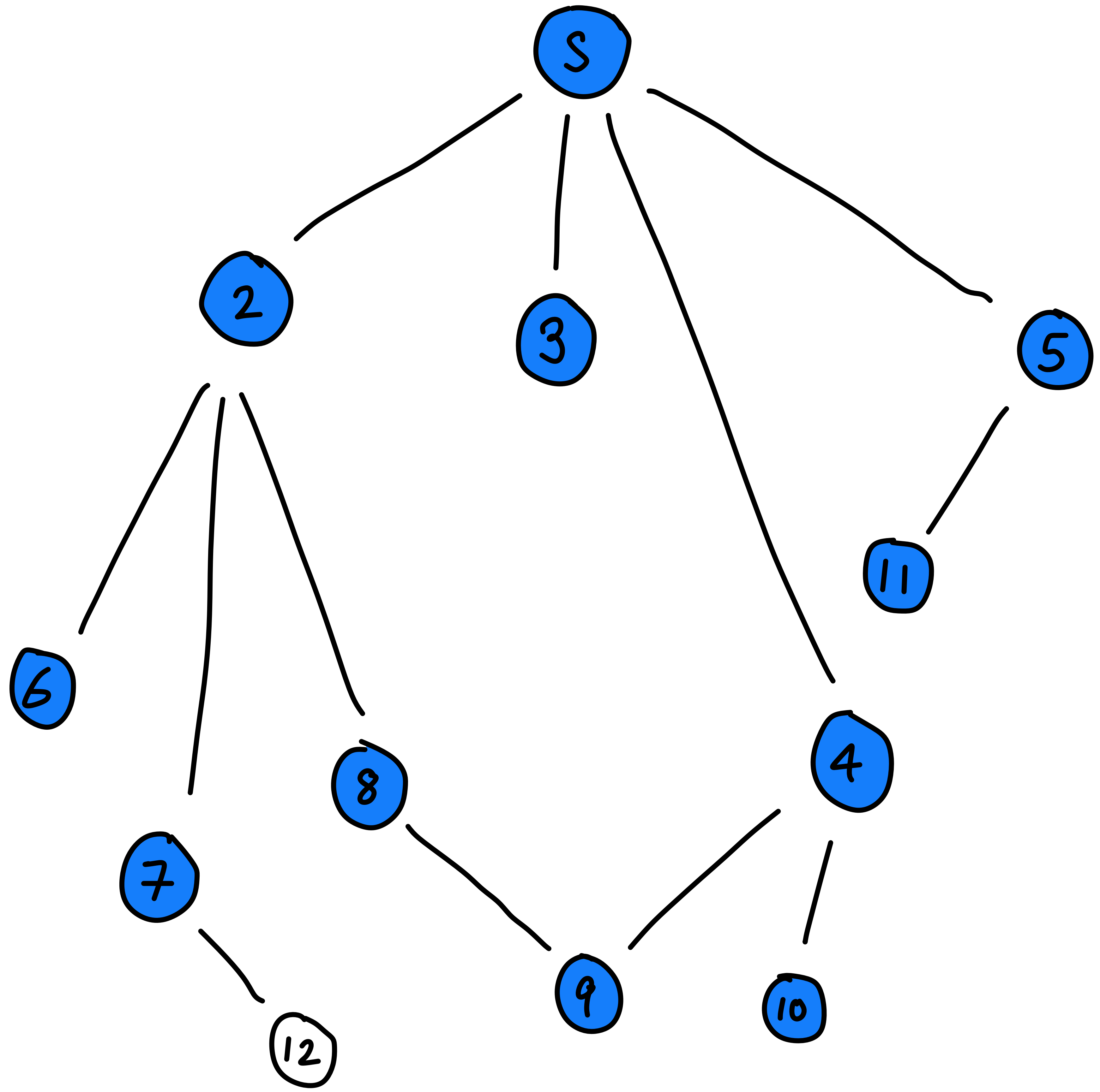


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

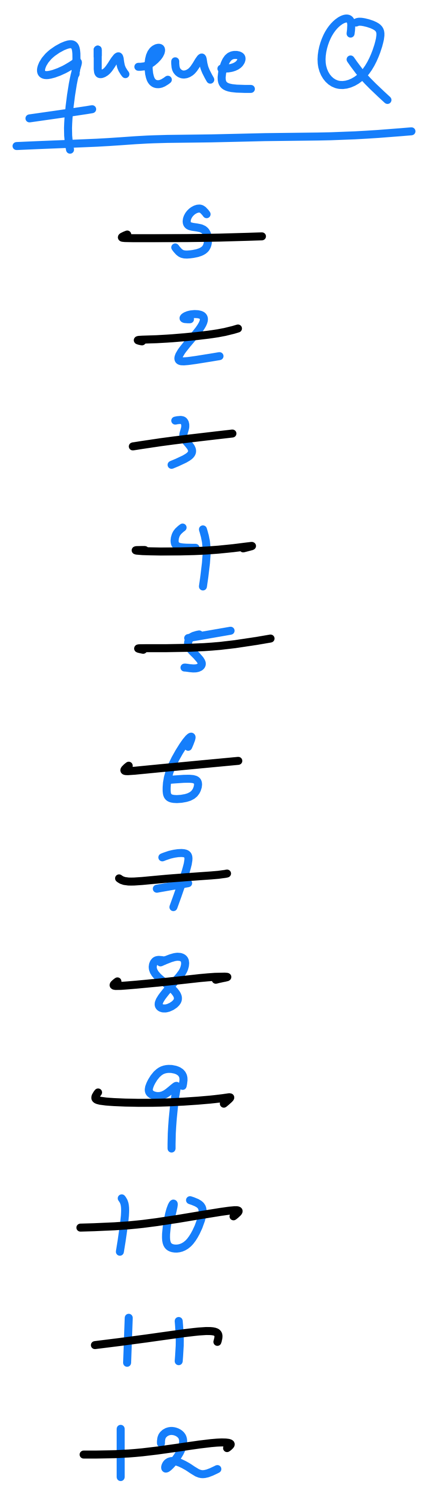
queue Q

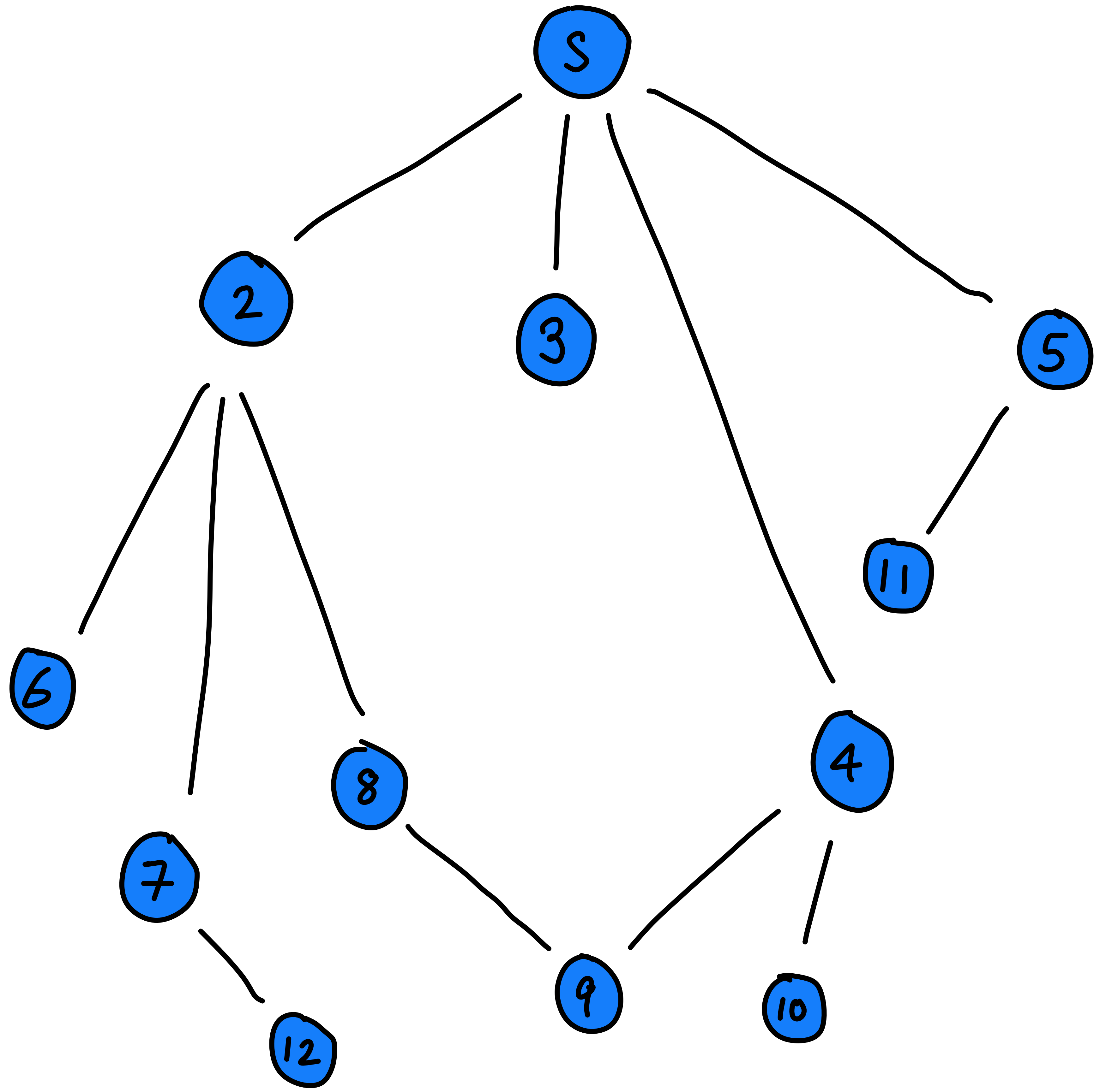
~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
~~10~~
~~11~~
12



Breadth-first search (BFS)

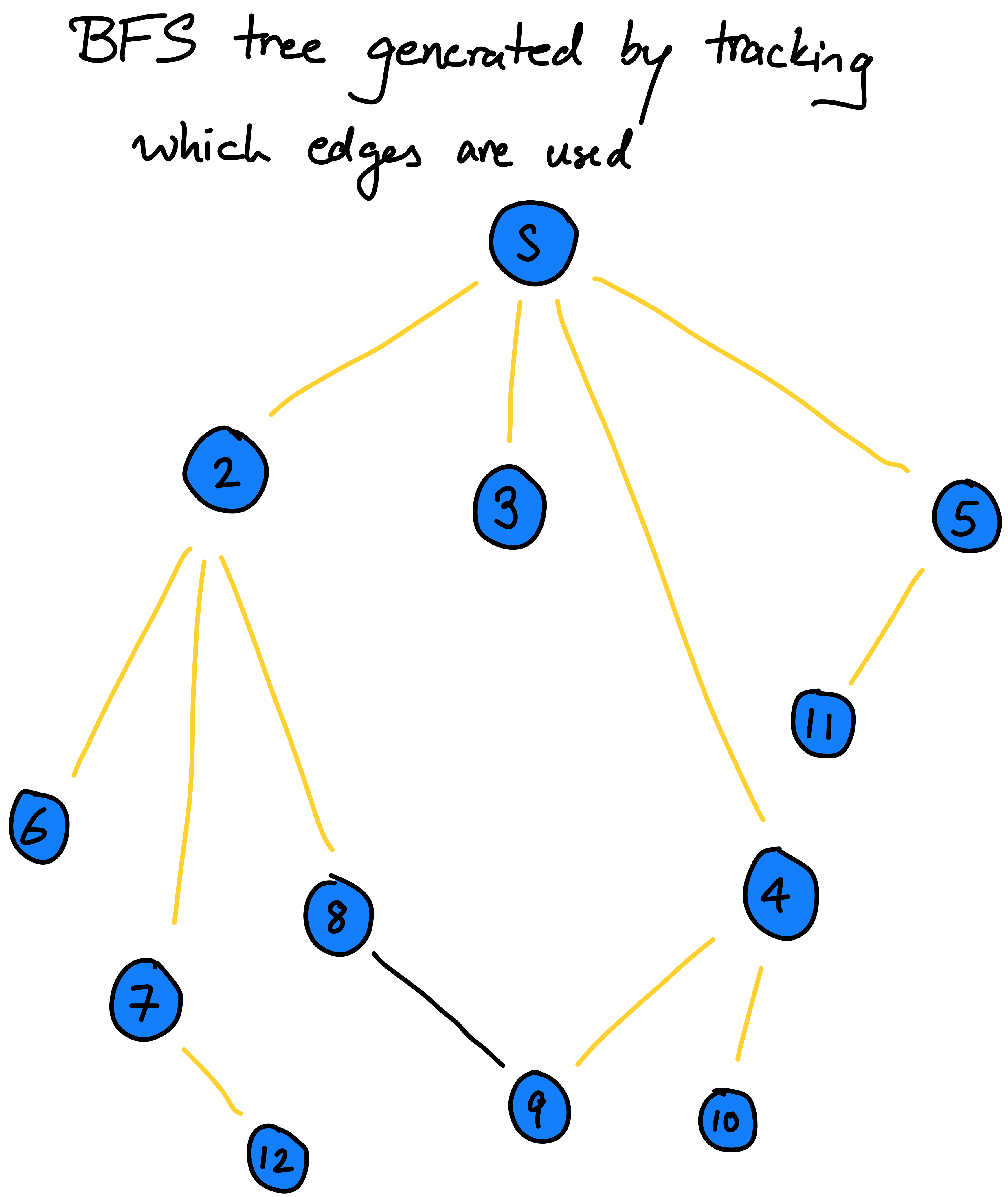
- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.





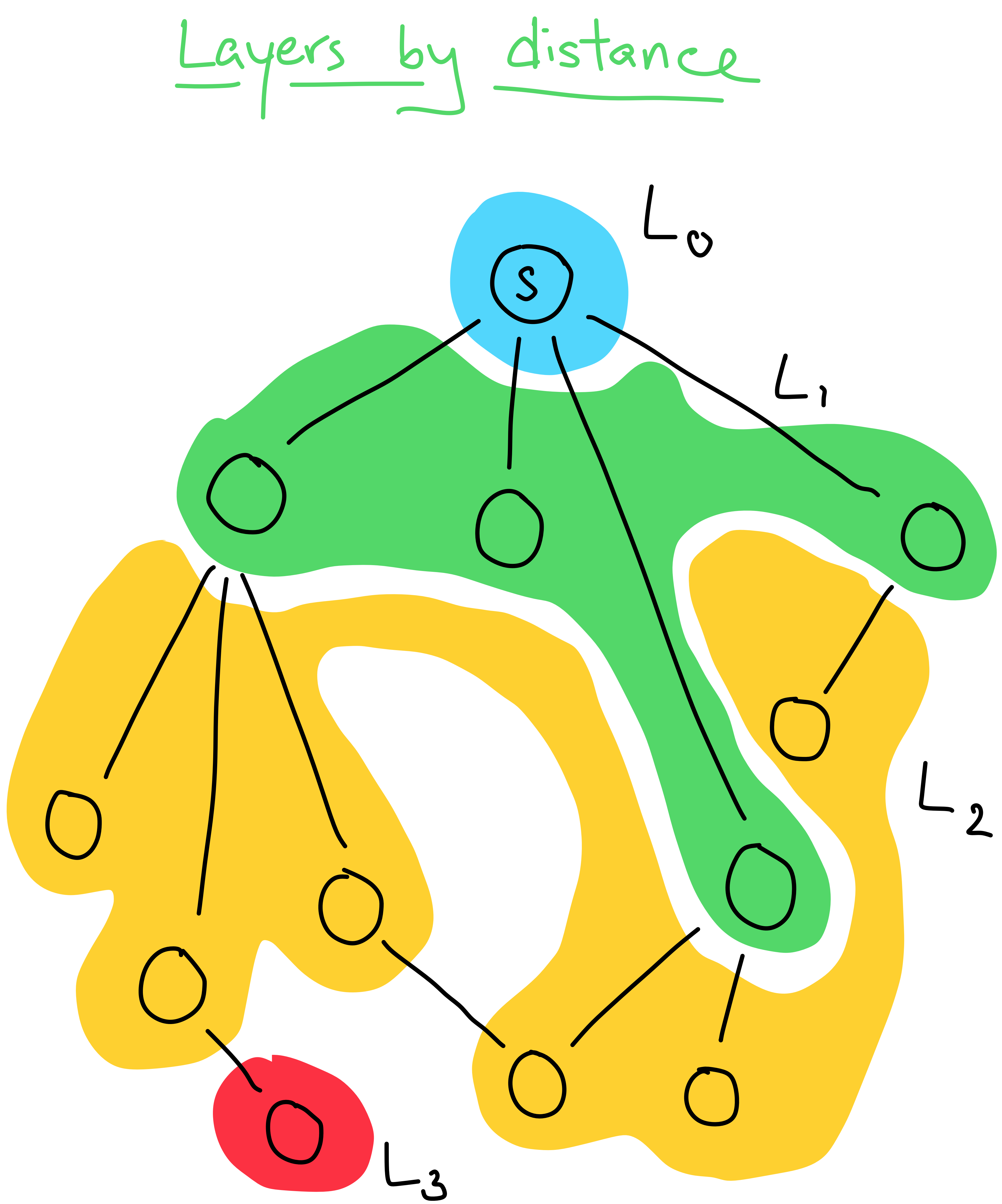
Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.



Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.



Graph search and traversal

- Used to discover the structure of a graph
- "Walk" from a fixed starting vertex s ("the source") to find all the vertices reachable from s

- **Generic traversal algorithm.**
  - **Input:** Graph G and vertex $s \in V$
  - **Find:** set $R \subseteq V$ reachable from s

Reachable(s):

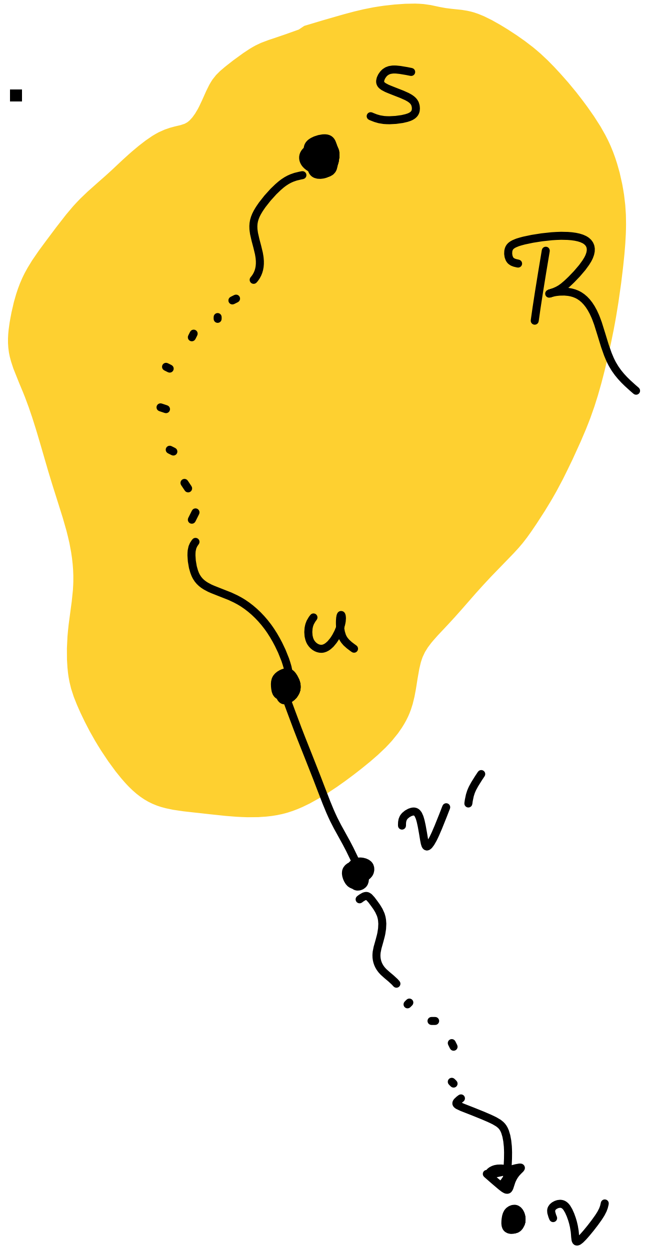
$R \leftarrow \{s\}$
While there exists a $(u, v) \in R \times (V \backslash R)$
  Add v to R: $R \leftarrow R \cup \{v\}$.
return R

Generic graph traversal

Reachable(s):

$R \leftarrow \{s\}$
While there exists a $(u, v) \in R \times (V \backslash R)$
  Add v to R: $R \leftarrow R \cup \{v\}$.
return R

- **Claim:** R is exactly the set of reachable vertices.
- **Proof:** We show both directions. (1): every vertex in R is reachable. (2): every reachable is in R.
  - **Direction 1**. For $v \in R$, there is a path $s \rightsquigarrow v$. Proved by induction on the generic graph traversal algorithm: If we added v by edge $(u, v) \in R \times (V \backslash R)$ then $s \rightsquigarrow u \rightarrow v$.
  - **Direction 2**. Assume (for $\perp$), there is a vertex v that is reachable but not $v \notin R$.
    - Let $p =$ the path $s \rightsquigarrow v$ and let v' be the **first** vertex on p such that $v' \notin R$.
    - Then u, the predecessor of v', satisfies $u \in R$ and $(u, v') \in R \times (V \backslash R)$.
    - Contradicts the definition of the generic graph traversal.



How to write algorithms and proofs

- The goal of writing an algorithm is to explain to another computer scientist how to algorithmically solve a particular problem **and** why the algorithm is correct/works.
- The goal is not to write **pseudocode** for the algorithm.
- A competent programmer should be able to take your answer and have an exercise in programming to generate an implementation in any programming paradigm.
- Your answer will also include a proof of correctness. More on this soon.

The three steps to an algorithm

Step 1: The algorithm

- Explain the steps necessary to implement the algorithm but not all the details
- This is similar to how you would right a lab report in a chemistry or physics lab today compared to what you would write in grade school.
  - The level of precision is different because you are writing to a different audience.
  - You are writing for a human audience. Don’t write C code, Java code, Python code, or any code for that matter. Write plain, technical English.

The three steps to an algorithm

Step 1: The algorithm

- For example, if you want to set m as the max of an array A,
  - **do not** write a *for* loop to find the max.
  - Instead use math notation: $m \leftarrow \max_{x \in A} x$
- Don’t spend an inordinate time trying to find ‘off-by-one’ errors.
- Use simplifications such as ‘apply X here’ or ‘modify X by doing (…)’

The three steps to an algorithm

Step 2: The runtime

- Runtime analysis will be the easiest step of your solution.
- Use big-O notation when analyzing the runtime.
- Don't forget that data structures don't magically whisk away complexity!
  - For example, min heaps have $O(1)$ time to find the minimum and $O(\log n)$ to add an element.
  - Make sure to analyze any novel data structures you construct!

The three steps to an algorithm

Step 3: Correctness

- This will be the hardest step for most of you.
- When proving the correctness of an optimization algorithm, you need to prove
  - (a) why the output is feasible — ex. a valid assignment/schedule/etc.
  - (b) why no other output is better.
- Proving (b) is often much harder than (a). We will see how to do this with induction proofs, proofs by contradiction, and much more.
- Let’s see more via example.

A writeup for breadth-first search

- **Input**: an undirected graph $G = (V, E)$ and a starting root s
- **Output**: A tree T such that $d_T(s, v) = d_G(s, v)$ for any vertex $v \in G$. (For any unreachable vertex v , by convention, $d_G(s, v) = \infty$ and v is not included in T.)
- **Algorithm:**
  - **Details:** Initialize a queue Q with s and empty tree T. While Q isn't empty, pop v off and mark as visited. Then and add all unvisited neighbors w of v to the queue and add edge (v, w) to T.
  - **Runtime:** Each edge and vertex is visited/referenced at most $O(1)$ times so total complexity is at most $O(|V| + |E|)$.