Lecture 9

Instructor: Subrahmanyam Kalyanasundaram

16th September 2019

Plan

- Last class, we saw B-trees and Binary (max) Heaps

Plan

- Last class, we saw B-trees and Binary (max) Heaps
- Today, we see BuildHeap
- After that, we see graphs

Heaps

Heap

A max-heap supports the following functions:

- INSERT(*val*) – Inserts *val* into the heap.
- EXTRACTMAX() – Returns and removes the maximum element from the heap.

Binary max heap

A binary max-heap satisfies the following properties:

1. Structural Property: Is a complete binary tree except possibly for the lowest level, which is "left-filled".
2. Heap Property: The value of a node is greater than that of both its children.

Binary max heap

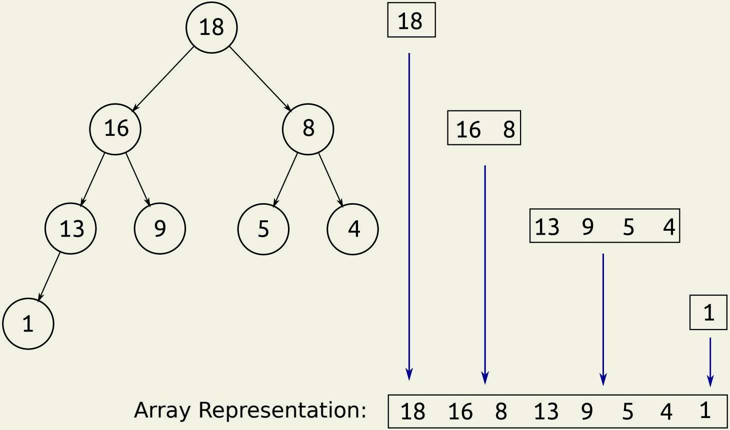
A binary max-heap satisfies the following properties:

1. Structural Property: Is a complete binary tree except possibly for the lowest level, which is "left-filled".
2. Heap Property: The value of a node is greater than that of both its children.

Note: It is not a search tree.

Data Structure

Read off from top to bottom, left to right.



Questions

About Heaps:

1. How many nodes does a height h heap have? (both bounds)
2. What is the maximum height of a heap with n nodes?

About the array implementation:

1. What is the array index of the children of the node at $A[i]$?
2. What is the array index of the right sibling of the node at $A[i]$?

Heaps using arrays

Typically, a heap is built starting with an arbitrary array:

- Procedure BuildHeap(Array A) – Takes an array and rearranges the elements to form a heap.

In Object Oriented languages, BuildHeap is essentially the *Constructor* of class Heap.
The procedure BuildHeap works by using a method called Heapify(*node*).

Heapify

The HEAPIFY(*node*) procedure:

- If *node* violates the heap property:
  1. Swap value of *node* with the largest of its two children.
  2. Call HEAPIFY on the child replaced.
- Else, do nothing and return.

Heapify

The Heapify(*node*) procedure:

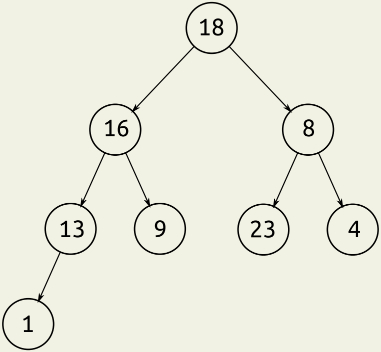
- If *node* violates the heap property:
  1. Swap value of *node* with the largest of its two children.
  2. Call Heapify on the child replaced.
- Else, do nothing and return.

Note:

- The Heapify procedure assumes that both the subtrees under *node* are already heaps.
- It merely resolves the possible conflict between the value at *node* and its children and recurses.
- Can take $O(\log n)$ time.

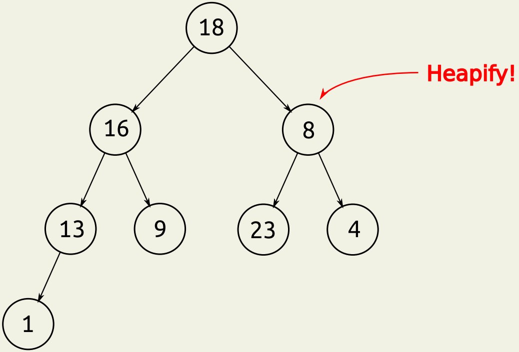
Heapify

Example:



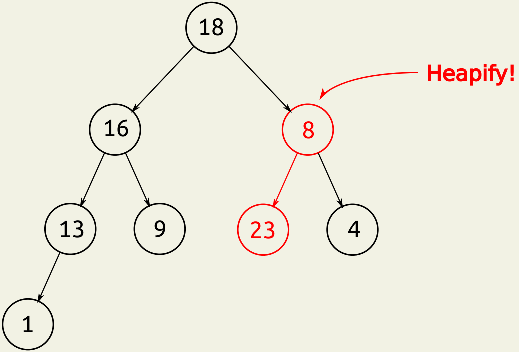
Heapify

Example:



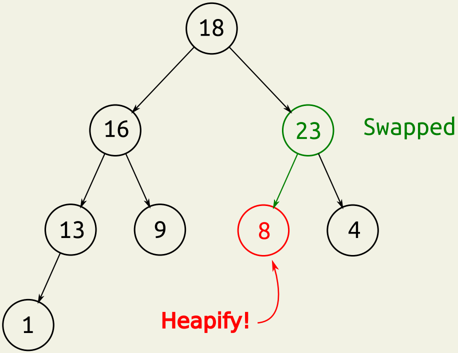
Heapify

Example:



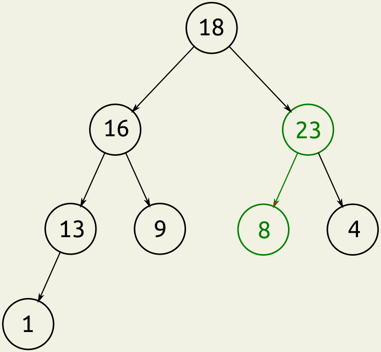
Heapify

Example:



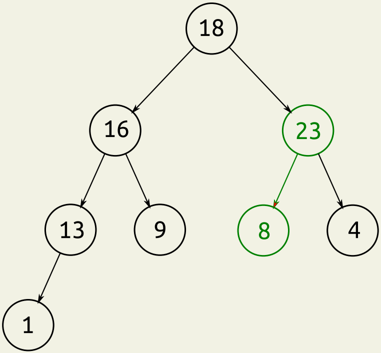
Heapify

Example:



Heapify

Note that Heapify only resolves conflicts downwards.



Exercises

Write the following procedures:

- INSERT(*val*):
  - Insert new value as the last element in the array.
  - Repeatedly Heapify *upwards* from the new element.
  - This can also be viewed as "sifting".
- EXTRACTMAX(): Swap positions of root with last leaf. Heapfiy at new root.

Exercises

Write the following procedures:

- INSERT(*val*):
  - Insert new value as the last element in the array.
  - Repeatedly Heapify *upwards* from the new element.
  - This can also be viewed as "sifting".
- EXTRACTMAX(): Swap positions of root with last leaf. Heapfiy at new root.
- Running time?

Building a Heap

Two ways:

- William's method: Take each element and use INSERT procedure.
- Floyd's method: Take all elements in an arbitrary array. Heapify repeatedly.

Building a Heap

Two ways:

- William's method: Take each element and use Insert procedure. Takes $O(n \log n)$ time.
- Floyd's method: Take all elements in an arbitrary array. Heapify repeatedly.

Building a Heap

The procedure BUILDHEAP(A) by Floyd is the following:

- For i from n to 1:
  - HEAPIFY(i)

Building a Heap

The procedure $\textsc{BuildHeap}(A)$ by Floyd is the following:

- For i from n to 1:
  - $\textsc{Heapify}(i)$

Note: Indices $n/2$ to n form leaves of the heap.
The leaves are already heaps (trivially).
Hence it suffices to run the above loop from $n/2$ to 1.

Visualization

On the board

Analysis of Floyd's Method

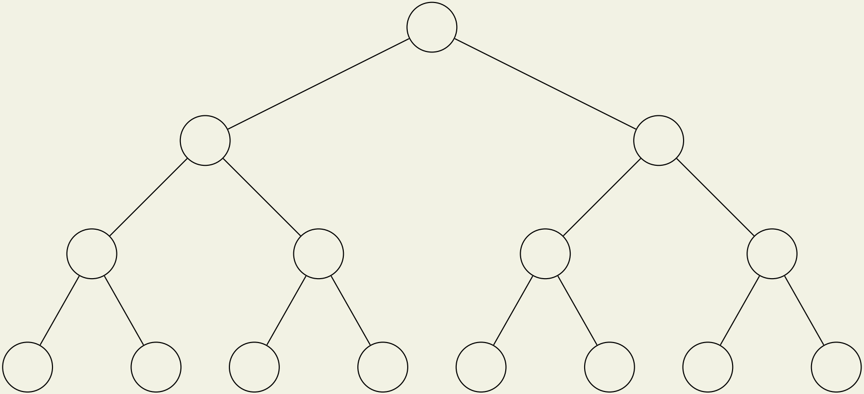
- We need to do $n/2$ Heapify operations
- Each Heapify can take $O(\log n)$ time
- So total time is $O(n \log n)$

Analysis of Floyd's Method

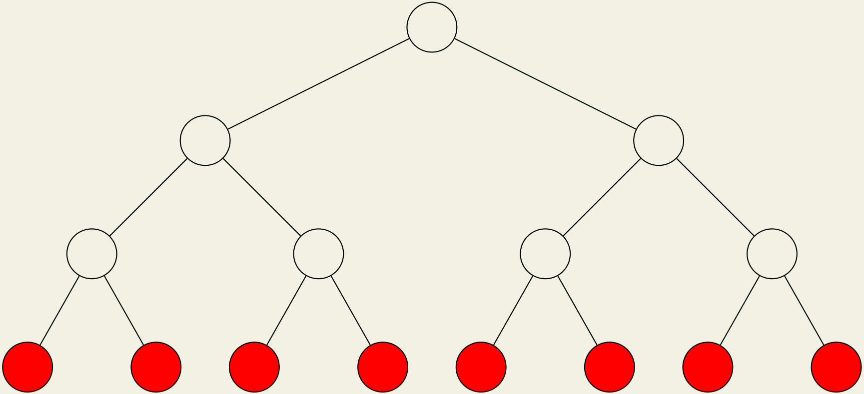
- We need to do $n/2$ Heapify operations
- Each Heapify can take $O(\log n)$ time
- So total time is $O(n \log n)$

- But most of the Heapify operations are small
- We have $n/2$ nodes at height 1, $n/4$ nodes at height 2 and so on
- It can be shown that BuildHeap(A) takes only $O(n)$ time

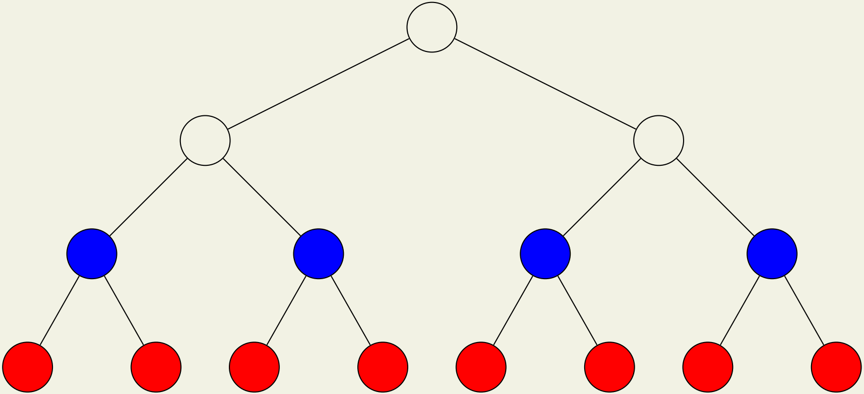
Analysis of Floyd’s Method



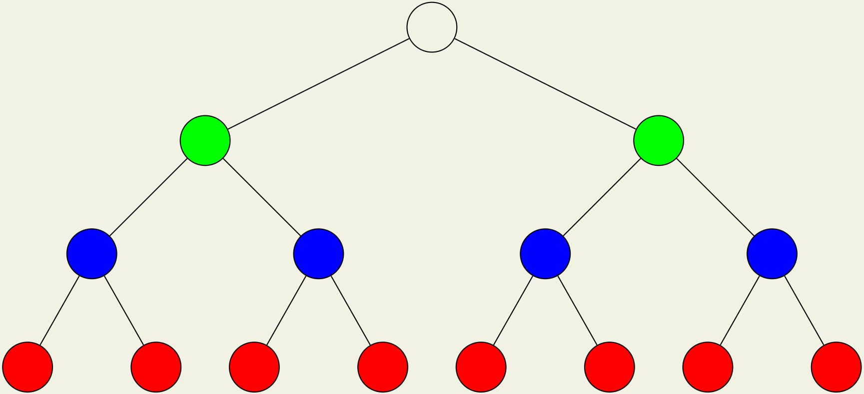
Analysis of Floyd's Method



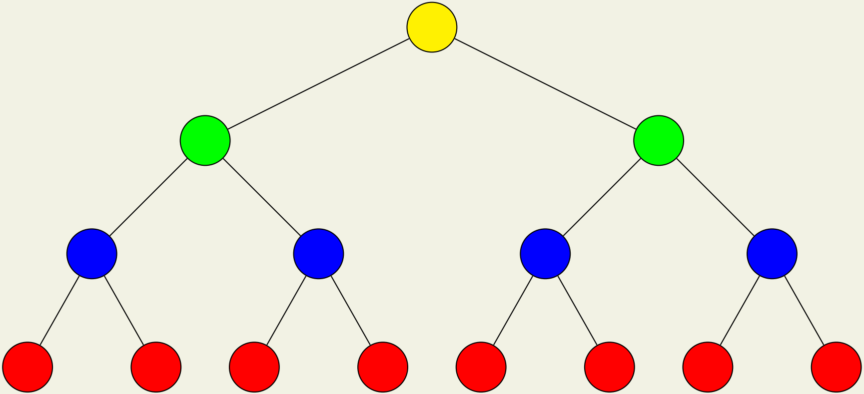
Analysis of Floyd's Method



Analysis of Floyd's Method



Analysis of Floyd's Method



Analysis of Floyd's Method

- We have $n/2$ values needing at most 1 swap
- We have $n/4$ values needing at most 2 swaps
- We have $n/8$ values needing at most 3 swaps, and so on.

Analysis of Floyd's Method

- We have $n/2$ values needing at most 1 swap
- We have $n/4$ values needing at most 2 swaps
- We have $n/8$ values needing at most 3 swaps, and so on.

$$
\begin{aligned}
\text{Total no. of swaps} &= \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots \\
&= n \left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \dots \right) \\
&= n \sum_{i=1}^{\log n} \frac{i}{2^i} \leq n \sum_{i=1}^{\infty} \frac{i}{2^i}
\end{aligned}
$$

Analysis of Floyd's Method

- The number of swaps is at most $n \sum_{i=1}^{\infty} \frac{i}{2^i}$
- This can be shown to be at most $2n$
- Thus BuildHeap is performed in $O(n)$ time

Heap Sort

- Given an array A:
- Run BuildHeap(A)
- Repeatedly do ExtractMax()
- What is the total time?

Graphs

Graph

(directed)

A (directed) graph G is a two tuple (V, E) where:

- V is a set of elements called “vertices”.
- $E \subseteq V \times V$ is a binary relation. Elements in E are called “edges”.

Note: There are several definitions and variants of graphs.
Graphs are a way to study the relationships among a set of elements.

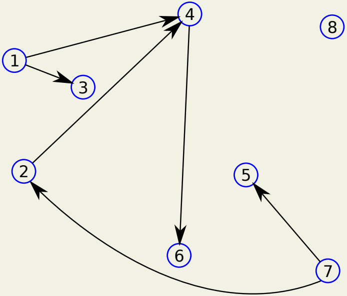
Example - directed graph

Consider:

$$\begin{aligned} V =& \{1, 2, 3, 4, 5, 6, 7, 8\} \\ E =& \{\ (1, 3), (1, 4), \\ & (2, 4), (4, 6), \\ & (7, 2), (7, 5)\ \} \end{aligned}$$

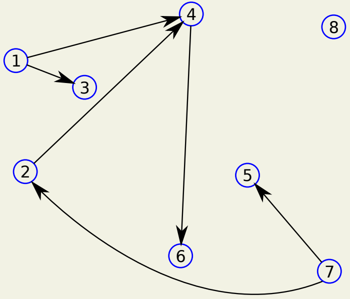
Example - directed graph

$$
\begin{aligned}
V =& \{1, 2, 3, 4, 5, 6, 7, 8\} \\
E =& \{\ (1, 3), (1, 4), \\
& (2, 4), (4, 6), \\
& (7, 2), (7, 5)\ \}
\end{aligned}
$$



Example - directed graph

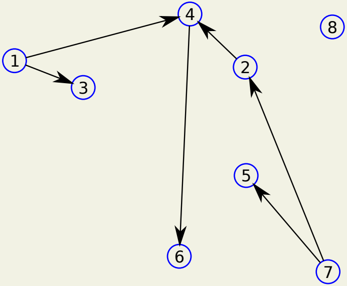
$$
\begin{aligned}
V =& \{1, 2, 3, 4, 5, 6, 7, 8\} \\
E =& \{\ (1, 3), (1, 4), \\
& (2, 4), (4, 6), \\
& (7, 2), (7, 5)\ \}
\end{aligned}
$$



The vertices can be drawn anywhere! The edges are what matter.

Example - directed graph

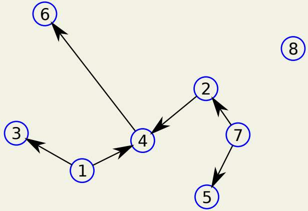
$$
\begin{aligned}
V =& \{1, 2, 3, 4, 5, 6, 7, 8\} \\
E =& \{ (1, 3), (1, 4), \\
& (2, 4), (4, 6), \\
& (7, 2), (7, 5) \}
\end{aligned}
$$



The vertices can be drawn anywhere! The edges are what matter.

Example - directed graph

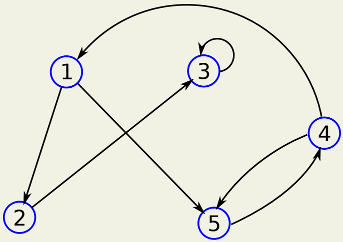
$$
\begin{aligned}
V =& \{1, 2, 3, 4, 5, 6, 7, 8\} \\
E =& \{ (1, 3), (1, 4), \\
& (2, 4), (4, 6), \\
& (7, 2), (7, 5) \ \}
\end{aligned}
$$



The vertices can be drawn anywhere! The edges are what matter.

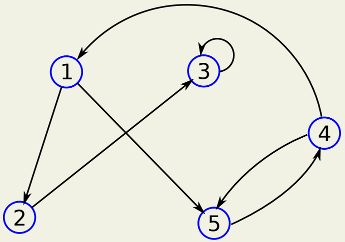
Example - directed graph

$$
\begin{aligned}
V =& \{1, 2, 3, 4, 5\} \\
E =& \{\ (1, 2), (1, 5), \\
& (2, 3), (3, 3), \\
& (4, 1), (4, 5), \\
& (5, 4)\ \}
\end{aligned}
$$



Example - directed graph

$$
\begin{aligned}
V =& \{1, 2, 3, 4, 5\} \\
E =& \{\ (1, 2), (1, 5), \\
& (2, 3), (3, 3), \\
& (4, 1), (4, 5), \\
& (5, 4)\ \}
\end{aligned}
$$



Terminology:

- A vertex v is a *neighbour* or *adjacent* to u if $(u, v) \in E$.
- The neighbourhood $\mathcal{N}(u)$ of a vertex u is the set of all neighbours of u.

Graph

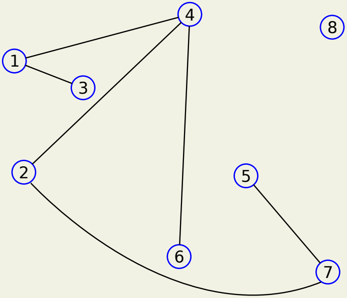
(undirected)

A (undirected) graph G is a two tuple (V, E) where:

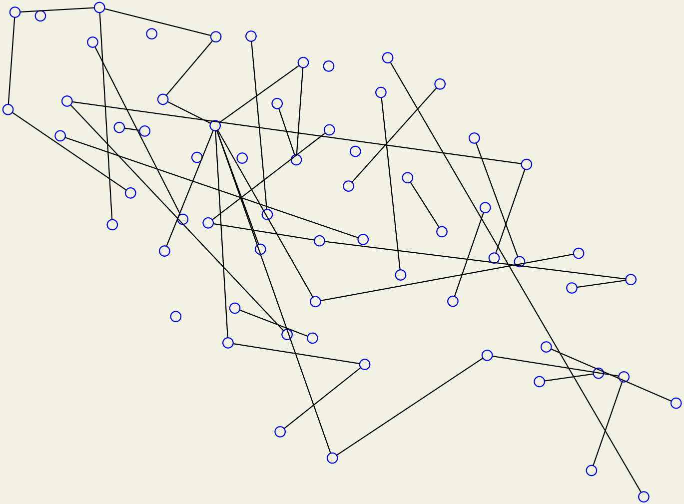
- V is a set of elements called "vertices".
- E is a set of (unordered) pairs of vertices from V.

Example - undirected graph

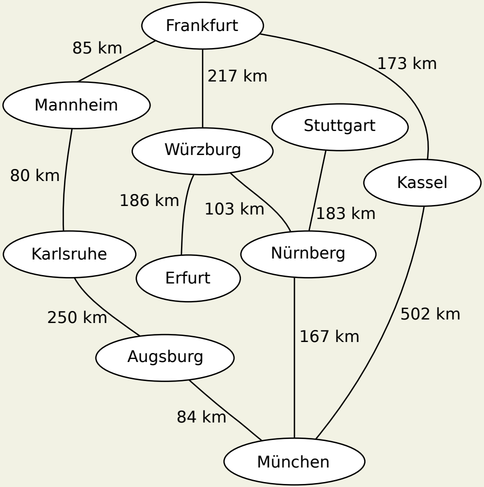
$$
\begin{aligned}
V =& \{1, 2, 3, 4, 5, 6, 7, 8\} \\
E =& \{\ \{1, 3\}, \{1, 4\}, \\
& \{2, 4\}, \{4, 6\}, \\
& \{7, 2\}, \{7, 5\}\ \}
\end{aligned}
$$



Example - undirected graph



Example - undirected graph



(source: wikipedia.org)

Weighted graphs have a *weight* assigned to each edges using a weight function.

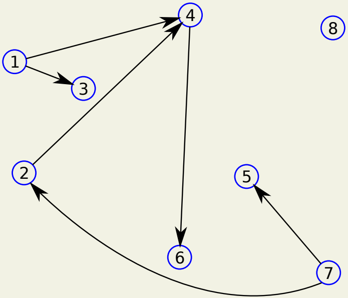
Data structure

Two standard data structures to represent graphs:

- Adjacency matrix
- Adjacency list

Adjacency Matrix

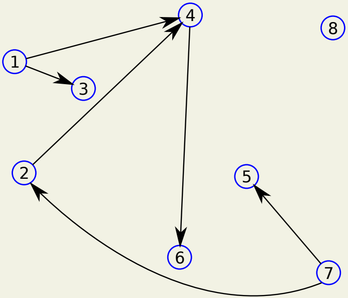
$$
\begin{array}{c|c}
A & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} \\
\hline
\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix} &
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$



$$A[u, v] = 1 \iff (u, v) \in E$$

Adjacency Matrix

$$\begin{array}{c|c} A & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} \\ \hline \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix} & \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

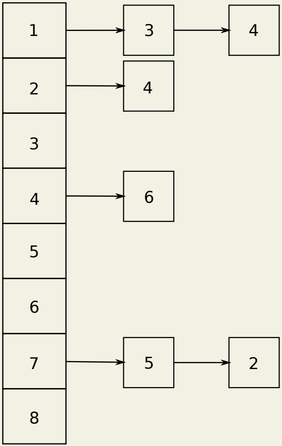


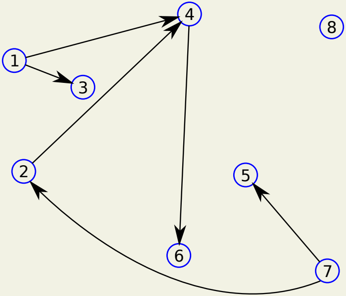
$$A[u, v] = 1 \iff (u, v) \in E$$

For an undirected graph:

- $u, v \in E \iff A[u, v] = A[v, u] = 1$
- The adjacency matrix for an undirected graph is a symmetric matrix

Adjacency Lists





Graph algorithms

Some natural question to ask about an input graph:

- Starting from a vertex s, what vertices are reachable?
- What is the shortest path from a vertex s to a vertex v?

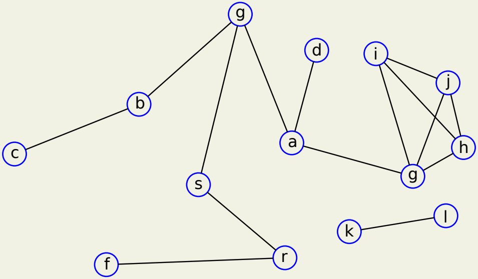
Algorithms that work on an input graph are called graph algorithms. One of the fundamental graph algorithms is the Breadth-first Search.

Breadth-first Search

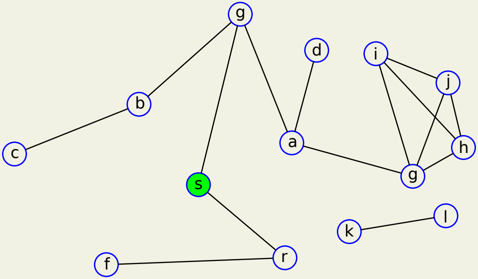
The idea is to explore the graph “radially outward” from the source.

In each step, we expand our exploration by visiting the neighborhood of all explored vertices.

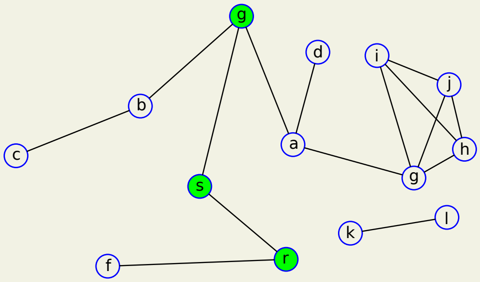
Breadth-first Search (idea)



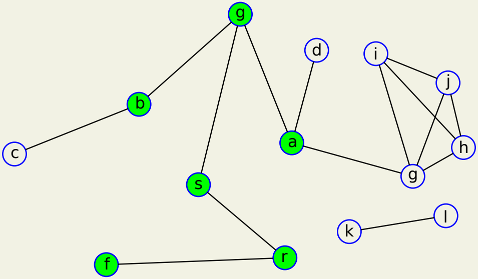
Breadth-first Search (idea)



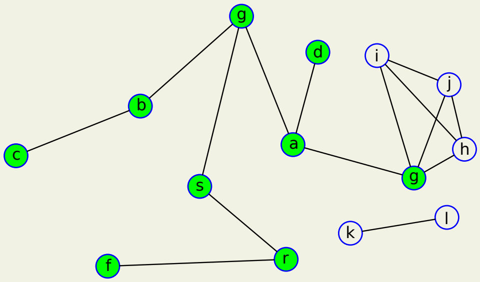
Breadth-first Search (idea)



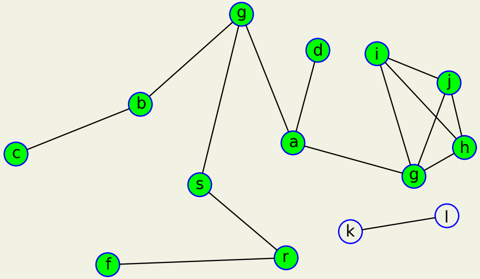
Breadth-first Search (idea)



Breadth-first Search (idea)



Breadth-first Search (idea)



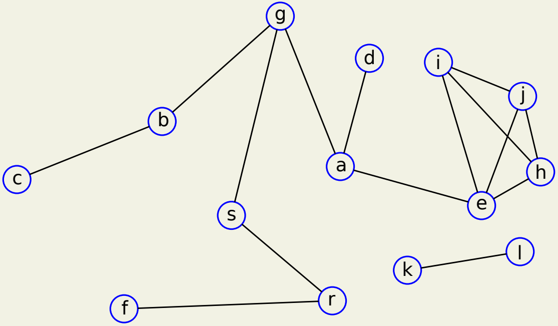
Breadth-first Search (idea)

Important:

- Do not visit an already explored vertex.
- Keep track of distance from source.
- Terminate algorithm when no new vertices can be explored.

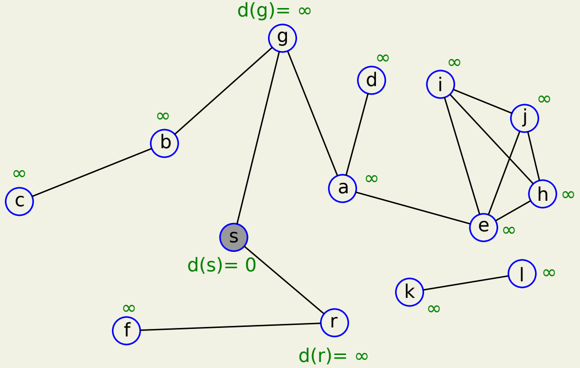
Breadth-first Search

Queue: $\emptyset$



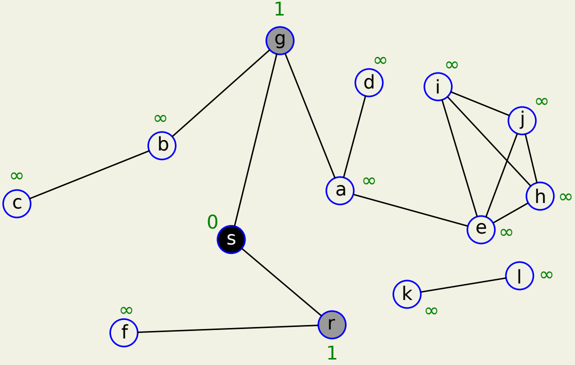
Breadth-first Search

Dequeued vertex: $\boxed{}$ Queue: $\boxed{s}$



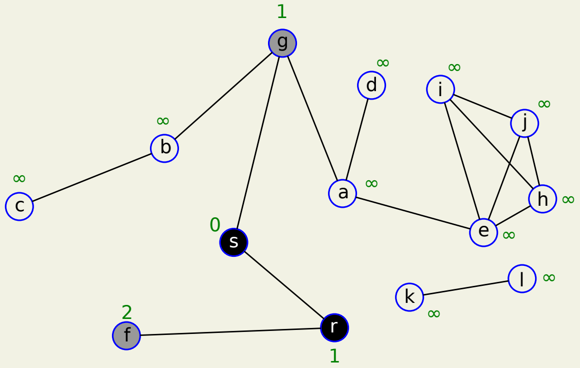
Breadth-first Search

Dequeued vertex: $\boxed{s}$ Queue: $\boxed{r \mid g}$



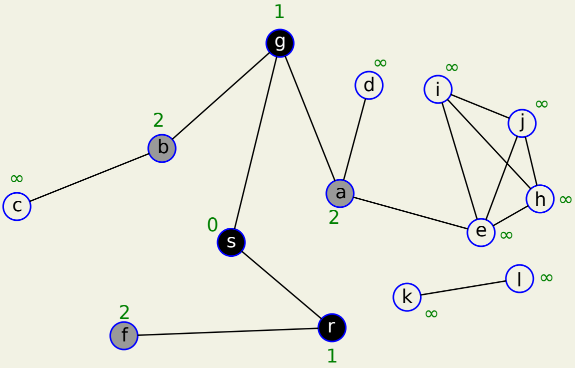
Breadth-first Search

Dequeued vertex: $\boxed{r}$ Queue: $\boxed{g \mid f}$



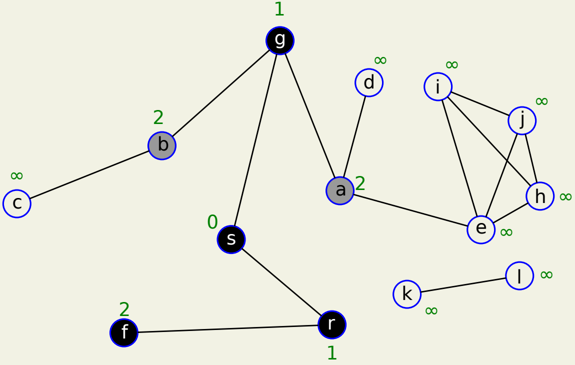
Breadth-first Search

Dequeued vertex: | g | Queue: | f | a | b |



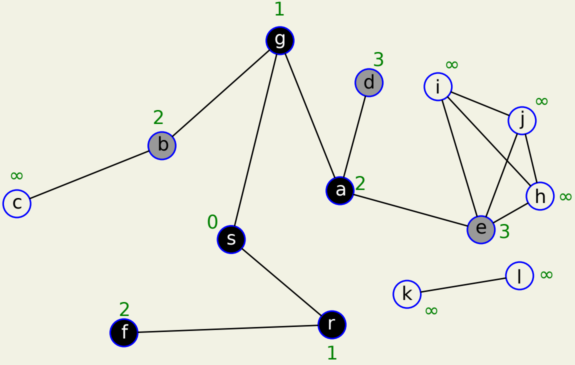
Breadth-first Search

Dequeued vertex: $\boxed{f}$ Queue: $\boxed{a \mid b}$



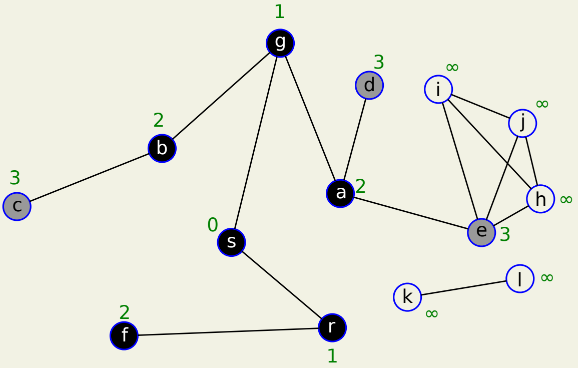
Breadth-first Search

Dequeued vertex: | *a* | Queue: | *b* | *e* | *d* |



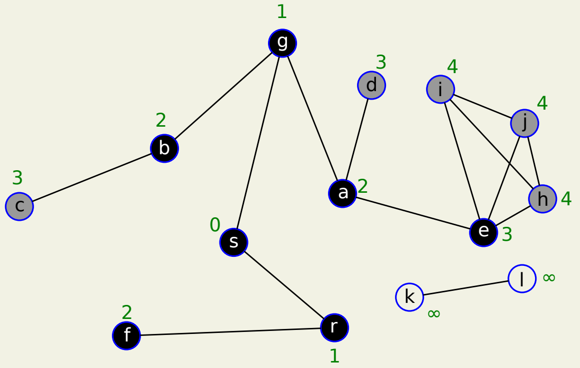
Breadth-first Search

Dequeued vertex: $\boxed{b}$ Queue: $\boxed{e \mid d \mid c}$



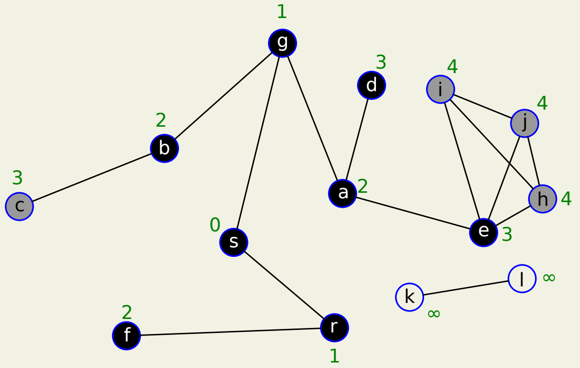
Breadth-first Search

Dequeued vertex: $\boxed{e}$ Queue: | d | c | j | h | i |



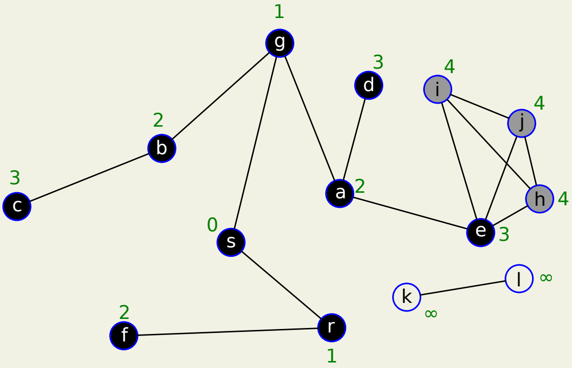
Breadth-first Search

Dequeued vertex: | *d* | Queue: | *c* | *j* | *h* | *i* |



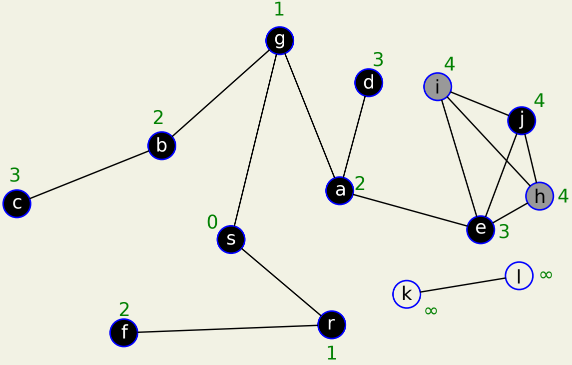
Breadth-first Search

Dequeued vertex: $\boxed{c}$ Queue: $\boxed{j \mid h \mid i}$



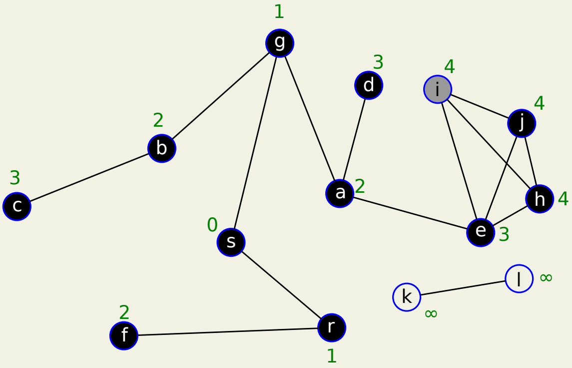
Breadth-first Search

Dequeued vertex: $\boxed{j}$ Queue: $\boxed{h \mid i}$



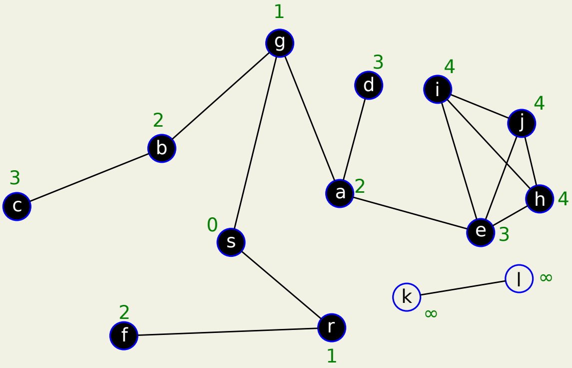
Breadth-first Search

Dequeued vertex: $\boxed{h}$ Queue: $\boxed{i}$



Breadth-first Search

Dequeued vertex: $\boxed{i}$ Queue: $\emptyset$



Algorithm 1 Breadth-first Search from vertex s

```
Color all vertices WHITE.
For all u ∈ V, d[u] ← ∞, π[u] ← NIL.
d[s] ← 0.
Initialize queue Q ← ∅.
ENQUEUE(Q, s)
while Q ≠ ∅ do
    u ← DEQUEUE(Q)
    for each v ∈ 𝒩(u) do
        if color(v) =WHITE then
            color[v] ← GRAY
            d[v] ← d[u] + 1
            π[v] ← u
            ENQUEUE(Q, v)
        end if
    end for
    color[u] ← BLACK.
end while
```

Correctness of BFS

Notation: Let $\delta(s, v)$ denote the minimum number of edges on a path from s to v.

Theorem

Let $G = (V, E)$ be a graph. When BFS is run on G from vertex $s \in V$:

1. Every vertex that is reachable from s gets discovered.
2. On termination, $d[v] = \delta(s, v)$.

Show (1) is an exercise.