

Data Structures

Stefan D. Bruda

CS 317, Fall 2025



- **Stack** (FILO): push, pop, empty – constant time
- **Queue** (FIFO): insert, delete, empty – constant time
- **Heaps**: implementation of priority queue
 - Operations: insert ($O(\log n)$), peek (highest priority, $O(1)$), delete (highest priority, $O(\log n)$)
 - Tree representation, with children values smaller (maxheap) or larger (minheap) than the vertex value (weakly sorted)
 - Most efficiently implemented using arrays
 - Efficient sorting (**heapsort**)



- **Trees**: simple connected graph, one vertex may be designated as root
 - For a graph T with n vertices the following statements are equivalent:
 - T is a tree
 - T is connected and acyclic
 - T is connected and has $n - 1$ edges
 - T is acyclic and has $n - 1$ edges
 - Concepts: parent, ancestor, child, descendant, sibling, leaf, internal node
- **Binary tree**: each node had at most two children (left and right)
 - In a binary tree of height h with n nodes we have $h \geq \log_2 n$ (or $n \leq 2^h$)
 - Binary tree traversals ($O(n)$ complexity):

algorithm PREORDER(T):

```
if  $\neg$ EMPTY( $T$ ) then
    VISIT( $T$ )
    PREORDER(LEFT( $T$ ))
    PREORDER(RIGHT( $T$ ))
```

algorithm INORDER(T):

```
if  $\neg$ EMPTY( $T$ ) then
    INORDER(LEFT( $T$ ))
    VISIT( $T$ )
    INORDER(RIGHT( $T$ ))
```

algorithm POSTORDER(T):

```
if  $\neg$ EMPTY( $T$ ) then
    POSTORDER(LEFT( $T$ ))
    POSTORDER(RIGHT( $T$ ))
    VISIT( $T$ )
```

- **Binary search tree**: the value in every vertex is larger than all the values in its left subtree and smaller than all the values in its right subtree
 - Operations: insert, delete, search ($O(n)$ worst case, $O(\log n)$ if the tree is balanced)
 - Inorder traversal yields sorted sequence



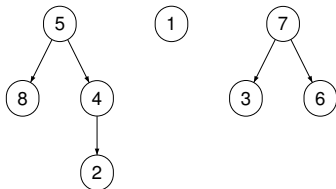
- Disjoint sets are non-empty, pairwise disjoint sets
 - Disjoint sets X_i , $1 \leq i \leq n$:
 $\forall 1 \leq i \leq n : X_i \neq \emptyset \quad \wedge \quad \forall 1 \leq i, j \leq n, i \neq j : X_i \cap X_j = \emptyset$
 - Each set has a member designated as the representative of that set
- Operations:
 - MAKESET(i): construct a set containing i as its sole element
 - FINDSET(i): return the representative of the set containing i
 - UNION(i, j): replaces the two sets containing i and j with their union; one of the two set representatives becomes the representative of the new set
- Representation: each set can be represented as a tree with the representative in the root
 - The tree does not have to be binary or balanced
- Implementation: disjoint sets over a domain D represented as an array *parent* indexed over D
 - $parent_i$ hold the parent of i in the tree representation, or i if i is the root



DISJOINT SETS (CONT'D)

- Example: $\{2, 4, \underline{5}, 8\}$, $\{\underline{1}\}$, $\{3, 6, \underline{7}\}$

Tree representation:



Array implementation:

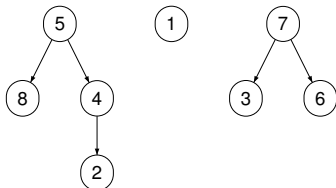
parent =

1	2	3	4	5	6	7	8
1	4	7	5	5	7	7	5

DISJOINT SETS (CONT'D)

- Example: $\{2, 4, 5, 8\}$, $\{1\}$, $\{3, 6, 7\}$

Tree representation:



Array implementation:

parent =

1	2	3	4	5	6	7	8
1	4	7	5	5	7	7	5

- A basic implementation:

algorithm MAKESET(*i*):

*parent*_{*i*} ← *i*

algorithm FINDSET(*i*):

while *parent*_{*i*} ≠ *i* **do** *i* ← *parent*_{*i*}
 return *i*

algorithm UNION(*i*, *j*):

x ← FINDSET(*i*)
 y ← FINDSET(*j*)
 if *x* ≠ *y* **then** MERGETREES(*x*, *y*)

algorithm MERGETREES(*i*, *j*):

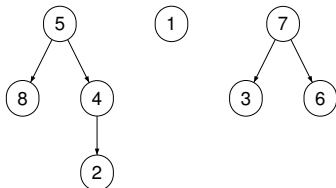
*parent*_{*i*} ← *j*



DISJOINT SETS (CONT'D)

- Example: $\{2, 4, 5, 8\}$, $\{1\}$, $\{3, 6, 7\}$

Tree representation:



Array implementation:

parent =

1	2	3	4	5	6	7	8
1	4	7	5	5	7	7	5

- A basic implementation:

algorithm MAKESET(*i*):

$\text{parent}_i \leftarrow i$

algorithm FINDSET(*i*):

while $\text{parent}_i \neq i$ **do** $i \leftarrow \text{parent}_i$
 return i

algorithm UNION(*i*, *j*):

$x \leftarrow \text{FINDSET}(i)$
 $y \leftarrow \text{FINDSET}(j)$
 if $x \neq y$ **then** MERGETREES(x, y)

algorithm MERGETREES(*i*, *j*):

$\text{parent}_i \leftarrow j$

- The tree representation can become very linear (depending on the sequence of calls to UNION), so the running times are as follows:

- MAKESET: $O(1)$
- FINDSET: $O(n)$
- UNION: $O(n)$ (since it calls FINDSET)



DISJOINT SETS (CONT'D)

- **Weighed union:** To maintain a smaller tree height for the union we decide what tree gets the root based on the heights of the operands
- Maintain a height for each set (tree)
- During union the tree with the smallest height is attached to the root of the set with the larger height
 - The height stays the same
- When the two operands have the same height attach one to another (no matter which, but consistently)
 - The height increases by one
 - Overall for every two sets joined we have a height increase of at most one so no height in the tree is going over $\log n$
 - Better running times:
 - MAKESET: $O(1)$
 - FINDSET: $O(\log n)$
 - UNION: $O(\log n)$ (since it calls FINDSET)

algorithm WUNION(i, j):

```
   $x \leftarrow \text{FINDSET}(i)$ 
   $y \leftarrow \text{FINDSET}(j)$ 
  if  $x \neq y$  then WMERGETREES( $x, y$ )
```

algorithm WMERGETREES(i, j):

```
  if  $\text{height}_i > \text{height}_j$  then  $\text{parent}_j \leftarrow i$ 
  else
     $\text{parent}_i \leftarrow j$ 
    if  $\text{height}_i = \text{height}_j$  then
       $\text{height}_j \leftarrow \text{height}_j + 1$ 
```




- **Collapsing find:** Each time we call FINDSET we collapse all the nodes we traverse so that they become connected directly to the root

algorithm CFINDSET(i):

```

    if  $i \neq \text{parent}_i$  then  $\text{parent}_i \leftarrow \text{CFINDSET}(\text{parent}_i)$ 
    return  $\text{parent}_i$ 

```

- When using weighted union alone n MAKESET and m WUNION/FINDSET takes $O(n + m \log n)$ time
- When using weighted union and collapsing find n MAKESET and m WUNION/CFINDSET takes $O(n + m + \alpha(n, m))$ time where $\alpha(n, m)$ is a constant for all practical purposes

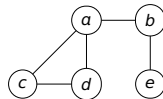
	MAKESET(i)	FIND(i)	UNION(i, j)	n MAKESET + m UNION/FIND
Basic impl.	$O(1)$	$O(n)$	$O(n)$	$O(n + nm)$
Weighted union	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n + m \log n)$
Weighted union + collapsing find	$O(1)$	$O(\log n)$	$O(\log n)$	$\approx O(n + m)$



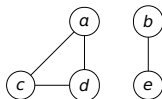
- **Directed graph** (digraph): $G = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is the set of edges
 - In a graphical representation edges are shown as arrows between vertices
- **Undirected graph**: A graph $G = (V, E)$ with the additional property that $(u, v) \in E$ iff $(v, u) \in E$
 - In a graphical representation edges are shown as lines between vertices
- **Weighted graph**: $G = (V, E, w)$ where (V, E) is a graph and $w : E \rightarrow \mathbb{R}$ associates a weight to each edge
 - In a graphical representation weights are shown as edge labels
- Concepts related to graphs:
 - adjacent vertices, degree, in degree, out degree
 - complement of $G = (V, E)$: $G' = (V, V \times V \setminus E)$
 - path, simple path, cycle, simple cycle
 - acyclic graph
 - length of the shortest path from u to v : $\text{DIST}(u, v)$
 - diameter of $G = (V, E)$: $\text{DIAM}(G) = \max\{\text{DIST}(u, v) : u, w \in V\}$
 - subgraph: a subset of edges along with all their vertices
 - induced subgraph: contains all the edges between its vertices
 - Hamiltonian cycle: cycle that contains each vertex exactly once
 - Euler cycle: cycle that contains each edge exactly once

- **(Strongly) connected graph**: graph that has a path between each pair of vertices
 - For a connected graph $G = (V, E)$ what is the minimum and the maximum $|E|$ (in terms of $|V|$)?
- **Weakly connected graph**: directed graph that is not connected but becomes connected if we transform it into an undirected graph
 - No concept of weak connectivity for undirected graphs (they are either connected or not)
- **Clique** or complete graph: $G = (V, V \times V)$
- **Sparse** vs **dense** graphs
- **Bipartite graph**: $G = (V_1 \uplus V_2, E)$ such that $E \subseteq V_1 \times V_2 \cup V_2 \times V_1$
 - **Complete bipartite graph**:
 $G = (V_1 \uplus V_2, V_1 \times V_2 \cup V_2 \times V_1)$

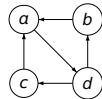
Connected:



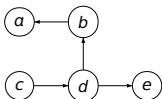
Unconnected:



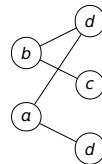
Strongly connected:



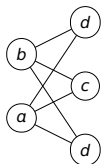
Weakly connected:



Bipartite:



Complete bipartite:



GRAPH REPRESENTATION

Adjacency matrix

- For $G = (V, E)$ establish an (arbitrary) order over V , such that we can consider $V = \{0, 1, \dots, n\}$
- Then G can be represented as the binary matrix $(G_{ij})_{0 \leq i, j \leq n}$ such that $G_{ij} = 1$ iff $(i, j) \in E$
- For a weighted $G = (V, E, w)$ set $G_{ij} = w(i, j)$ if $(i, j) \in E$ and $G_{ij} = \infty$ otherwise

Undirected:

	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	0	1
c	1	0	0	1	0
d	1	0	1	0	0
e	0	1	0	0	0

Directed:

	a	b	c	d	e
a	0	0	0	1	0
b	1	0	0	0	0
c	1	0	0	1	0
d	0	0	0	0	0
e	0	1	0	0	0

Weighted:

	a	b	c	d	e
a	∞	5	2	1	∞
b	5	∞	∞	∞	8
c	2	∞	∞	2	∞
d	1	∞	2	∞	∞
e	∞	8	∞	∞	∞

- Adjacency list:** For each vertex v use a list with exactly all the vertices u such that $(v, u) \in E$
 - Include the weights if it is a weighted graph

a	$\rightarrow b \rightarrow c \rightarrow d$	a	$\rightarrow d$	a	$\rightarrow b, 5 \rightarrow c, 2 \rightarrow d, 1$
b	$\rightarrow a \rightarrow e$	b	$\rightarrow a$	b	$\rightarrow a, 5 \rightarrow e, 8$
c	$\rightarrow a \rightarrow d$	c	$\rightarrow a \rightarrow d$	c	$\rightarrow a, 2 \rightarrow d, 2$
d	$\rightarrow a \rightarrow c$	d		d	$\rightarrow a, 1 \rightarrow c, 2$
e	$\rightarrow b$	e	$\rightarrow b$	e	$\rightarrow b, 8$

- Time/space efficiency?

algorithm TRAVERSE($G = (V, E)$):

foreach $v \in V$ **do**

$visit_v \leftarrow \text{false}$

 Let $v \in V$ such that $visit_v = \text{false}$

if v exists **then**

 LISTTRAVERSE(v)

algorithm LISTTRAVERSE($v \in V$):

$open \leftarrow \langle v \rangle$

$visit_v \leftarrow \text{true}$

while $open \neq \langle \rangle$ **do**

$u \leftarrow \text{HEAD}(open)$

 Output u

$new \leftarrow \langle x : (u, x) \in E \wedge \neg visit_x \rangle$

foreach $x \in new$ **do** $visit_x \leftarrow \text{true}$

$open \leftarrow \text{REST}(open) \oplus new$



algorithm TRAVERSE($G = (V, E)$):

foreach $v \in V$ **do**

$visit_v \leftarrow \text{false}$

 Let $v \in V$ such that $visit_v = \text{false}$

if v exists **then**

 LISTTRAVERSE(v)

algorithm LISTTRAVERSE($v \in V$):

$open \leftarrow \langle v \rangle$

$visit_v \leftarrow \text{true}$

while $open \neq \langle \rangle$ **do**

$u \leftarrow \text{HEAD}(open)$

 Output u

$new \leftarrow \langle x : (u, x) \in E \wedge \neg visit_x \rangle$

foreach $x \in new$ **do** $visit_x \leftarrow \text{true}$

$open \leftarrow \text{REST}(open) \oplus new$

Two different variants of $\oplus$ yield two different traversals:

- **Breadth-first traversal:** $L' \oplus L'' = L' + L''$

- New vertices are added at the end and so $open$ implements a **queue**

- **Depth-first traversal:** $L' \oplus L'' = L'' + L'$

- New vertices are added at the beginning and so $open$ implements a **stack**

- Depth-first traversal can also be implemented recursively:

algorithm DFS($G = (V, E)$):

foreach $v \in V$ **do** $visit_v \leftarrow \text{false}$

 Let $v \in V$ such that $visit_v = \text{false}$

if v exists **then** RECDFS(v)

algorithm RECDFS($v \in V$):

 Output v

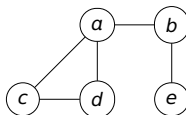
$visit_v \leftarrow \text{true}$

foreach $(v, u) \in E \wedge \neg visit_u$ **do**

 RECDFS(u)

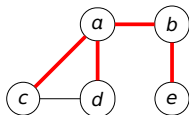


- Any traversal of a graph G avoids all edges that would result in cycles
- Therefore it only expands (and thus defines) an acyclic subgraph of G

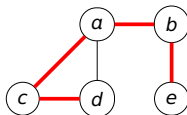


- Same traversal output starting from a : a, c, d, b, e
- Different traversal trees:

BFS tree:



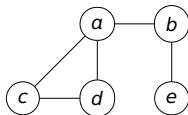
DFS tree:



- Both algorithms run in time $O(n + m)$

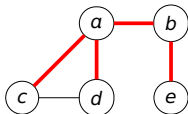


- Any traversal of a graph G avoids all edges that would result in cycles
- Therefore it only expands (and thus defines) an acyclic subgraph of G
= the **traversal (DFS or BFS) tree**

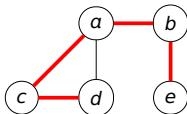


- Same traversal output starting from a : a, c, d, b, e
- Different traversal trees:

BFS tree:



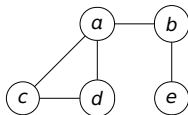
DFS tree:



- Both algorithms run in time $O(n + m)$

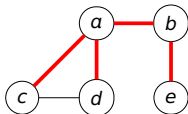


- Any traversal of a graph G avoids all edges that would result in cycles
- Therefore it only expands (and thus defines) an acyclic subgraph of G
= the **traversal (DFS or BFS) tree**

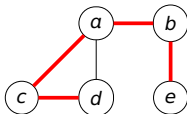


- Same traversal output starting from a : a, c, d, b, e
- Different traversal trees:

BFS tree:



DFS tree:



- Both algorithms run in time $O(n + m)$
- Space** requirements however are vastly different

TOPOLOGICAL SORTING ON DIRECTED GRAPHS

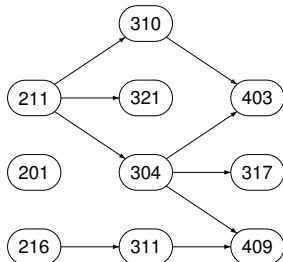


- Given a graph $G = (V, E)$, obtain a linear ordering of V such that for every edge $(u, v) \in E$, u comes before v in the ordering

algorithm TSORT($G = (V, E)$):

```
    order  $\leftarrow \langle \rangle$   
    S  $\leftarrow V$   
    while S  $\neq \emptyset$  do  
        Let  $v \in S$  with in-degree 0  
        order  $\leftarrow$  order +  $\langle v \rangle$   
        E  $\leftarrow E \setminus \{(v, u) \in E\}$   
        V  $\leftarrow V \setminus v$ 
```

- Many practical applications, e.g. sorting over a course prerequisite structure



algorithm TSORT'($G = (V, E)$):

```
    order  $\leftarrow \langle \rangle$   
    k  $\leftarrow n$   
    foreach  $v \in V$  do  $visit_v \leftarrow$  false  
    while  $\exists v \in V : visit_v = \text{false}$  do  
        RECTOPO( $v$ )
```

algorithm RECTOPO($v \in V$):

```
     $visit_v \leftarrow$  true  
    foreach  $(v, u) \in E \wedge \neg visit_u$  do  
        RECTOPO( $u$ )  
    orderk  $\leftarrow v$   
    k  $\leftarrow k - 1$ 
```

Possible order:

$\langle 211, 310, 321, 201, 304, 403, 317, 216, 311, 409 \rangle$