CS 261 Data Structures

Spring 2025

Priority Queue, Graph

Tianshuo Zhou

May 27 & 29, 2025

Roadmap

- priority queue
  - concept of priority
  - binary heap
  - heap sort
  - assignment: heap implementation
- graph
  - concept, types, representations
  - BFS
  - DFS
  - topological sort

Priority Queue

- examples of priority?

Priority Queue

- examples of priority?
  - emergency room
  - process niceness
  - email responses
  - queue

Priority Queue

Priority Queue

- priority queue is an ADT
  - a data structure is an instance of ADT

Priority Queue

- priority queue is an ADT
  - a data structure is an instance of ADT
- operations of priority queue?

Priority Queue

- priority queue is an ADT
  - a data structure is an instance of ADT
- operations of priority queue
  - push(P, x) / insert(P, x)
  - pop(P) / dequeue(P)
  - build heap / heapify
  - peek(P) / max(P)

Priority Queue

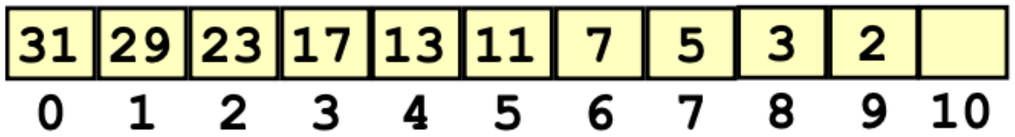
- priority queue is an ADT
  - a data structure is an instance of ADT
- operations of priority queue
  - push(P, x) / insert(P, x)
  - pop(P) / dequeue(P)
  - build heap / heapify
  - peek(P) / max(P)
- implementation?

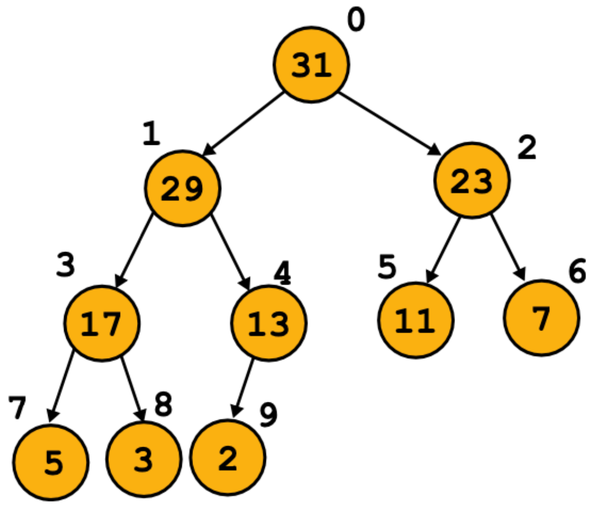
Priority Queue Implementations

	sorted array	reversely sorted array	sorted doubly linked list	balanced BST	binary heap
push	O(n)	O(n)	O(n)	**O(log n)**	**O(log n)**
pop-max	**O(1)**	O(n)	**O(1)**	O(log n)	O(log n)
peek-max	**O(1)**	**O(1)**	**O(1)**	O(log n)	**O(1)**
build	O(n log n)	O(n log n)	O(n log n)	O(n log n)	**O(n)**

Binary Heap

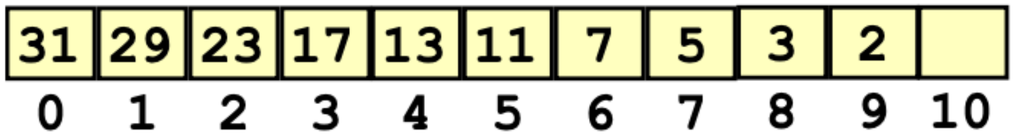
- structure or topology
  - logically: complete tree
  - physically: linear array

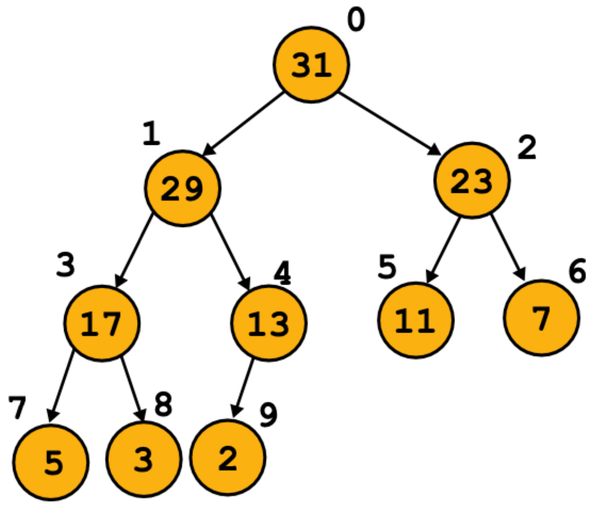




Binary Heap

- structure or topology
  - logically: complete tree
  - physically: linear array



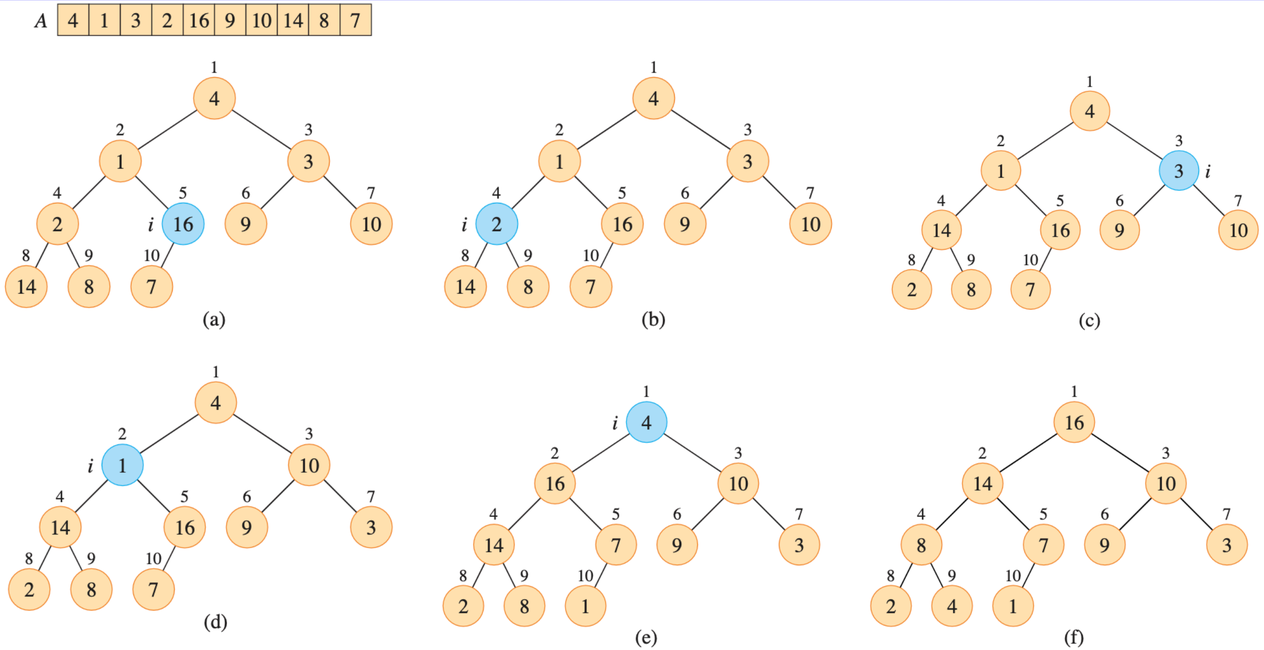


- complexity
  - push $\rightarrow$ bubble up: $O(h) = O(\log n)$
  - pop $\rightarrow$ bubble down: $O(h) = O(\log n)$
  - build_heap or heapify: $O(n)$

Binary Heap

- build a heap or heapify

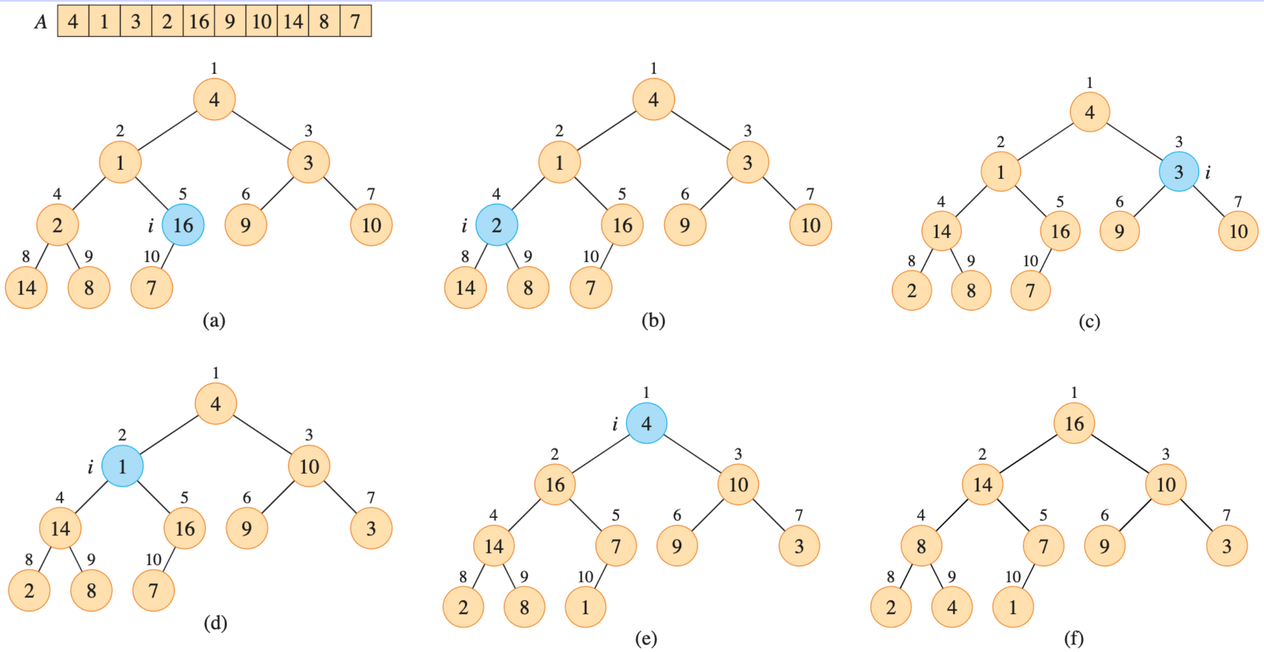
function BUILD-HEAP(A)
  $n \leftarrow A$.size
  for $i \leftarrow \lfloor n/2 \rfloor - 1$ **downto** 0 **do**
    BUBBLE-DOWN(A, i)



Binary Heap

- build a heap or heapify

function BUILD-HEAP(A)
  $n \leftarrow A.\text{size}$
  for $i \leftarrow \lfloor n/2 \rfloor - 1$ **downto** 0 **do**
    BUBBLE-DOWN(A, i)

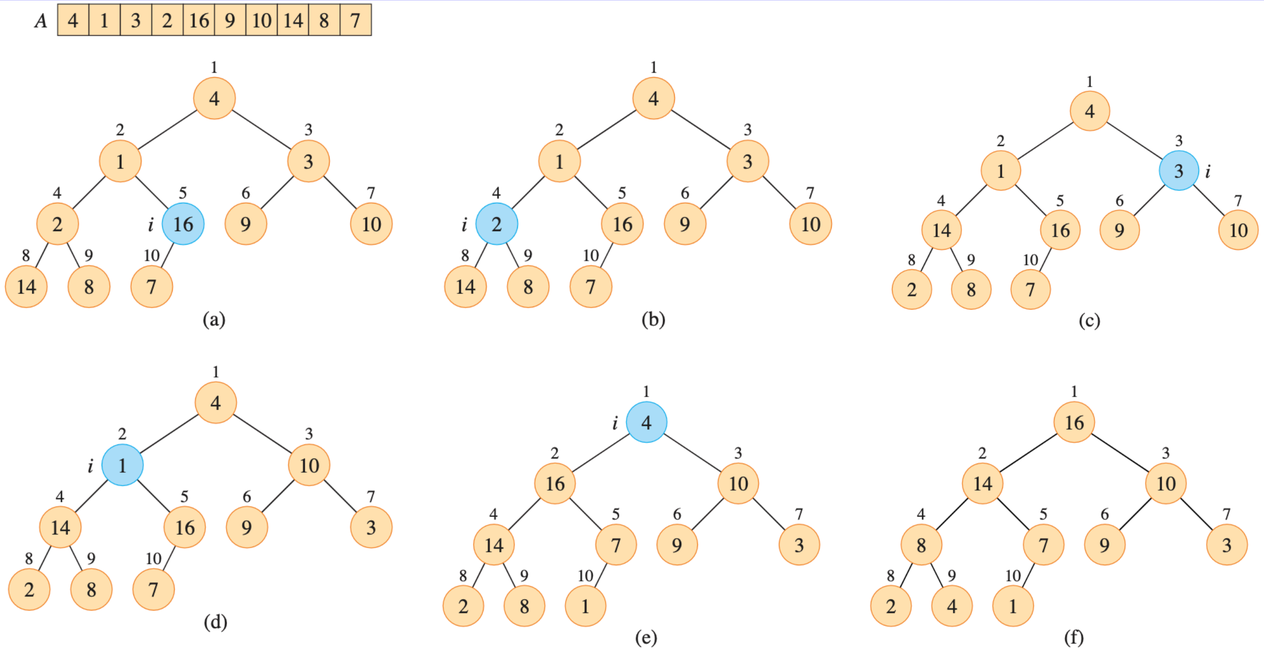


- complexity analysis
  - count how many bubble-down steps
  - $\frac{n}{2} \times 0 + \frac{n}{4} \times 1 + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 + \cdots$
    - arithmetico-geometric sequence

Binary Heap

- build a heap or heapify

function BUILD-HEAP(A)
  $n \leftarrow A.\text{size}$
  for $i \leftarrow \lfloor n/2 \rfloor - 1$ **downto** 0 **do**
    BUBBLE-DOWN(A, i)



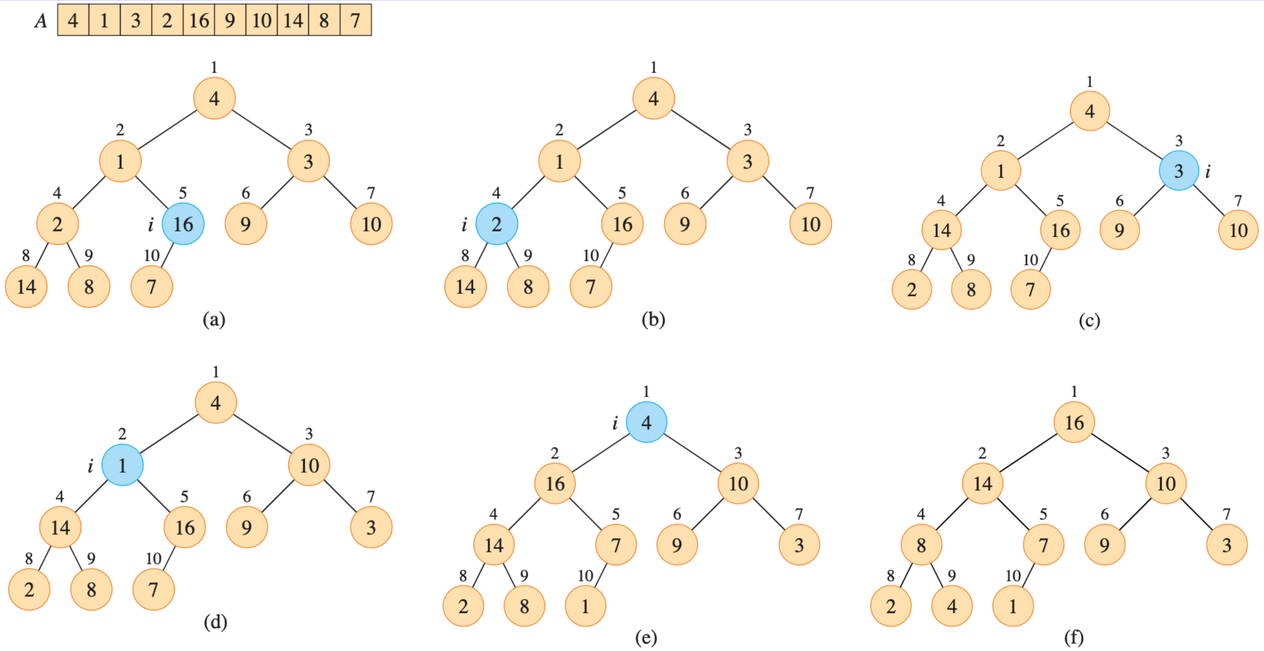
- complexity analysis
  - count how many bubble-down steps
  - $\frac{n}{2} \times 0 + \frac{n}{4} \times 1 + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 + \cdots$
    - arithmetico-geometric sequence

$$
\begin{aligned}
&\frac{n}{4} \times 1 \quad + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 + \frac{n}{32} \times 4 + \cdots \\
=&\frac{n}{4} \times 1 \quad + \frac{n}{8} \times 1 + \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&\quad + \frac{n}{8} \times 1 + \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&\quad\quad + \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&\quad\quad\quad + \frac{n}{32} \times 1 + \cdots \\
= &\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \cdots = n
\end{aligned}
$$

Binary Heap

- build a heap or heapify

function BUILD-HEAP(A)
  $n \leftarrow A.\text{size}$
  for $i \leftarrow \lfloor n/2 \rfloor - 1$ **downto** 0 **do**
    BUBBLE-DOWN(A, i)



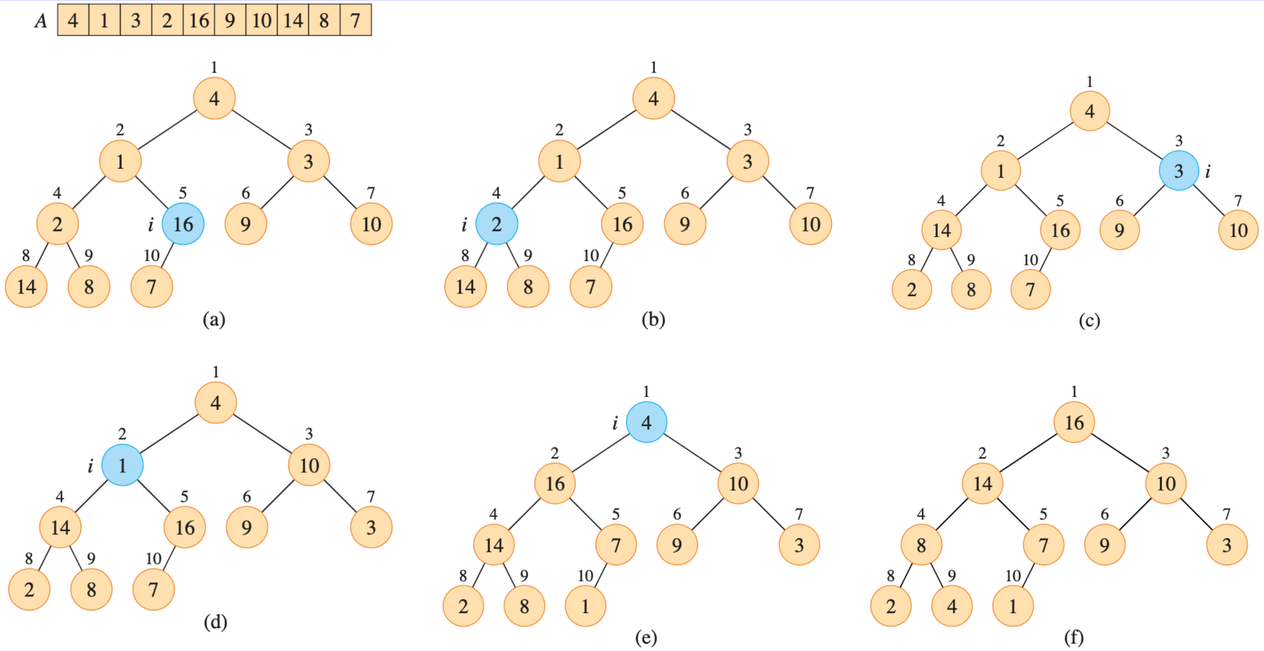
- complexity analysis
  - count how many bubble-down steps
  - $\frac{n}{2} \times 0 + \frac{n}{4} \times 1 + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 + \cdots$
    - arithmetico-geometric sequence

$$
\begin{aligned}
&\frac{n}{4} \times 1 \quad + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 + \frac{n}{32} \times 4 + \cdots \\
=&\frac{n}{4} \times 1 \quad + \frac{n}{8} \times 1 + \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&\quad + \frac{n}{8} \times 1 + \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&\quad\quad + \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&\quad\quad\quad + \frac{n}{32} \times 1 + \cdots \\
= &\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \cdots = n
\end{aligned}
$$

Binary Heap

- build a heap or heapify

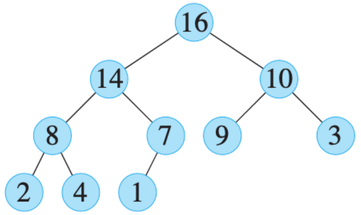
function BUILD-HEAP(A)
  $n \leftarrow A.\text{size}$
  for $i \leftarrow \lfloor n/2 \rfloor - 1$ **downto** 0 **do**
    BUBBLE-DOWN(A, i)



- complexity analysis
  - count how many bubble-down steps
  - $\frac{n}{2} \times 0 + \frac{n}{4} \times 1 + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 + \cdots$
    - arithmetico-geometric sequence

$$\begin{aligned}
&\frac{n}{4} \times 1 + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 + \frac{n}{32} \times 4 + \cdots \\
=&\frac{n}{4} \times 1 + \frac{n}{8} \times 1 + \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&+ \frac{n}{8} \times 1 + \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&+ \frac{n}{16} \times 1 + \frac{n}{32} \times 1 + \cdots \\
&+ \frac{n}{32} \times 1 + \cdots \\
= &\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \cdots = n
\end{aligned}$$

HeapSort

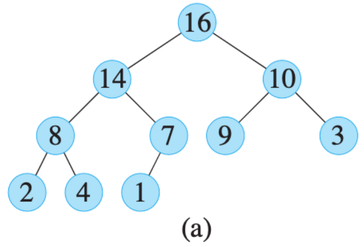


(a)

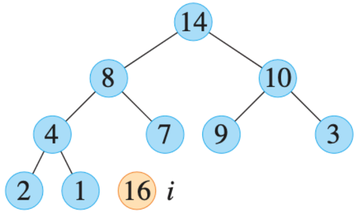
HeapSort

- two-stage algorithm

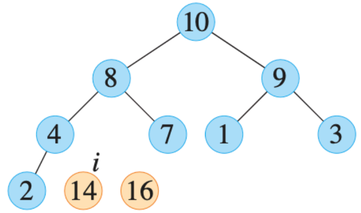
function HEAPSORT(A)
    BUILD-HEAP(A)
    $n \leftarrow A$.size
    for $i \leftarrow n-1$ **downto** 1 **do**
        SWAP($A[i], A[0]$)
        BUBBLE-DOWN($A, 0$)



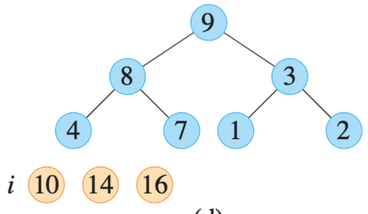
(a)



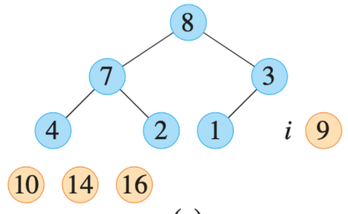
(b)



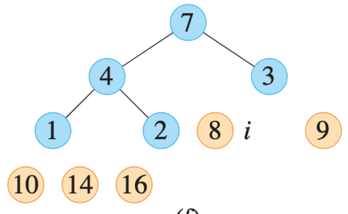
(c)



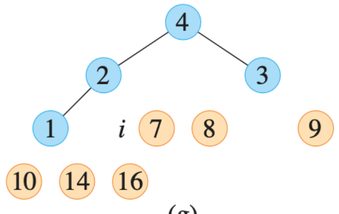
(d)



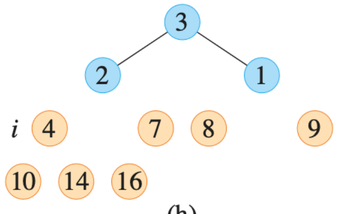
(e)



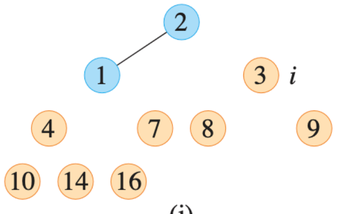
(f)



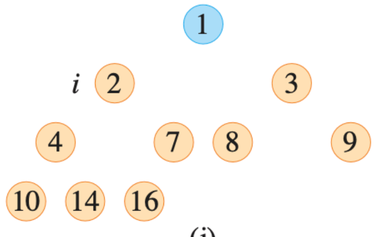
(g)



(h)



(i)



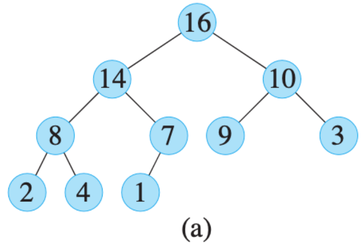
(j)

(k)

HeapSort

- two-stage algorithm

function HEAPSORT(A)
    BUILD-HEAP(A)
    $n \leftarrow A$.size
    for $i \leftarrow n - 1$ **downto** 1 **do**
        SWAP($A[i], A[0]$)
        BUBBLE-DOWN($A, 0$)



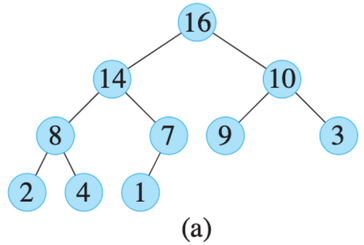
(a)

- complexity analysis
  - build heap: $O(n)$
  - $n - 1$ pops (bubble-down): $O(n \log(n))$
  - total cost: $O(n \log(n))$

HeapSort

- two-stage algorithm

function HEAPSORT(A)
    BUILD-HEAP(A)
    $n \leftarrow A$.size
    for $i \leftarrow n-1$ **downto** 1 **do**
        SWAP($A[i], A[0]$)
        BUBBLE-DOWN($A, 0$)



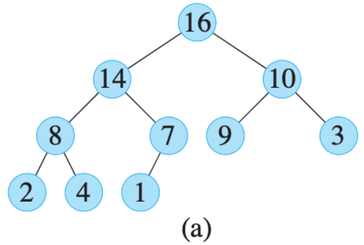
(a)

- complexity analysis
  - build heap: $O(n)$
  - $n-1$ pops (bubble-down): $O(n \log(n))$
  - total cost: $O(n \log(n))$
    - is $n \log(n)$ tight?

HeapSort

- two-stage algorithm

function HEAPSORT(A)
    BUILD-HEAP(A)
    $n \leftarrow A$.size
    for $i \leftarrow n-1$ **downto** 1 **do**
        SWAP($A[i], A[0]$)
        BUBBLE-DOWN($A, 0$)



(a)

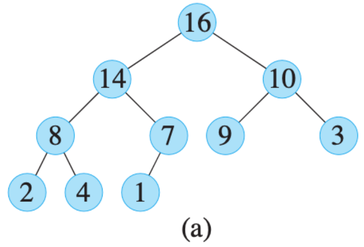
- asymptotic notations
  - upper bound: $T(n) = O(f(n))$
    - $T(n) \leq c \cdot f(n)$
  - lower bound: $T(n) = \Omega(f(n))$
    - $T(n) \geq c \cdot f(n)$
  - tight bounds: $T(n) = \Theta(f(n))$
    - $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$

- complexity analysis
  - build heap: $O(n)$
  - $n-1$ pops (bubble-down): $O(n \log(n))$
  - total cost: $O(n \log(n))$
    - is $n \log(n)$ tight?

HeapSort

- two-stage algorithm

function HEAPSORT(A)
  BUILD-HEAP(A)
  $n \leftarrow A$.size
  for $i \leftarrow n-1$ **downto** 1 **do**
    SWAP($A[i], A[0]$)
    BUBBLE-DOWN($A, 0$)



(a)

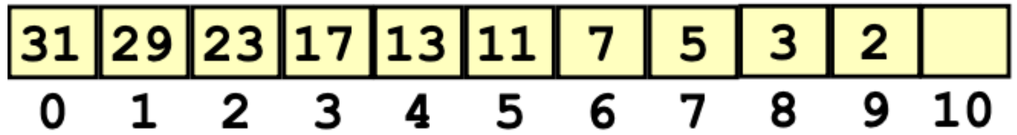
- asymptotic notations
  - upper bound: $T(n) = O(f(n))$
    - $T(n) \leq c \cdot f(n)$
  - lower bound: $T(n) = \Omega(f(n))$
    - $T(n) \geq c \cdot f(n)$
  - tight bounds: $T(n) = \Theta(f(n))$
    - $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$

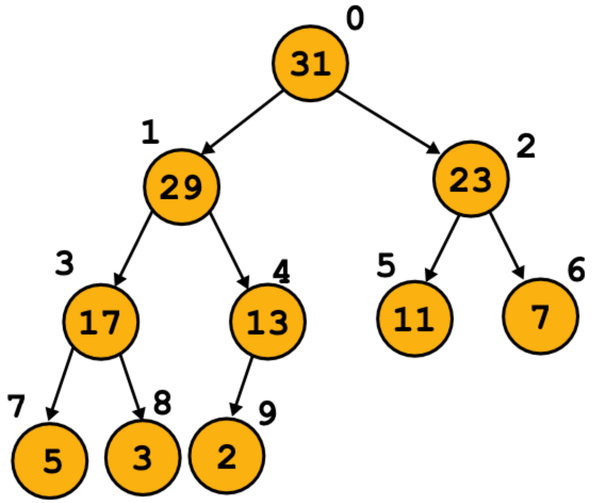
- complexity analysis
  - build heap: $O(n)$
  - $n-1$ pops (bubble-down): $O(n \log(n))$
  - total cost: $O(n \log(n))$
    - is $n \log(n)$ tight? cost is at least $\Theta(\frac{n}{2} \times \log \frac{n}{2}) = \Theta(n \log n)$
    - yes! first half of the nodes have a height of $\log(n/2)$ when popped

Assignment 6

- objectives
  - heap data: dynamic arrays
  - heap operation: insert, percolate_up, percolate_down

- structure or topology
  - logically: complete tree
  - physically: linear array



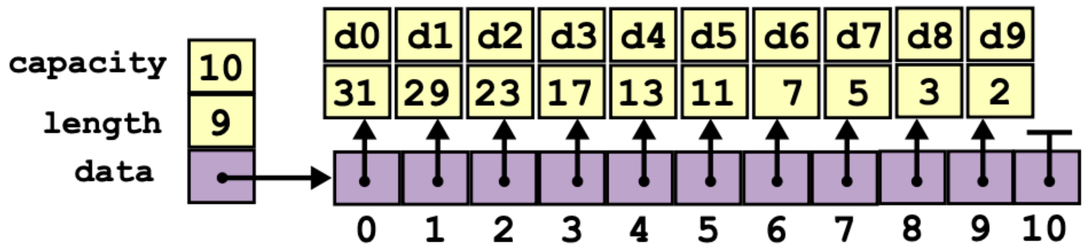


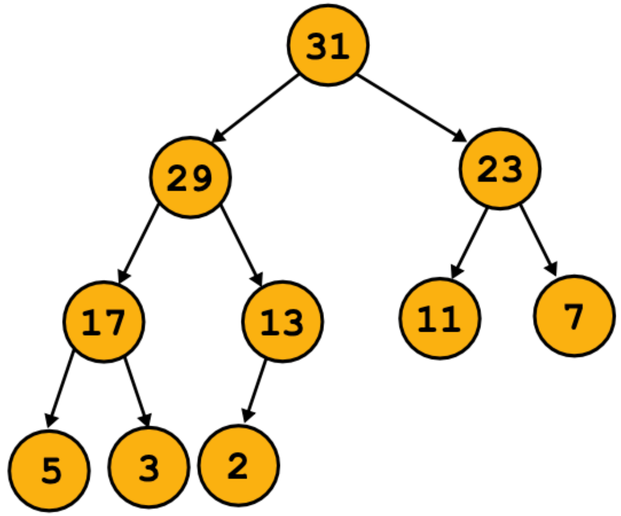
Assignment 6

- objectives
  - heap data: dynamic arrays of key(priority) and value
  - heap operation: insert, percolate_up, percolate_down

```c
struct element {
  void* data;
  int priority;
};
```

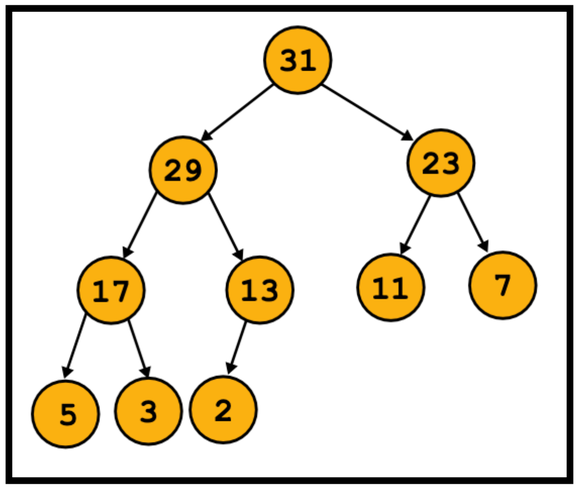
```c
struct dynarray {
  void** data;
  int length;
  int capacity;
};
```





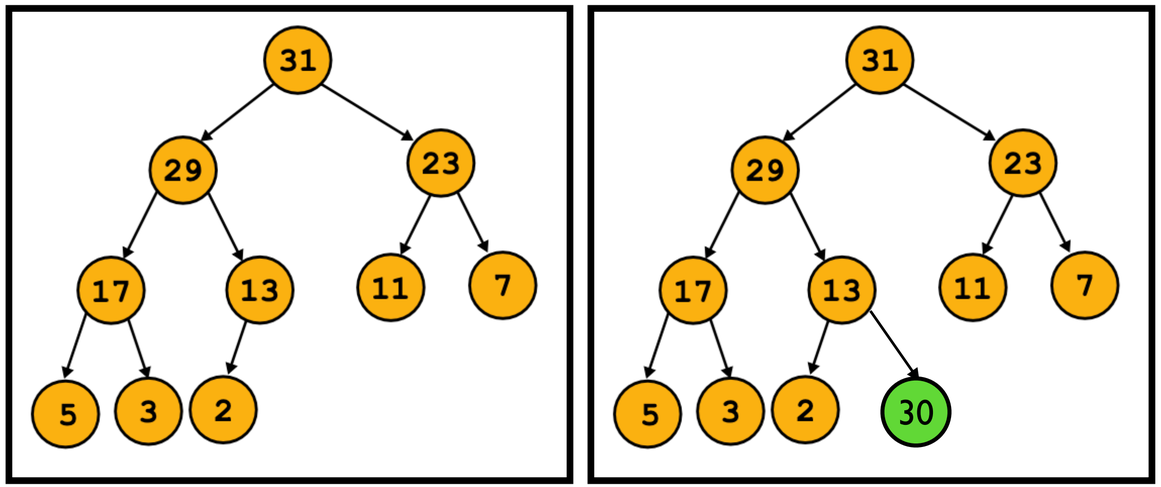
Assignment 6

- push / insert
  - append new element
  - percolate / bubble up



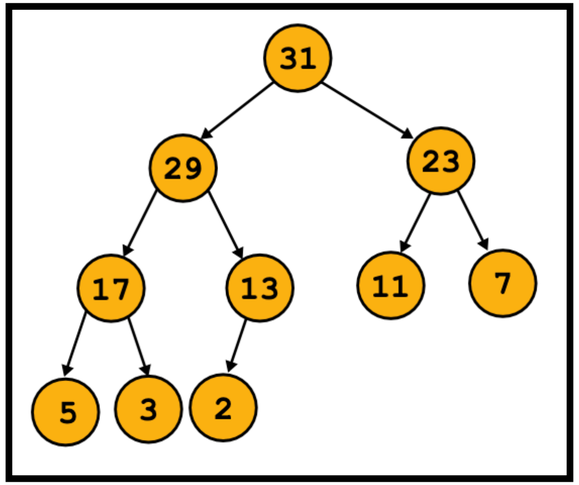
Assignment 6

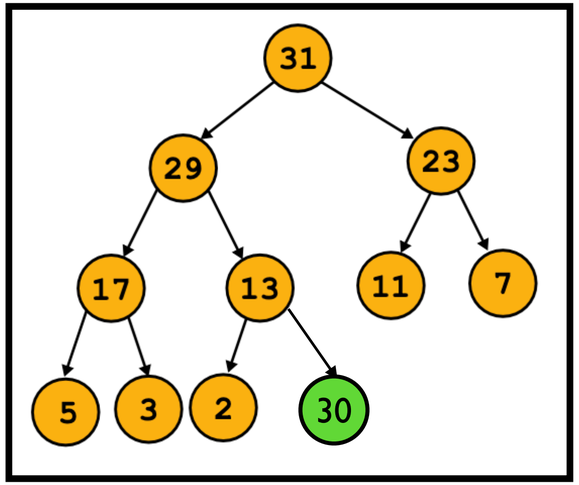
- push / insert
  - append new element
  - percolate / bubble up

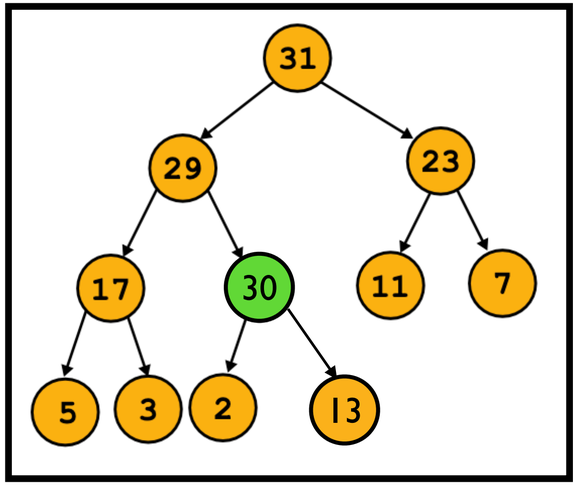


Assignment 6

- push / insert
  - append new element
  - percolate / bubble up

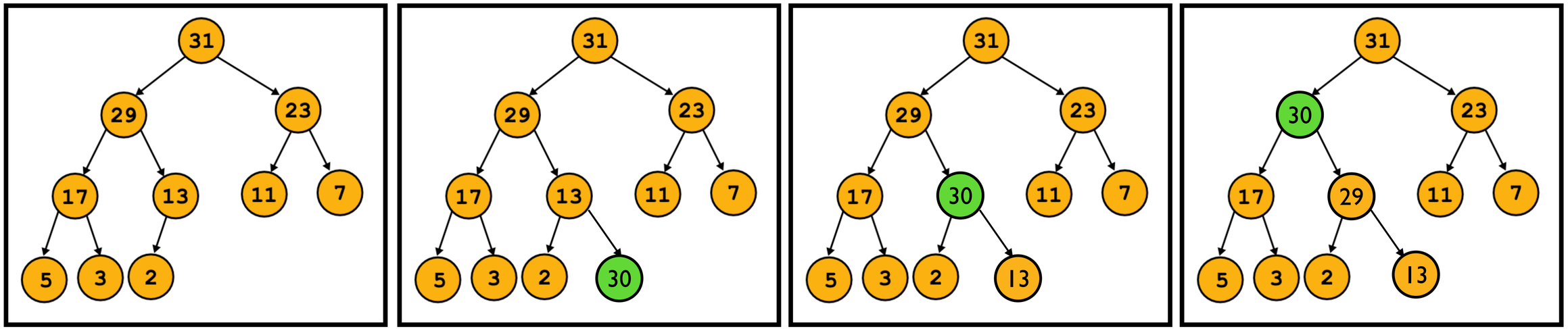






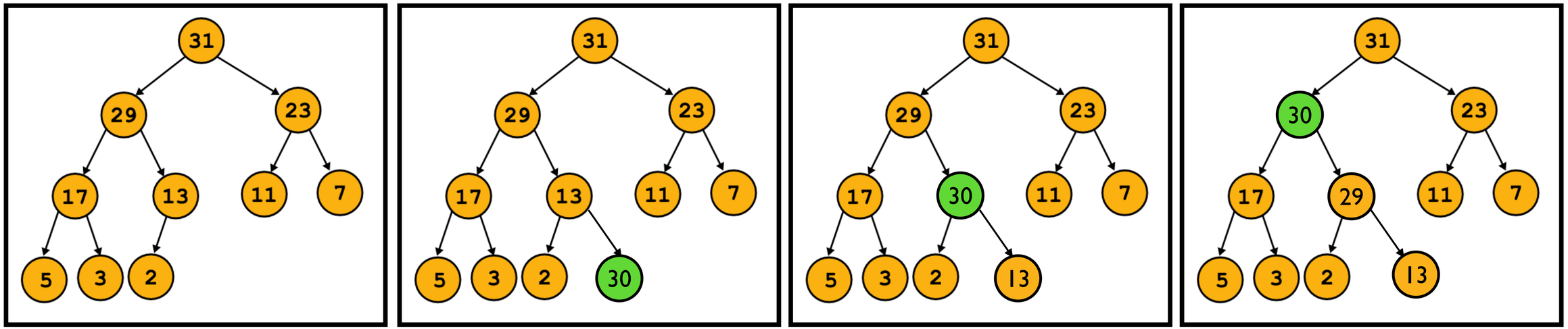
Assignment 6

- push / insert
  - append new element
  - percolate / bubble up



Assignment 6

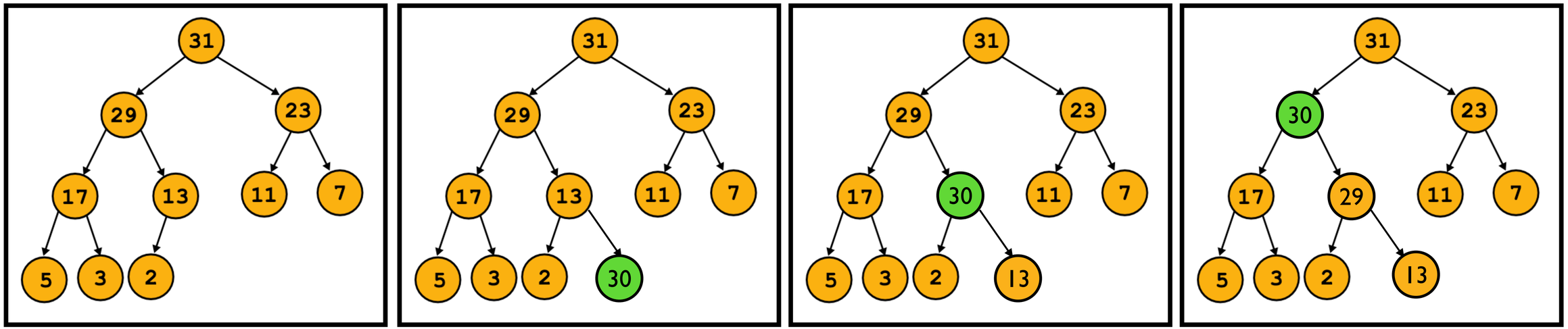
- push / insert
  - append new element
  - percolate / bubble up
    - current node is not the root
    - higher priority than parent



Assignment 6

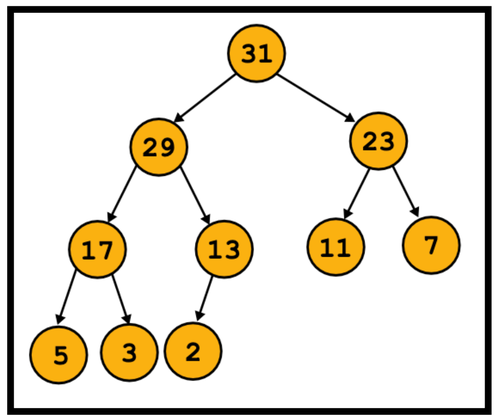
- push / insert
  - append new element
  - percolate / bubble up
    - current node is not the root
    - higher priority than parent

```c
void pq_insert(struct pq* pq, void* data, int priority) {
  struct element* new_element;

  //Ensure pre-conditions are met
  assert(pq);

  //Create new element
  new_element = (struct element*) malloc(sizeof(struct element));
  new_element->priority = priority;
  new_element->data = data;

  //Add to end of priority queue
  dynarray_insert(pq->array, _pq_length(pq), new_element);

  //Percolate up from just-inserted element
  _percolate_up(pq, _pq_length(pq) - 1);
}
```



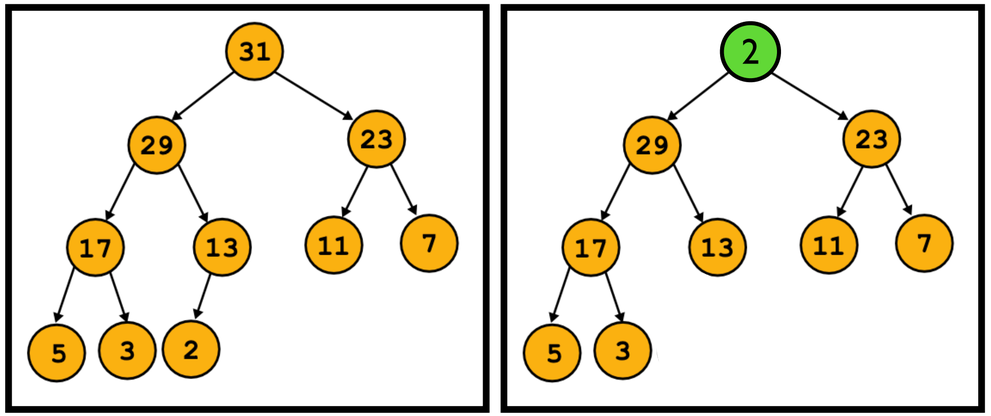
Assignment 6

- pop / dequeue
  - move the last element to the root
  - percolate / bubble down



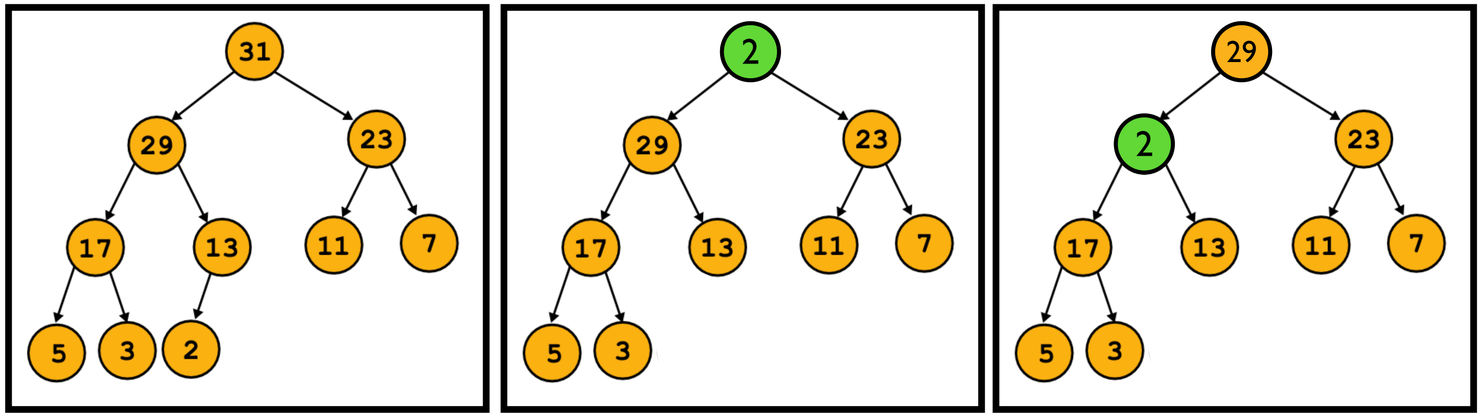
Assignment 6

- pop / dequeue
  - move the last element to the root
  - percolate / bubble down



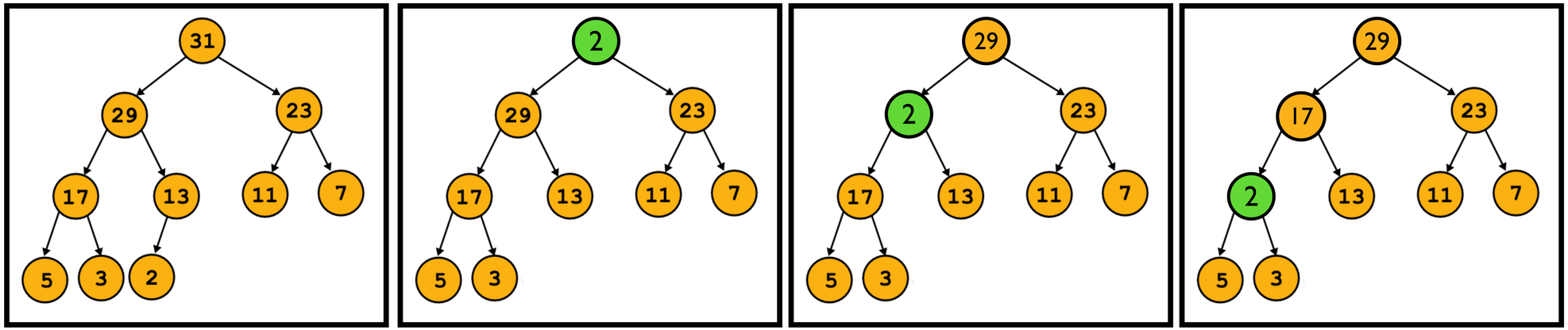
Assignment 6

- pop / dequeue
  - move the last element to the root
  - percolate / bubble down



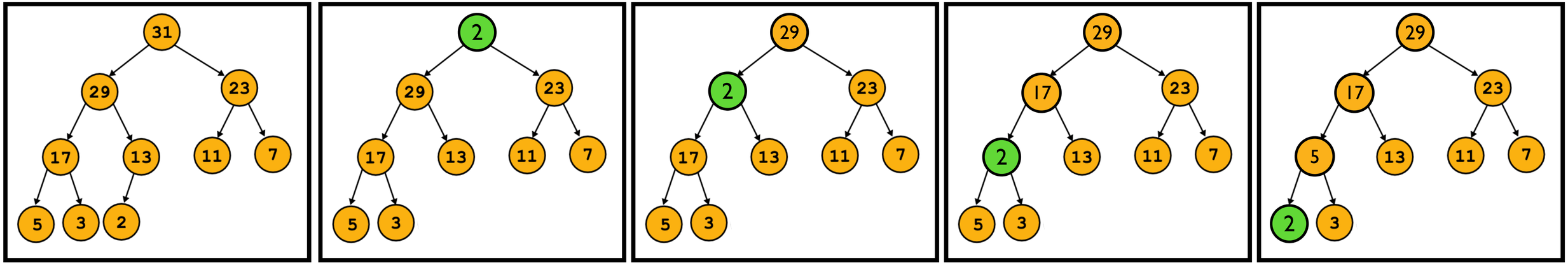
Assignment 6

- pop / dequeue
  - move the last element to the root
  - percolate / bubble down



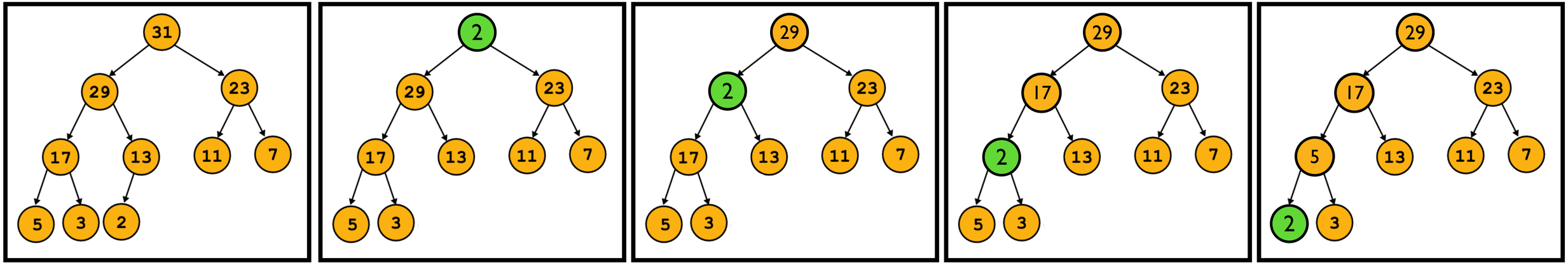
Assignment 6

- pop / dequeue
  - move the last element to the root
  - percolate / bubble down



Assignment 6

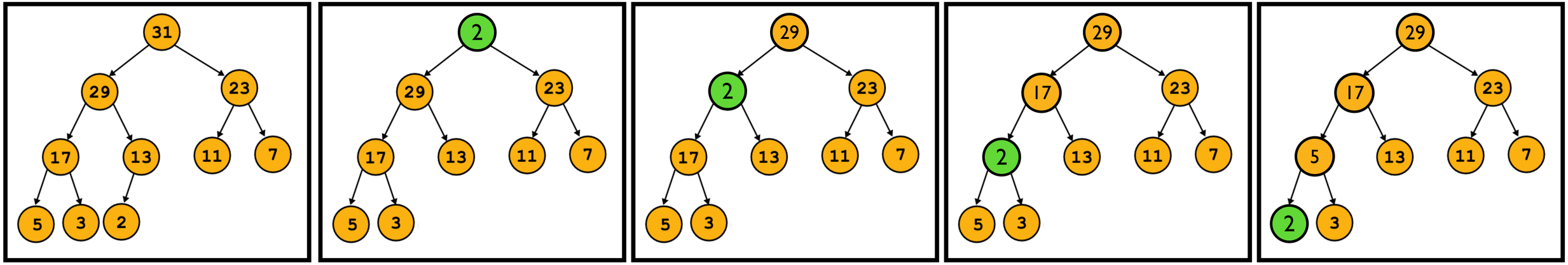
- pop / dequeue
  - move the last element to the root
  - percolate / bubble down
    - current node is not a leaf
    - lower priority than children



Assignment 6

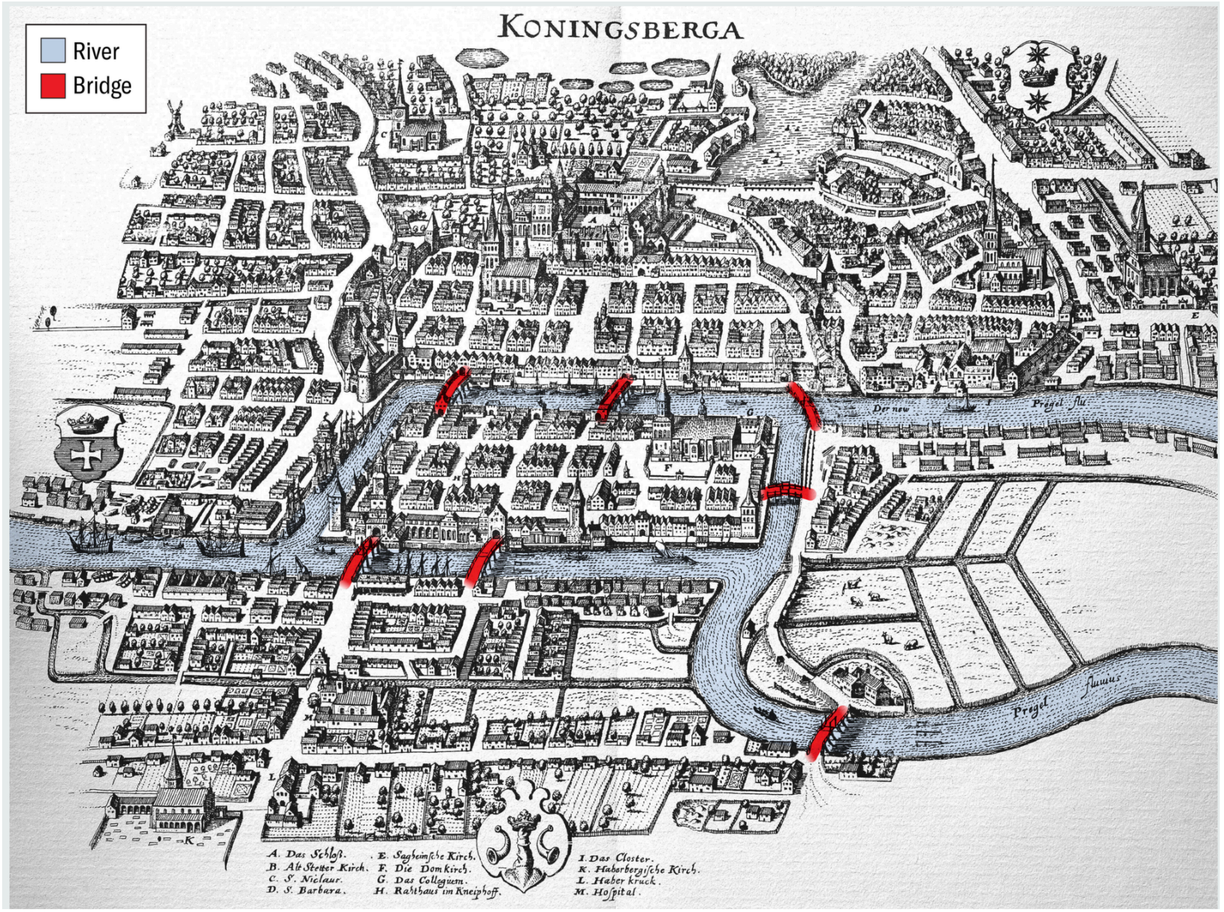
- pop / dequeue
  - move the last element to the root
  - percolate / bubble down
    - current node is not a leaf
    - lower priority than children

```
void* pq_max_dequeue(struct pq* pq) {
  void* value = pq_max(pq);

  //Ensure pre-conditions are met
  assert(!pq_isempty(pq));

  //Swap first and last elements
  _swap(pq, 0, _pq_length(pq) - 1);

  //Remove last element and free memory
  free(dynarray_get(pq->array, _pq_length(pq) - 1));
  dynarray_remove(pq->array, _pq_length(pq) - 1);

  //Percolate down from root to maintain max-heap
  _percolate_down(pq);

  return value;
}
```



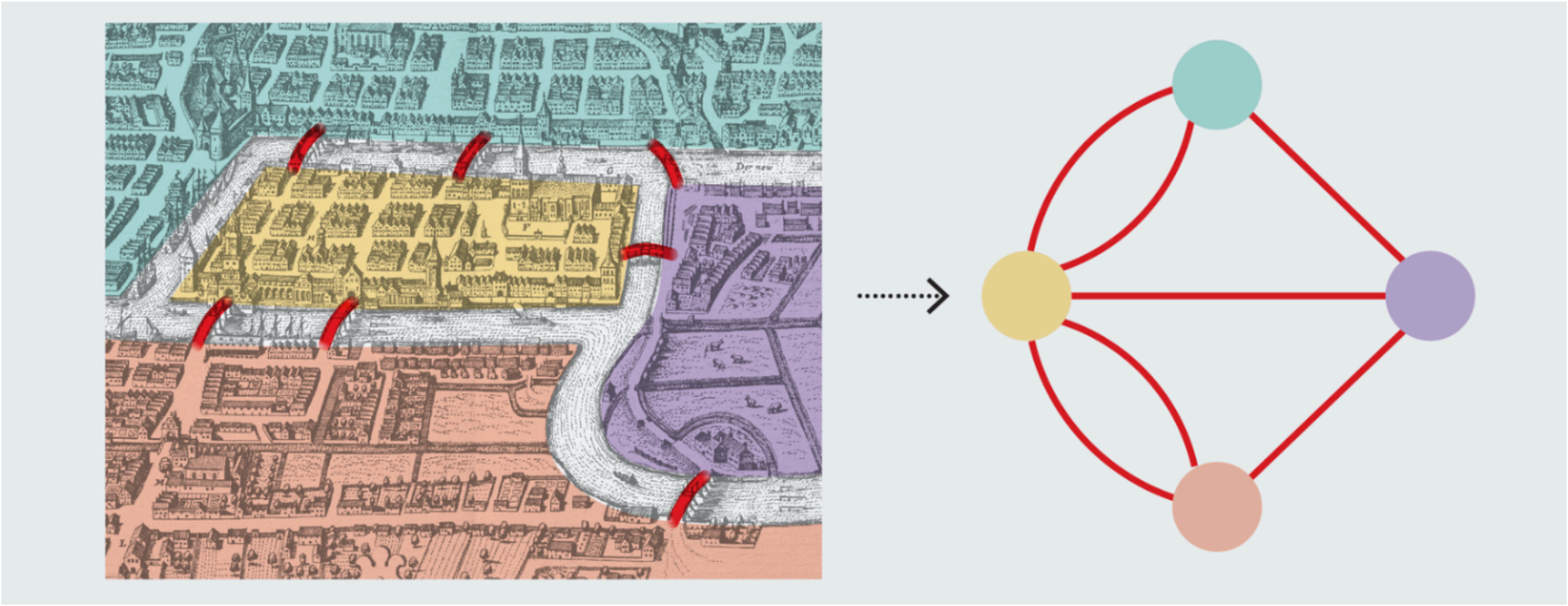
Graph

- Seven Bridges of Königsberg
  - find a walking path that can cross each bridge once and only once



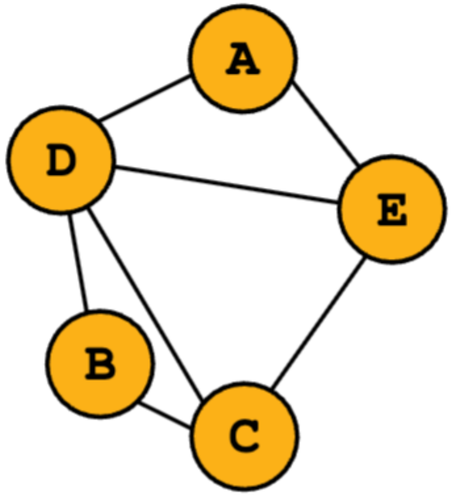
Graph

- Seven Bridges of Königsberg
  - find a walking path that can cross each bridge once and only once
  - Euler's abstraction
    - lands as nodes; bridges as edges



Graph

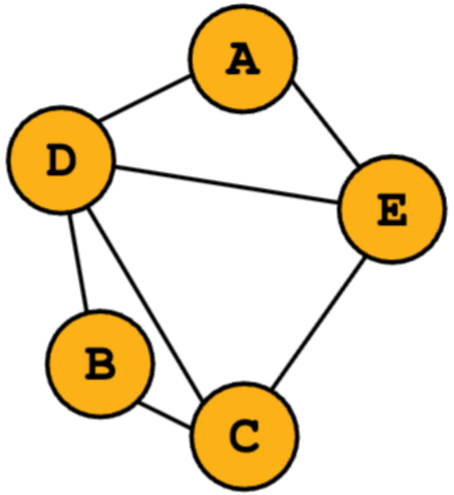
- G = (V, E)
  - vertices: $V = \{v_1, v_2, \ldots, v_n\}$
  - edges: $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$



$V = \{A, B, C, D, E\}$, and
$E = \{\{A, D\}, \{A, E\}, \{D, B\}, \{D, C\}, \{C, B\}, \{C, E\}\}$

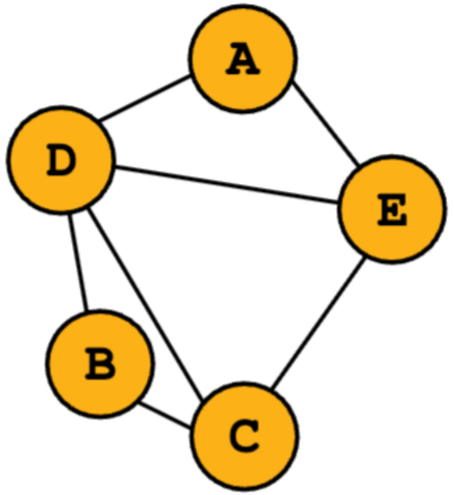
Graph

- examples of graph?



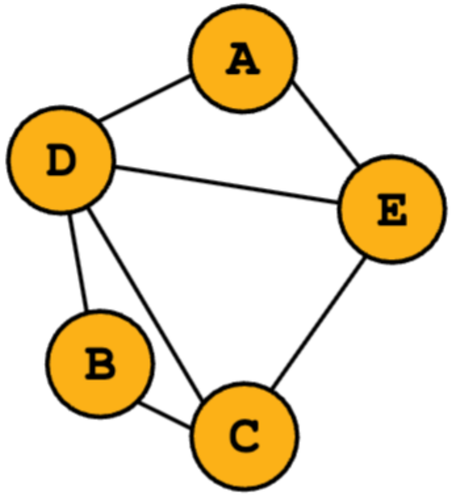
Graph

- examples of graph?
  - google map
  - flight networks
  - social networks
  - course prerequisites graph
  - software dependency graph
  - wikipedia linked documents



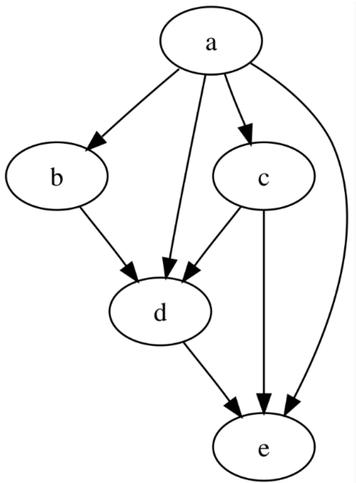
Graph Types

- many dimensions to describe graphs



Graph Types

- many dimensions to describe graphs
  - undirected vs. directed
    - Facebook vs. Twitter
  - sparse vs. dense
    - dense: $|E| \sim |V|^2$ (rarely)
    - sparse: $|E| << |V|^2$ (most graphs)
    - six degree of separation (due to hubs)
    - Dunbar's number (~150)
  - acyclic vs. cyclic
  - DAG: Directed Acyclic Graph
    - course & software dependence graph



Graph Types

- many dimensions to describe graphs (cont.)
  - connected vs. disconnected
    - undirected:
    - directed: strongly vs. weakly connected
      - strongly connected component (SCC)
      - SCC-DAG

Graph Types

- many dimensions to describe graphs (cont.)
  - connected vs. disconnected
    - undirected: 

    - directed: strongly vs. weakly connected
      - strongly connected component (SCC)
      - SCC-DAG

Graph Types

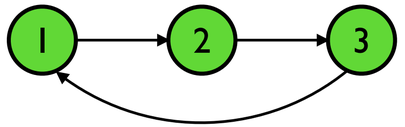
- many dimensions to describe graphs (cont.)
  - connected vs. disconnected
    - undirected: 

    - directed: strongly vs. weakly connected
      - strongly connected component (SCC)
      - SCC-DAG

Graph Types

- many dimensions to describe graphs (cont.)
  - connected vs. disconnected
    - undirected:

    - directed: strongly vs. weakly connected

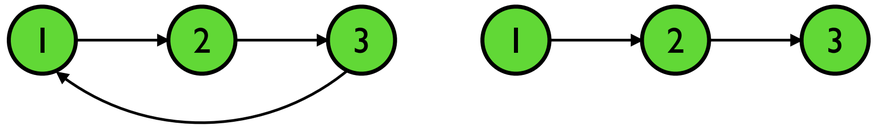


      - strongly connected component (SCC)
      - SCC-DAG

Graph Types

- many dimensions to describe graphs (cont.)
  - connected vs. disconnected
    - undirected:

    - directed: strongly vs. weakly connected

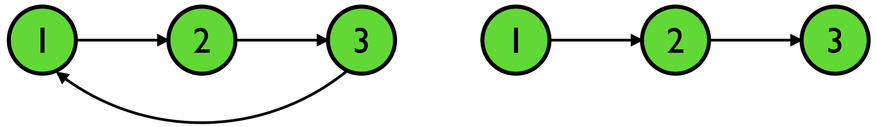


      - strongly connected component (SCC)
      - SCC-DAG

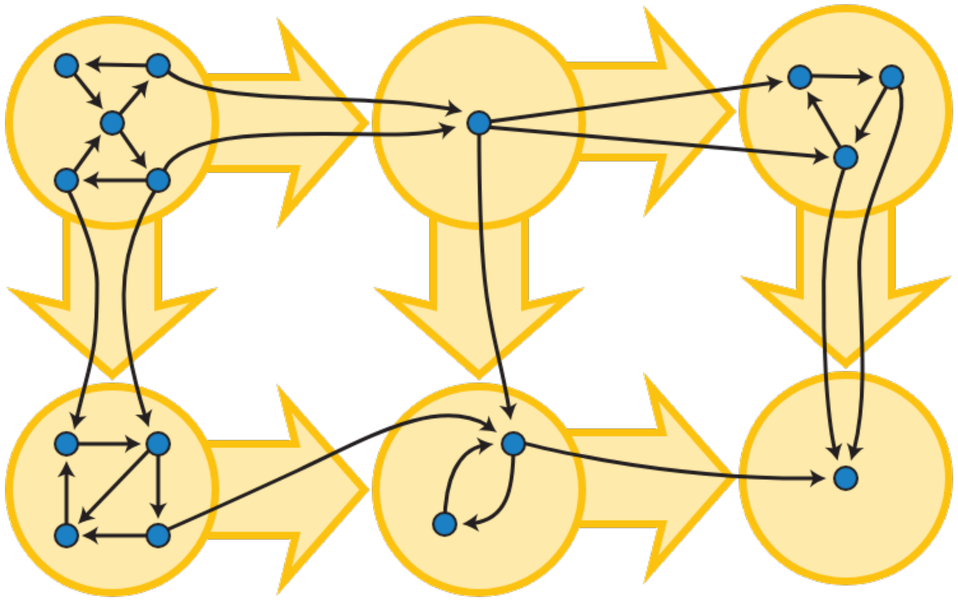
Graph Types

- many dimensions to describe graphs (cont.)
  - connected vs. disconnected
    - undirected:

    - directed: strongly vs. weakly connected



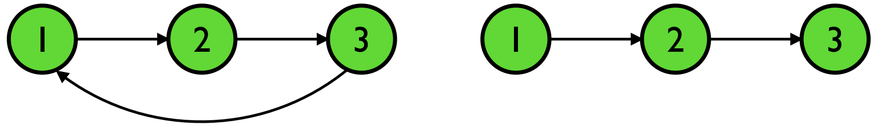
      - strongly connected component (SCC)
      - SCC-DAG



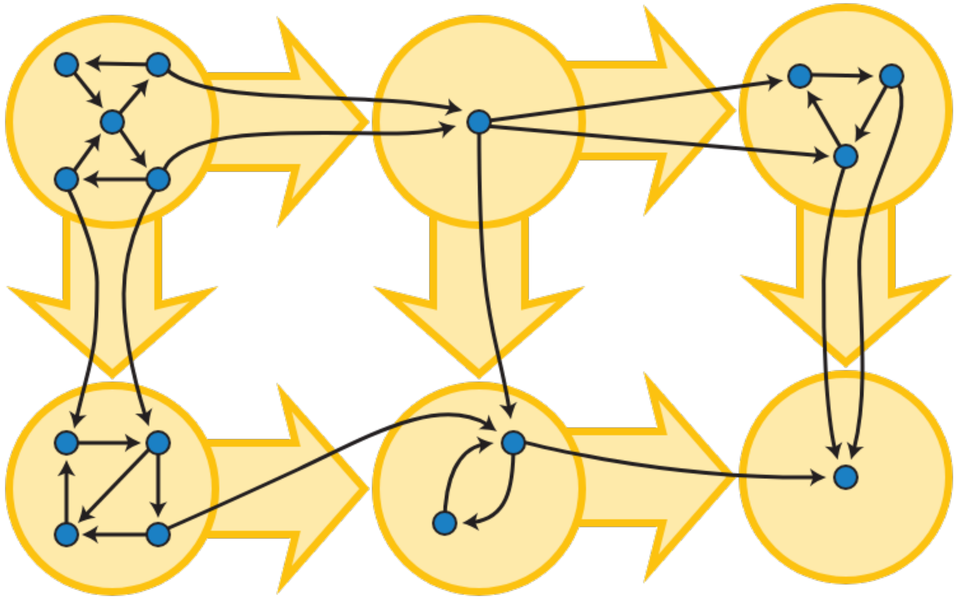
Graph Types

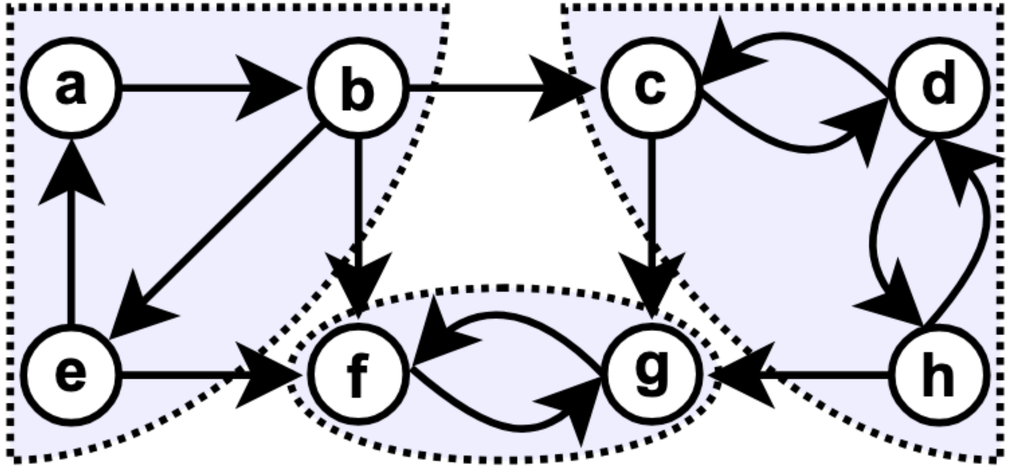
- many dimensions to describe graphs (cont.)
  - connected vs. disconnected
    - undirected:

    - directed: strongly vs. weakly connected



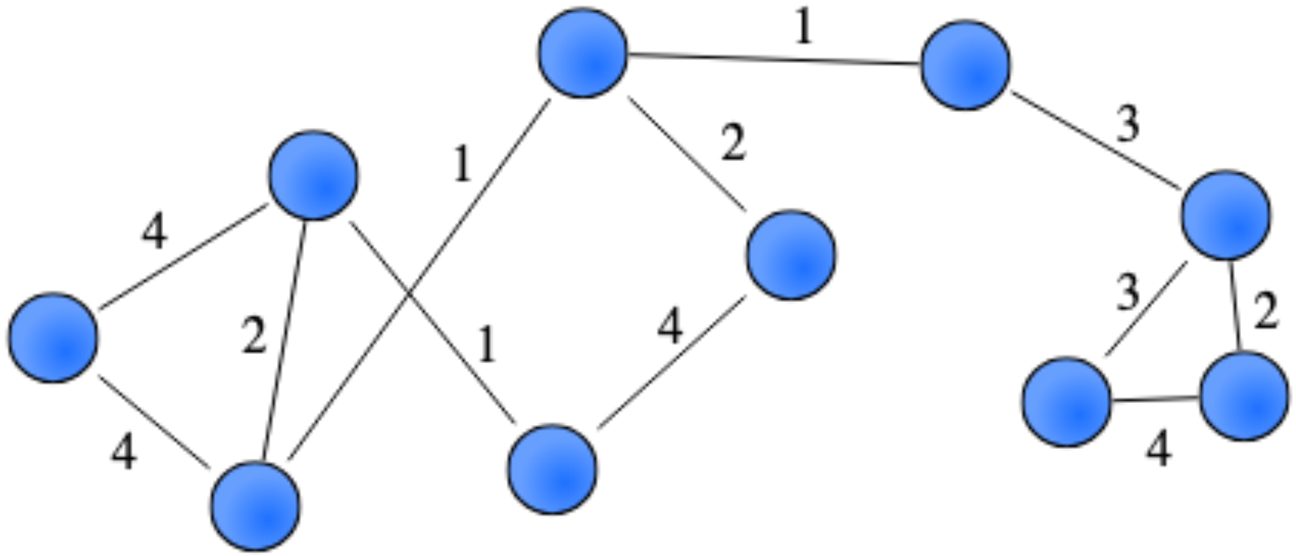
      - strongly connected component (SCC)
      - SCC-DAG





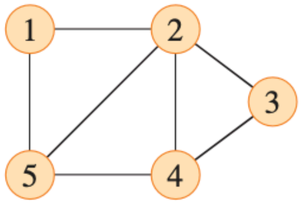
Graph Types

- many dimensions to describe graphs (cont.)
  - unweighted vs. weighted
    - vertices: $V = \{v_1, v_2, \ldots, v_n\}$
    - edges: $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$
    - weight function: $w : E \to \mathbb{R}$

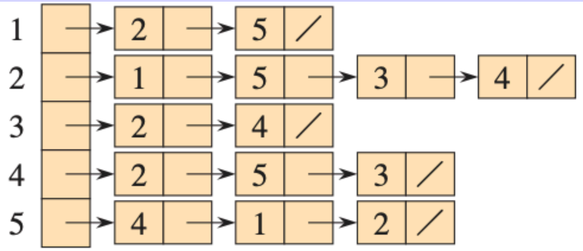


Graph Representations

- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors



undirected graph



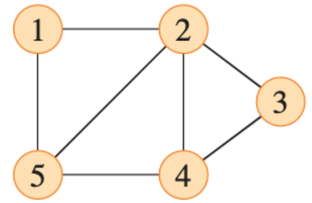
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

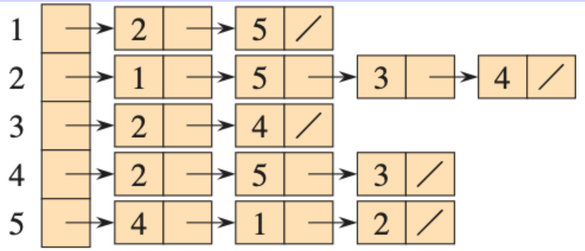
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Graph Representations

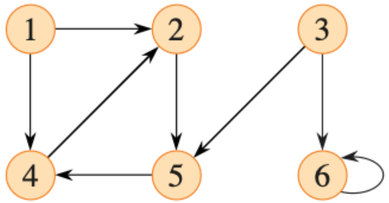
- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors



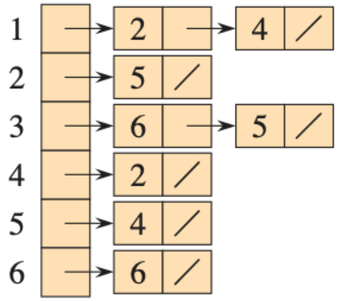
undirected graph



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

- adjacency matrix
  - $A = (a_{ij})$: $|V| \times |V|$ matrix

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

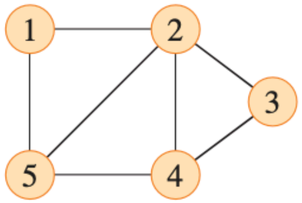
Graph Representations

- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors

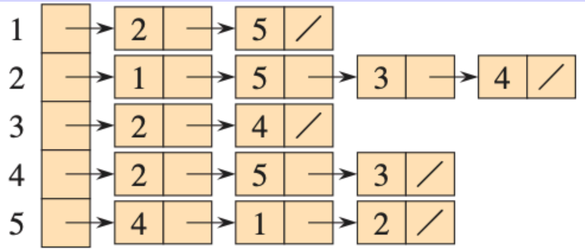
```
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* Adj[MAX_VERTICES];
```

  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

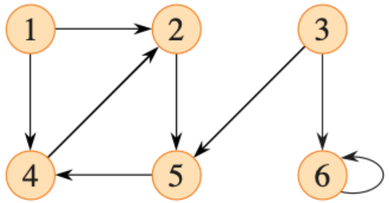
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$



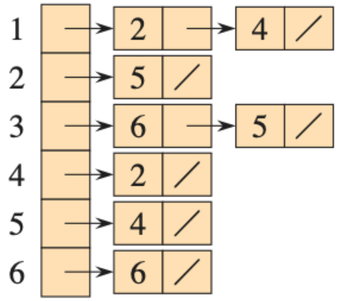
undirected graph



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Graph Representations

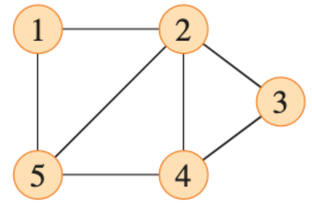
- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors

```
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* Adj[MAX_VERTICES];
```

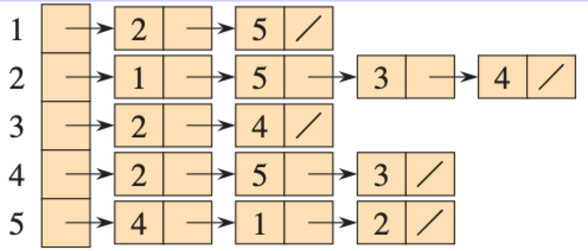
  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

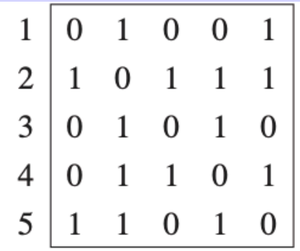
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

```
int A[MAX_VERTICES][MAX_VERTICES];
```

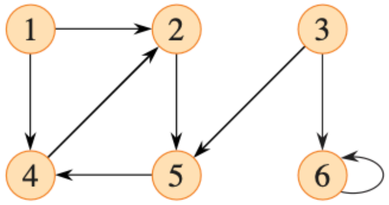


undirected graph

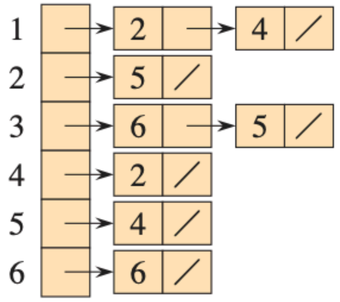


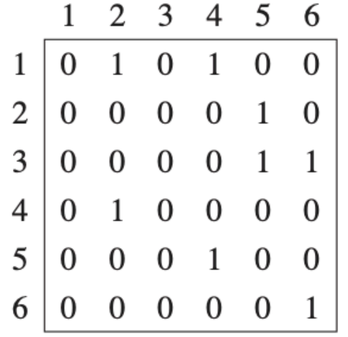


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Graph Representations

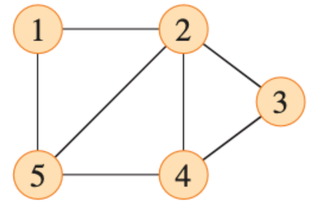
- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors

```
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* Adj[MAX_VERTICES];
```

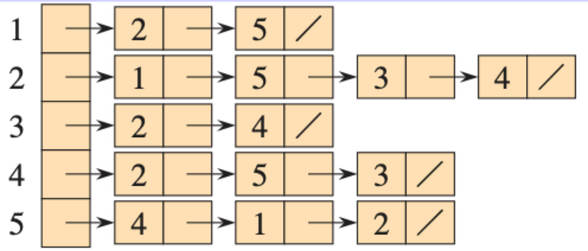
  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

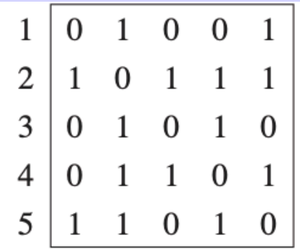
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

```
int A[MAX_VERTICES][MAX_VERTICES];
```

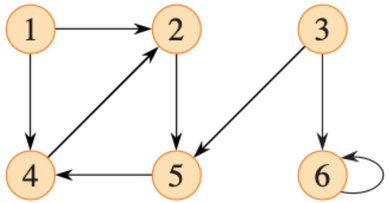


undirected graph

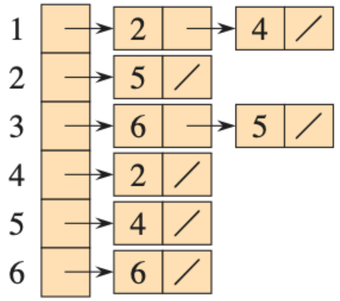


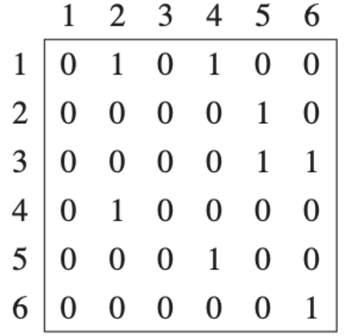


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Feature	Adjacency Matrix	Adjacency List
space complexity		
edge lookup		
iterating neighbors		
best for		
implementation complexity		

Graph Representations

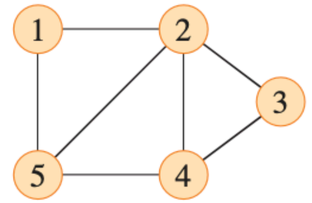
- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors

```
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* Adj[MAX_VERTICES];
```

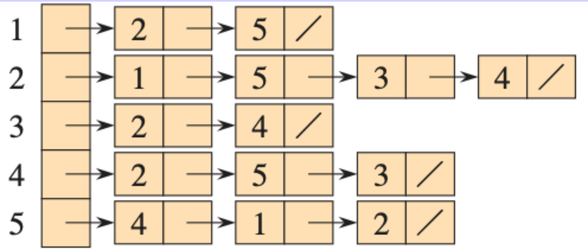
  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

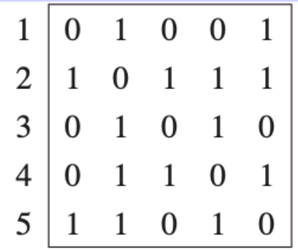
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

```
int A[MAX_VERTICES][MAX_VERTICES];
```

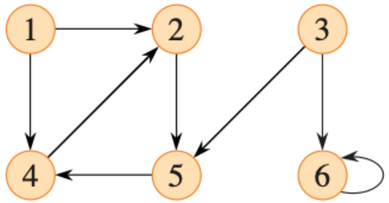


undirected graph

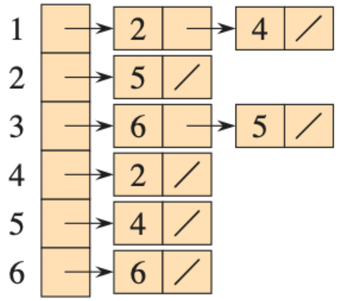


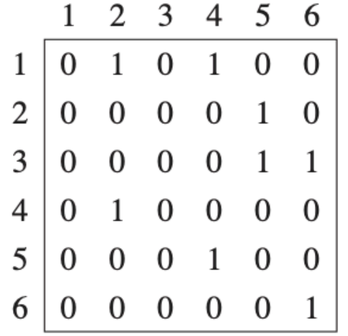


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Feature	Adjacency Matrix	Adjacency List
space complexity	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$
edge lookup		
iterating neighbors		
best for		
implementation complexity		

Graph Representations

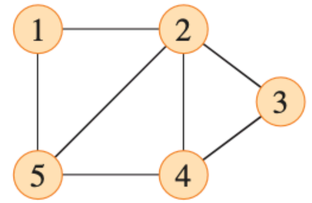
- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors

```
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* Adj[MAX_VERTICES];
```

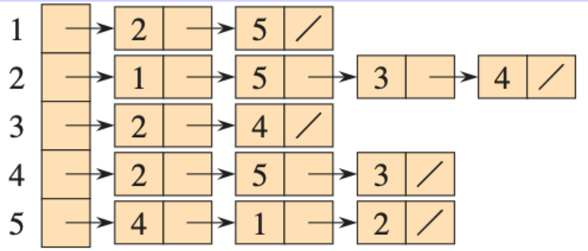
  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

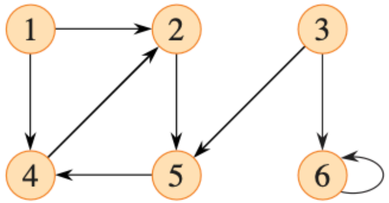
```
int A[MAX_VERTICES][MAX_VERTICES];
```



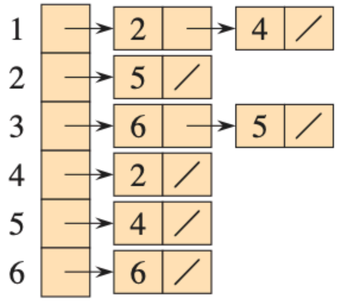
undirected graph



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Feature	**Adjacency Matrix**	**Adjacency List**
space complexity	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$
edge lookup	$O(1)$	$O(\text{degree})$
iterating neighbors		
best for		
implementation complexity		

Graph Representations

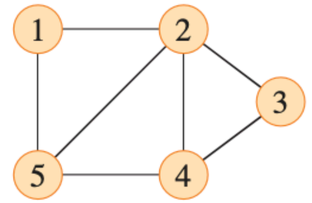
- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors

```
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* Adj[MAX_VERTICES];
```

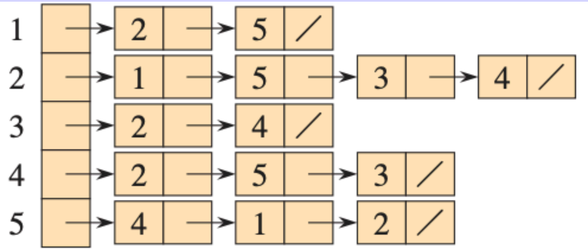
  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

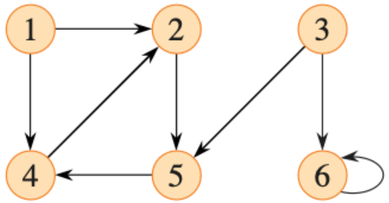
```
int A[MAX_VERTICES][MAX_VERTICES];
```



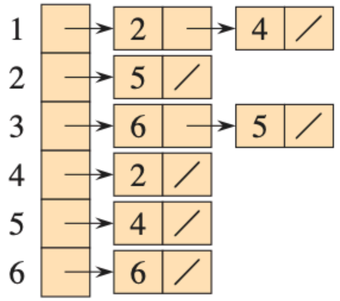
undirected graph



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Feature	**Adjacency Matrix**	**Adjacency List**
space complexity	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$
edge lookup	$O(1)$	$O(\text{degree})$
iterating neighbors	$O(\|V\|)$	$O(\text{degree})$
best for		
implementation complexity		

Graph Representations

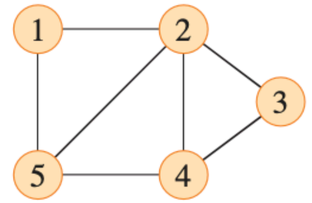
- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors

```
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* Adj[MAX_VERTICES];
```

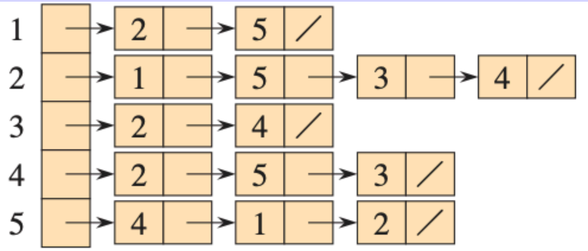
  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

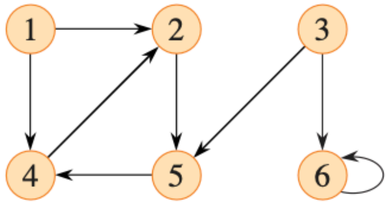
```
int A[MAX_VERTICES][MAX_VERTICES];
```



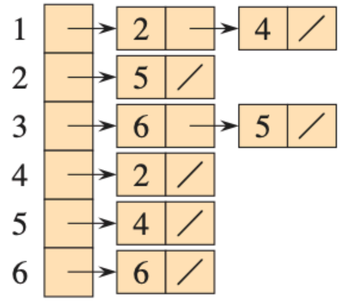
undirected graph



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Feature	**Adjacency Matrix**	**Adjacency List**
space complexity	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$
edge lookup	$O(1)$	$O(\text{degree})$
iterating neighbors	$O(\|V\|)$	$O(\text{degree})$
best for	dense graphs	sparse graphs
implementation complexity		

Graph Representations

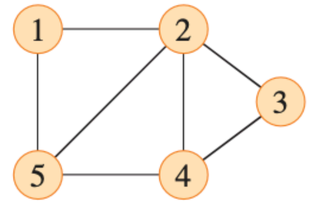
- G = (V, E)
  - adjacency list
    - Adj: $|V|$ lists
    - $Adj[v]$: v's neighbors

```
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* Adj[MAX_VERTICES];
```

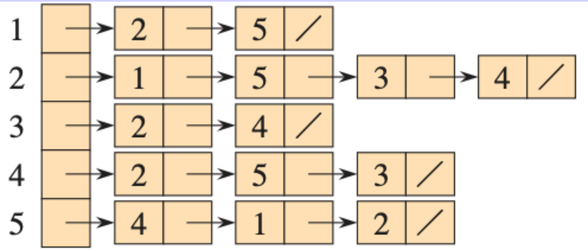
  - adjacency matrix
    - $A = (a_{ij})$: $|V| \times |V|$ matrix

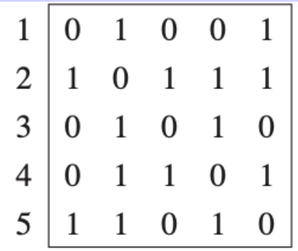
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

```
int A[MAX_VERTICES][MAX_VERTICES];
```

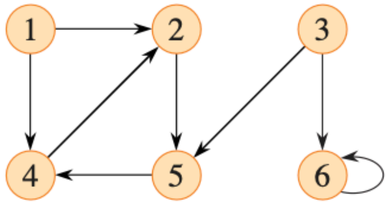


undirected graph

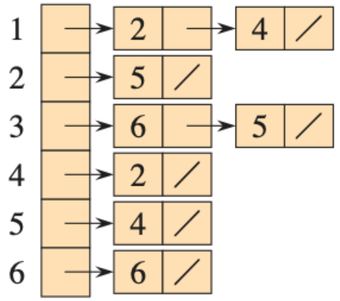


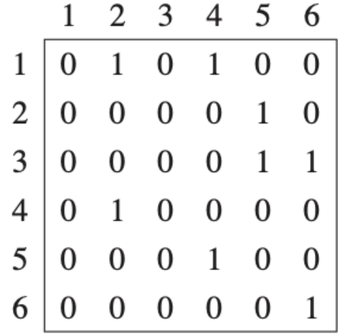


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



directed graph



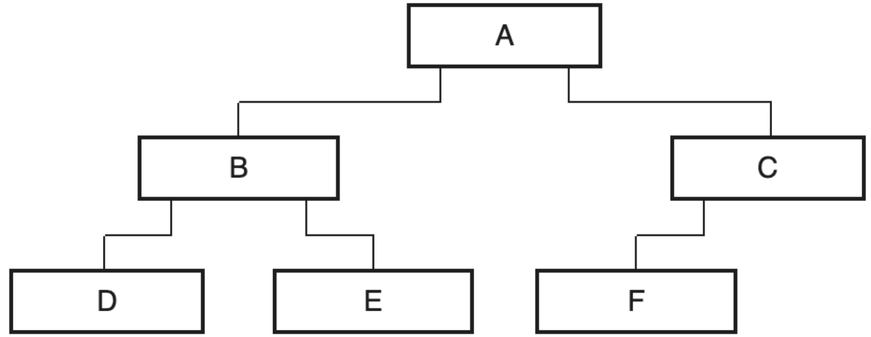


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Feature	**Adjacency Matrix**	**Adjacency List**
space complexity	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$
edge lookup	$O(1)$	$O(\text{degree})$
iterating neighbors	$O(\|V\|)$	$O(\text{degree})$
best for	dense graphs	sparse graphs
implementation complexity	simple	slightly complex

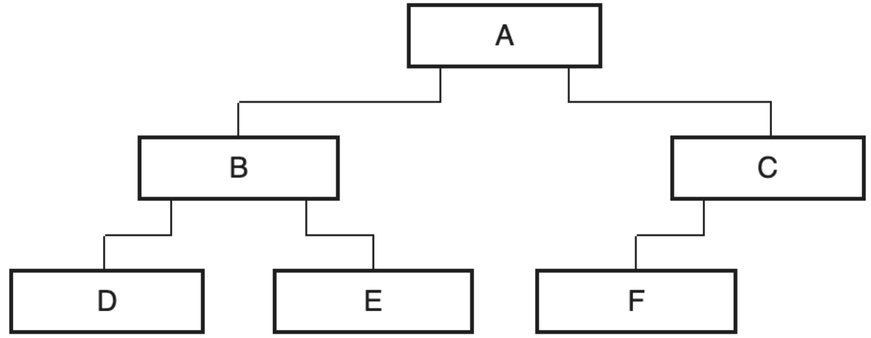
Graph Traversal

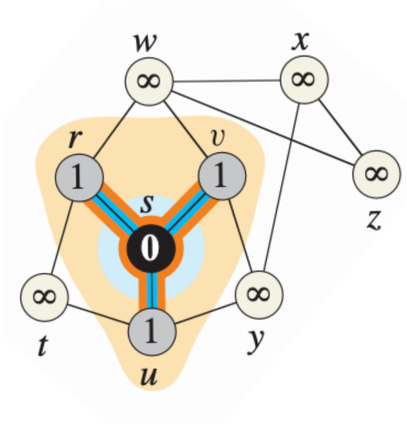
- recall binary tree traversals
  - Breadth First Search (BFS)
    - level-order: [A | B C | D E F]
  - Depth First Search (DFS)
    - pre-order: A (B (D) (E)) (C (F))
    - post-order: ((D) (E) B) ((F) C) A
    - in-order: ((D) B (E)) A ((F) C)
  - Tree is actually a special graph!

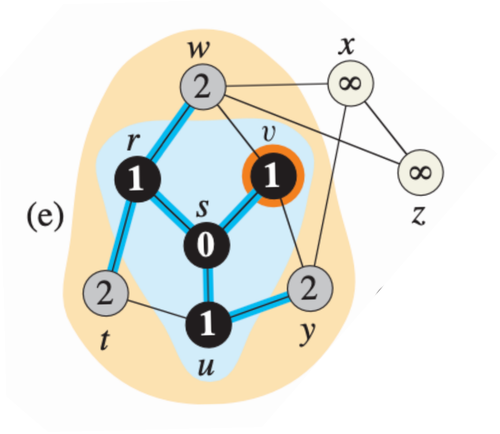


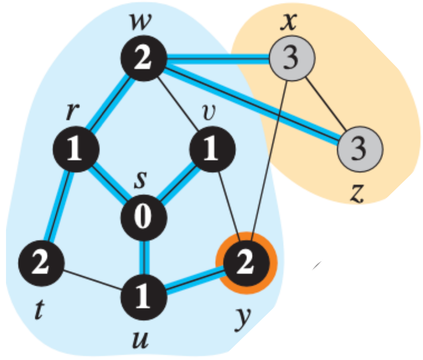
Graph BFS

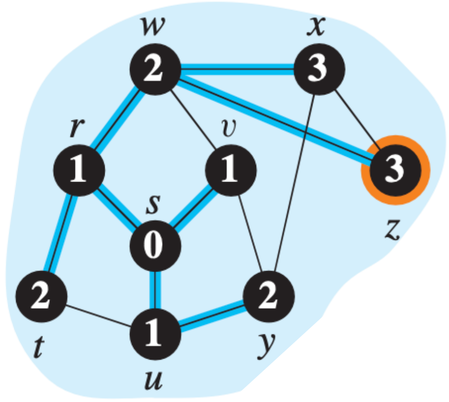
- tree BFS can be generalized to graph BFS
  - level from the root → distance from the source





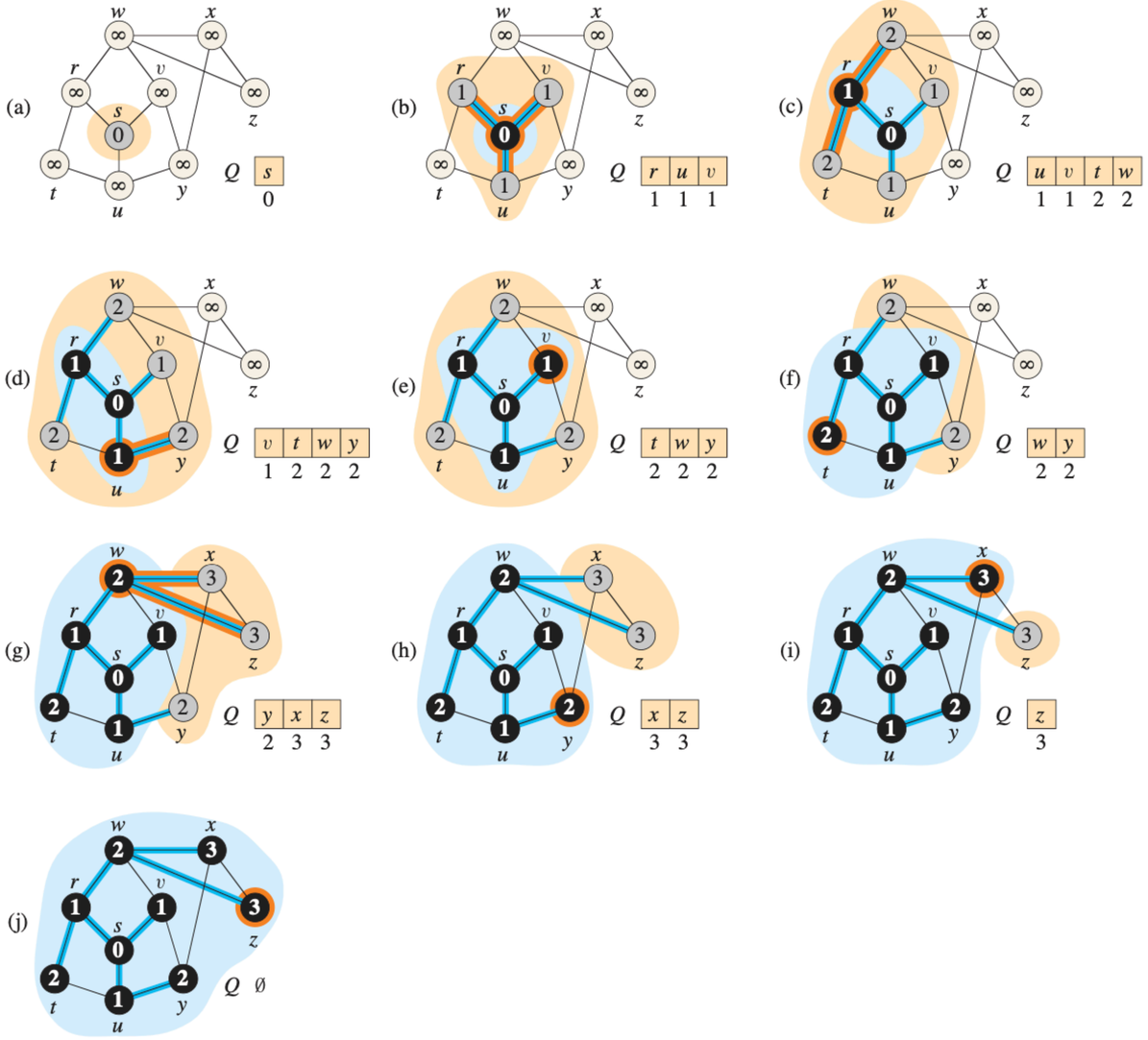






Graph BFS via Queue

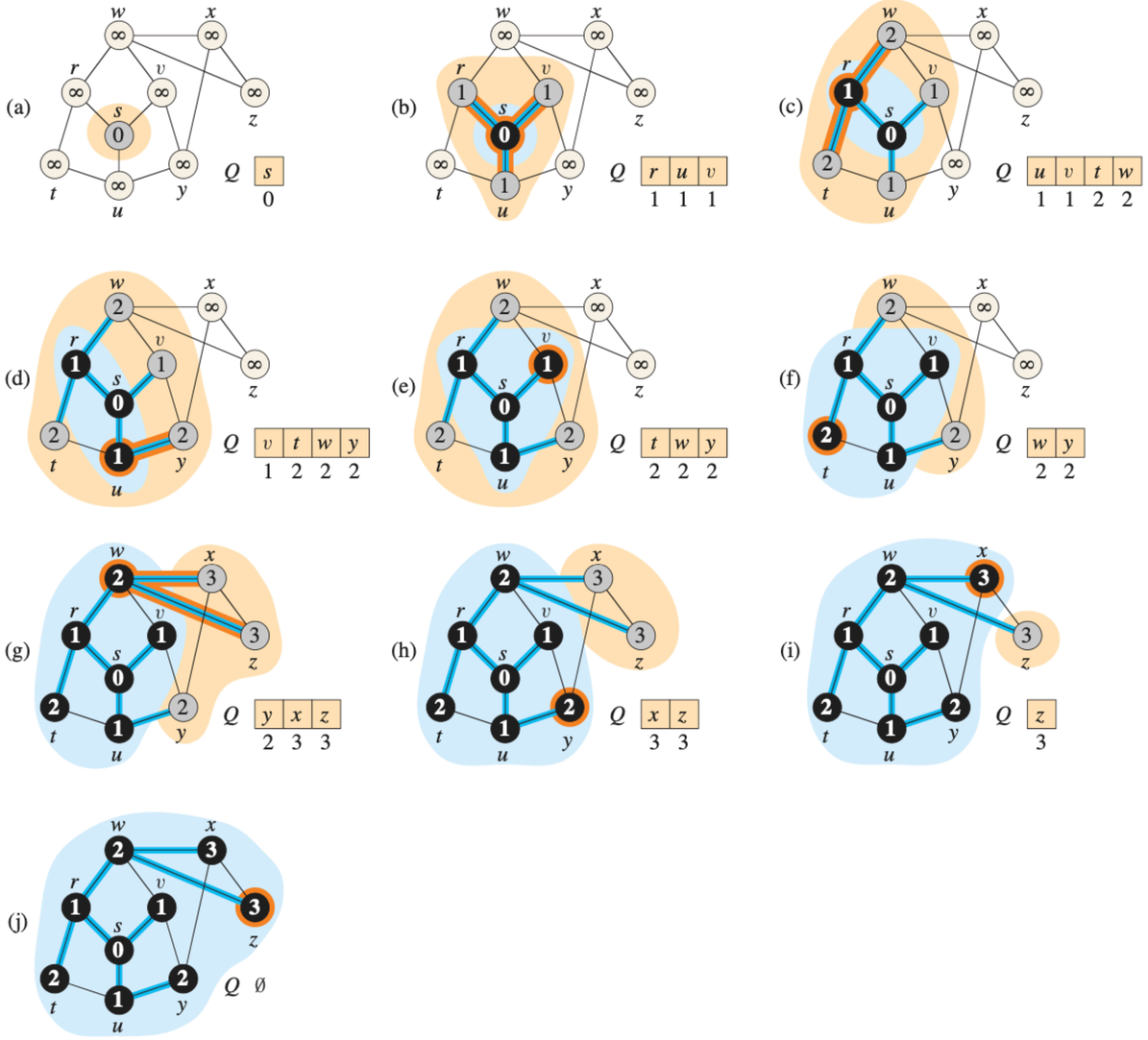
white: undiscovered gray: being visited black: visited




```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3
 4
 5  s.color = GRAY
 6
 7
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]      // search the neighbors of u
13          if v.color == WHITE            // is v being discovered now?
14              v.color = GRAY
15
16
17              ENQUEUE(Q, v)              // v is now on the frontier
18      u.color = BLACK                    // u is now behind the frontier
```

Graph BFS via Queue

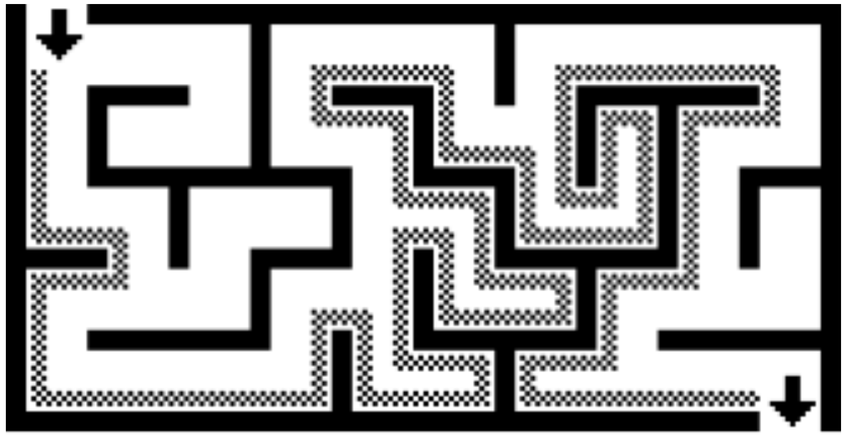
white: undiscovered gray: being visited black: visited



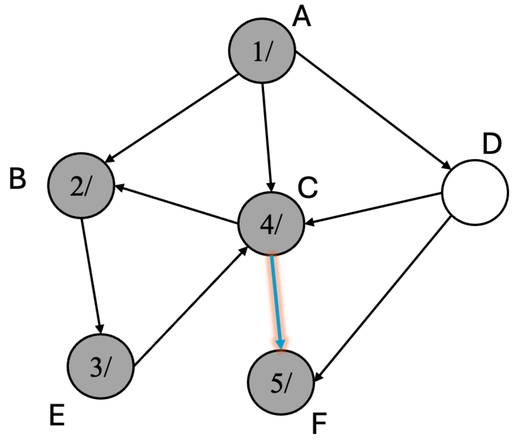

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞          //distance to source
 4      u.π = NIL        //parent in BFS
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]    // search the neighbors of u
13          if v.color == WHITE          // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)            // v is now on the frontier
18      u.color = BLACK                  // u is now behind the frontier
```

Graph DFS

- how to solve a maze
  - go as far as possible until hitting a wall
  - backtrack

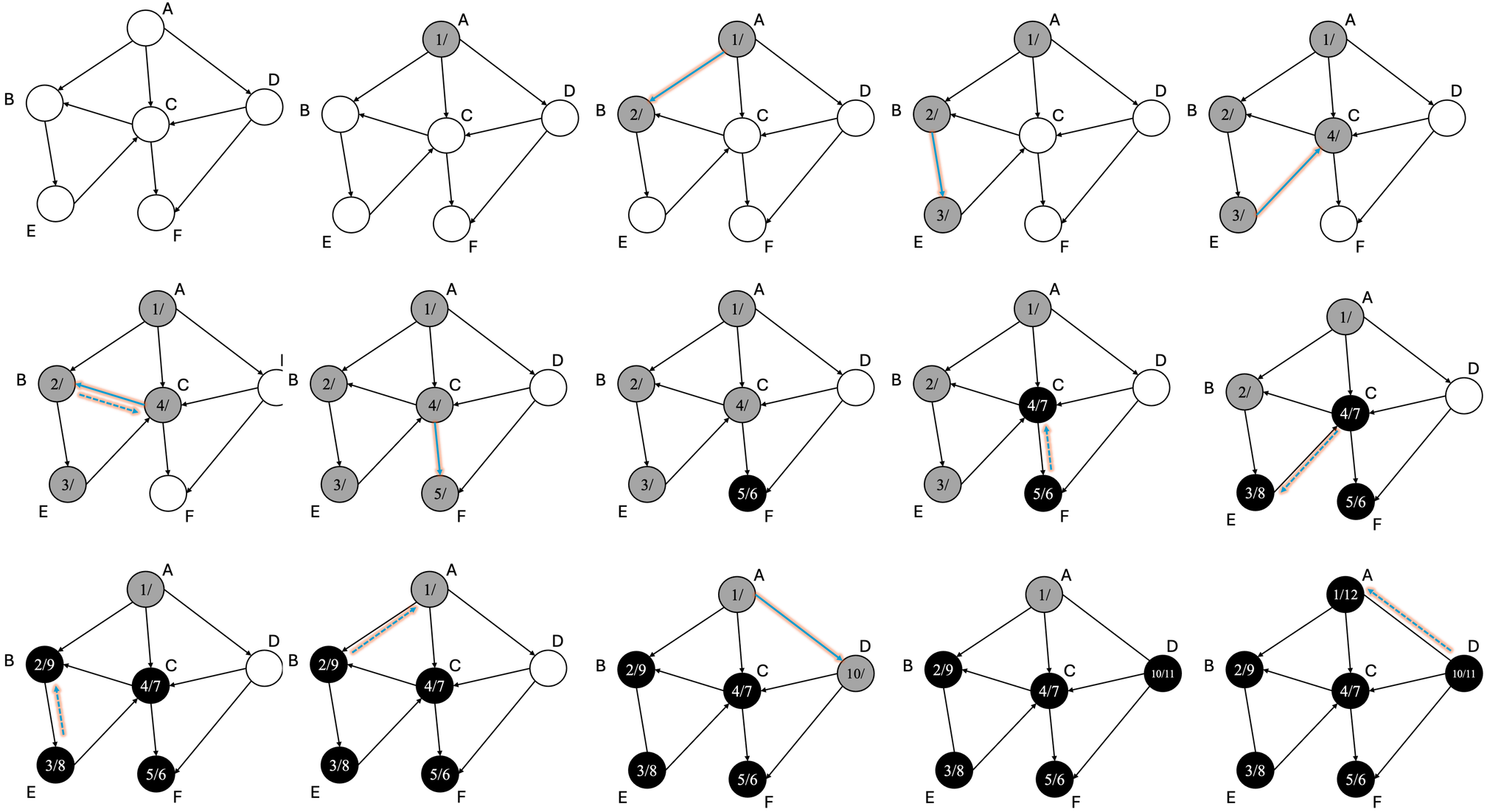


- Depth First Search (DFS)
  - go as deep as possible while avoiding revisit
  - backtrack



Graph DFS via Recursion

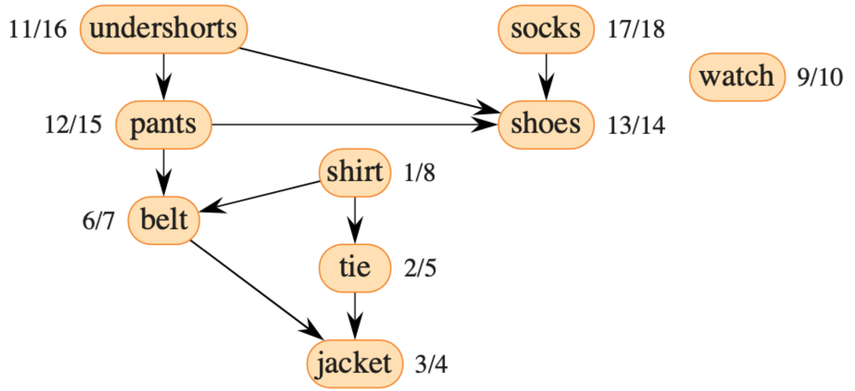
white: undiscovered gray: being visited black: visited




```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time //discover time
3  u.color = GRAY
4  for each vertex v in G.Adj[u]
5      if v.color == WHITE
6
7          DFS-VISIT(G, v)
8  time = time + 1
9  u.f = time //finish time
10 u.color = BLACK
```

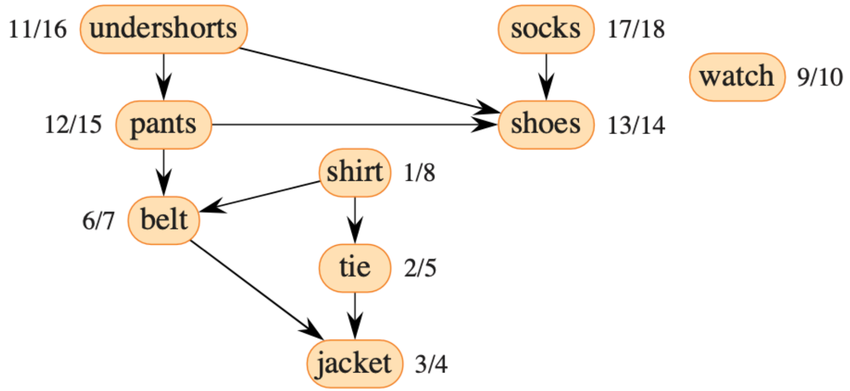
Topological Sort

- DAG (Directed Acyclic Graph)
  - course dependence graph
  - clothes wearing graph
- topological order
  - u comes before v if (u, v) in E
- Topological sort
  - DFS-based
  - Kahn's Algorithm (BFS-Based)
    - keep removing nodes with an in-dgree of 0

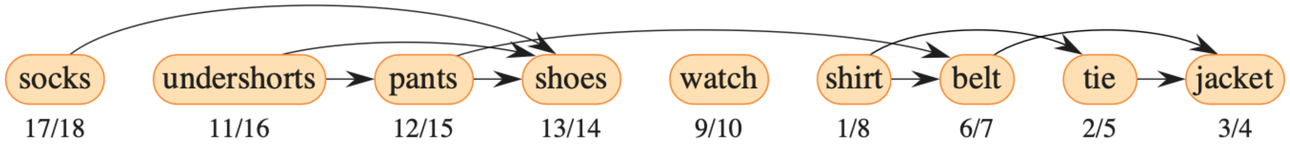


Topological Sort

- DAG (Directed Acyclic Graph)
  - course dependence graph
  - clothes wearing graph



- topological order
  - u comes before v if (u, v) in E
- Topological sort
  - DFS-based
  - Kahn's Algorithm (BFS-Based)
    - keep removing nodes with an in-dgree of 0



Topological Sort

- DAG (Directed Acyclic Graph)
  - course dependence graph
  - clothes wearing graph
- topological order
  - u comes before v if (u, v) in E
- Topological sort
  - DFS-based
  - Kahn's Algorithm (BFS-Based)
    - keep removing nodes with an in-dgree of 0