- an undirected graph is $G = (V, E)$, with V the vertices, E the edges. $V = \{V_1, V_2, ..., V_n\}$ is a set of objects, E a set of connections between the objects s.t. $\forall e \in E, e = \{u, v\}$ with $u, v \in V$.
- a directed graph differs in its edges E, where $\forall e \in E, e = (u, v)$ with $u, v \in V$.
- a subgraph of G is $H = (V_H, E_H)$ where $V_n \subseteq V, E_H \subseteq E$, and $(\forall e = (u, v) \in E)(u, v \in V_H)$
- connected component: maximal connected subgraph (can't connect more nodes)
- a cycle in $G = (V, E)$ is a sequence of nodes $\{v_k\} \in V$ s.t. $\{v_i, v_{i+1}\} \in E \wedge \{v_1, v_k\} \in E$
- a tree is an acyclic connected graph
- greedy algorithm:
  - local optimal decision to try solving a global problem
  - decisions are irrevocable (helpful for easy time complexity proofs)
- Minimum spanning tree (MST) problem: given an undirected connected graph G and nonnegative weights $d(e) = d(u, v)$, find the subgraph T that connects all vertices and minimizes

$$\text{cost}(T) = \sum_{\{u,v\} \in T} d(u, v)$$

  - think about why T must be a tree. if there are cycles, you can trivially rm an edge to cut costs
  - MST Cycle Property: let a cycle C be in G. the heaviest edge on the cycle C is not in G's MST.
  - MST Cut Property: let $S \subseteq V$, s.t. $1 \leq |S| < |V|$, i.e. S is not empty but not all nodes. Call a pair $(S, V \setminus S)$ a cut of the graph. Every MST of G contains the lightest edge on the cut (where an "edge on the cut" is some $e = \{u, v\}, u \in S, v \in V \setminus S$).
  - Prim's alg
    - description
      - given graph G, choose arb node s, the start of T
      - repeating $|V| - 1$ times,
        - add to T, the lowest edge (and nodes) from "in T" to "not in T"
    - proof
      - The pair ("in T", "not in T") is a cut of G. By the Cut Property, the MST contains the lowest cost edge crossing this cut, which is by defn the next edge Prim's adds, i.e. Prim's adds edges in the MST.

        At any point in the algorithm, $T = (V_T, E_T)$ is a subgraph and a tree. T grows by 1 vertex and 1 edge at each step, stopping at $|V_G| - 1$ steps, and so results in a spanning tree. Since these edges are in the MST from above, the spanning tree is the MST. $\square$
    - impl
      - (min) heap ADT
        - interface
          - `fn new(items) -> Self;`
          - `fn min() -> Item;`
          - `fn insert(item);`
          - `fn delete(item);`
        - heap-ordered-tree (as an array) implementation

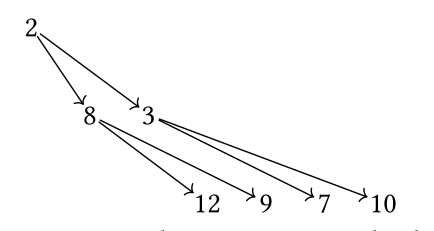

Figure 1: notice the monotonic decreasing upward. Also notice the array implementation, where traverse_left$(i) = 2i$ and traverse_right$(i) = 2i + 1$.

          - `new` $\in O(n)$; take items as array, for all elements (indices right to left), sift down
          - `min` $\in O(1)$; it's at the top of the tree
          - `insert` $\in O(\log n)$; insert as leaf preserving shape (aka dense array), sift up
          - `delete` $\in O(\log n)$; swap with leaf, delete, sift swapped down
      - work of Prim's with a heap is $O(m \log n)$
        1. $O(n)$; create empty heaparray (for max n items)
        2. $O(1)$; choose arb start node u
        3. $O(\log n)$ work per heap operation, $O(m)$ times total (because we only nbors the edges that much, trust. see your example); $\forall v \in$ `nbors`(u), if v is closer to T than our saved distance from v to T, update this distance (and also set parent of v to u). With a heap, do `del(v); insert(v, d(u, v))` or just `decreaseKey(v, d(u, v))`. TODO parent??
        4. $O(\log n)$; set u to closest node to T (min of heap) and delete it
        5. $O(n)$ repetitions; while u exists, go to (3)

        - this is $O(m \log n + n \log n)$ from (3) and repeating (4), which is $\in O(m \log n)$
  - Reverse Delete
    - description
      - start with G
      - repeating $|V| - 1$ times
        - remove edges by decreasing weight unless it'd split the graph
    - proof
      - tree bc we removed all cycles by defn of algorithm
      - spanning bc $|V| - 1$ thing and non-split clause
      - minimum: at some step, let e be the next edge removed. Since it's removed, it's in a cycle and must be the largest edge in it. By cycle property, all removed edges are not in the MST.
  - Kruskal's alg
    - description
      - start with $T = (V, \{\})$, i.e. all the nodes and no edges
      - repeating $|V| - 1$ times
        - add edges in increasing order unless it creates a cycle
    - proof
      - tree bc we didn't connect in a cycle... but why is output connected?
      - spanning bc $|V| - 1$ thing and tree
      - minimum: at some step, let e be the next edge added. All rejected edges would have created cycles, and must be the max in the cycle they create by algo defn. This means, by cycle property, it's MST.
    - impl
      - union-find abstract data type (ADT)
        - interface
          - `fn new(items) -> Self;` splits items into that many sets
          - `fn find(item) -> Label;` finds the set the item belongs to
          - `fn union(a, b);` merges two sets
        - array-based implementation

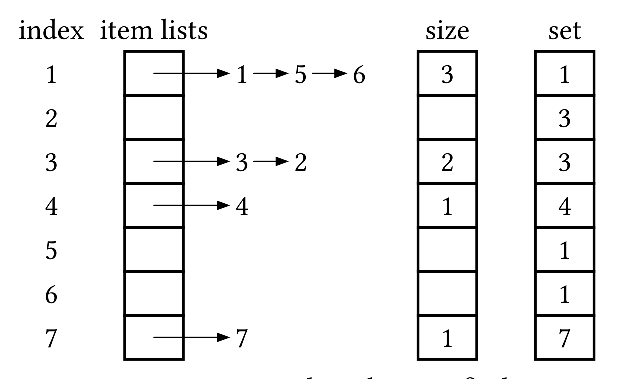

Figure 2: array-based union-find

          - `new` $\in O(n)$; creates the three lists
          - `find` $\in O(1)$; size array lookup
          - `union` $\in O(\log n)$ amortized;
            - procedure
              1. find smaller set $(x \leq y)$
              2. for each x_i, make `set[elem]` be y
              3. update sizes array
              4. prepend smaller to larger list
            - proof
              - (2) is computation. others $O(1)$
              - after k unions, $\leq 2k$ items touched
              - for any item v, `set[v]` is relabeled $\leq \log_2 2k$ times (TODO WTF WHY) ($2k$ is the # of items in the largest set????)
              - $2k \log_2 2k \in O(k \log k)$ work
        - tree-based implementation

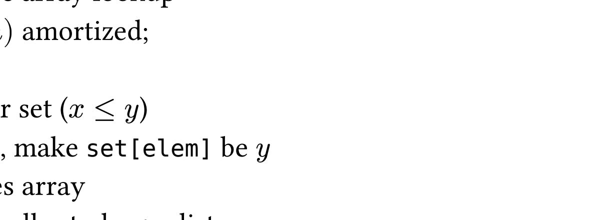

Figure 3: tree-based union-find.

          - `new` $\in O(n)$
            - let `struct Node { parent: Option<&Node>, height: uint }`
            - create an array of node addresses, so we can get from a node id to its node in $O(1)$
            - create all singleton nodes
          - `find` $\in O(\log n)$; follow ptrs to the root, which is the set label
            - proof
              - set at i is renamed at most $\log_2 n$ times because each renaming doubles size
              - depth of tree is number of renamings
          - `union` $\in O(1)$ (amortized?); move pointer, update height. is this really all we do?
      - work of Kruskal's for array-based union-find is $O(m \log n)$
        - sorting edges $\in O(m \log m) \in O(m \log n)$
        - because $m \leq n^2$, so $\log m \leq \log n^2 = 2 \log n$
        - at most $2m$ `find` operations $\in O(2m)$ (to check if edge would create cycle)
        - at most $n - 1$ `union` ops in $O(n \log n)$
        - total runtime $\in O(m \log n + 2m + n \log n) \in O(m \log n)$
      - work of Kruskal's for tree-based union-find is also $O(m \log n)$
        - sorting edges $\in O(m \log n)$ from above
        - at most $2m$ `find` ops $\in O(2m \log n)$
        - at most $n - 1$ union ops $\in O(n)$
        - total runtime $\in O(m \log n + 2m \log n + n) \in O(m \log n)$
- graph traversal problem: suppose a graph $G = (V, E)$. Find a path from a node s to t if it exists.
  - depth-first search

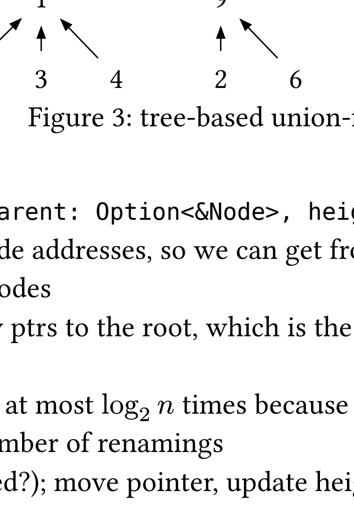

Figure 4: depth-first search

    - theorem: adjacent edges are not on the same level. That is, if $(x, y) \in E$, x is either an ancestor or descendant of y.
      - proof: WLOG, let x ancestor of y. When we pass x, we haven't seen y. All nodes between initially seeing x and leaving x are decendants of x. So, y must have been explored before leaving x, as a descendant.
  - breadth-first search

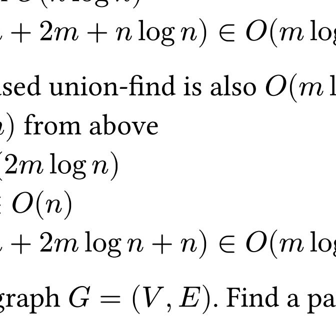

Figure 5: breadth-first search

    - theorem: adjacent edges are near the same level. That is, if $(x, y) \in E$, then $|\text{layer}(x) - \text{layer}(y)| \leq 1$
      - proof: 1) WLOG, AFSOC that $\text{layer}(x) < \text{layer}(y) - 1$. 2) All nbors of x are added in or before $\text{layer}(x) + 1$. By 2, $\text{layer}(y) \leq \text{layer}(x) + 1$. But by 1, $\text{layer}(y) > \text{layer}(x) + 1$. $\rightarrow\leftarrow$.
  - these (and Prim's!) are specifc types of tree-growing algorithms

```
TreeGrowingAlg(graph G, initial node v, fn findnext)
    T = ({v}, {})
    S = set of nodes adj to v
    while S has items,
        e = nextEdge(G, S)
        T = T + e
        S = updateFrontier(G, S, e)
```

    - `next`/`update` is pop/push for DFS and de/enqueue for BFS, $O(m)$ across entire program.
      - same logic as Prim's
    - everything else in $O(n)$ (including loop)
    - thus, DFS and BFS are $\in O(n + m)$
  - is-bipartite problem
    - definition: a graph is bipartite if $\exists$ a two-coloring. that is, if you can divide the nodes into two colors s.t. all edges connect nodes of diff colors.

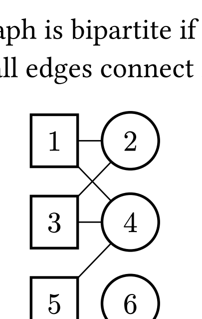

Figure 6: a bipartite graph

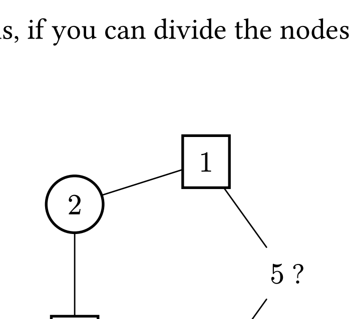

Figure 7: can't have odd cycles

    - solution: do a BFS from any node, swapping colors per layer. check if any edge is monolayer (since edges are between adj layers or same layer)
      - todo: prove correctness
- todo: topological sort with "exists node with no inc. edges" and "DFS finishing times"