

- Heap:**
- implemented as a complete binary tree in an array
  - for node at index  $i$ 
    - \* left child is at  $2i$
    - \* right child is at  $2i + 1$
    - \* parent is at  $\lfloor \frac{i}{2} \rfloor$  (floor)
  - insert: put at the last node and up-heap if needed.  $O(\log(n))$
  - remove min: replace the root node with the last node, and down-heap if needed.  $O(\log(n))$
  - up-heap: compare a node with it's parent. swap if needed. Repeat until you no longer need to swap, or at the root node
  - \* down-heap: same as up-heap, except going down. choice of left or right child is arbitrary.
  - merge two heaps (of equal height) with an extra element  $e$ : just connect  $e$  as the new root node, then down-heap if needed
  - bottom-up heap construction:
    - \* treat array as complete binary tree
    - \* down-heap starting at second-to-bottom row (bottom row is leaves, therefore satisfies heap property)
    - \* continue down-heap until you get up to the root node
    - \* not every node can be down-heaped the full  $h$ , because they started at different points
      - this is why it's  $O(n)$

- Priority Queue:**
- Comparison:

|                |              |              |
|----------------|--------------|--------------|
| implementation | insert       | removeMin    |
| unsorted list  | $O(1)$       | $O(n)$       |
| sorted list    | $O(n)$       | $O(1)$       |
| heap           | $O(\log(n))$ | $O(\log(n))$ |
  - priority queue sort: make the input into a priority queue (in-place) then remove each from the queue back into the list
    - \* complexity is  $O(n * \text{insert} + n * \text{removeMin})$
  - PQ sorts: selection, insertion, heap
    - \* selection: unordered list
    - \* insertion: ordered list
    - \* heap: heap (duh)

- Set:**
- ordered
  - union:  $A \cup B$ : all elements in either  $A$  or  $B$
  - intersection:  $A \cap B$ : all elements in both  $A$  and  $B$
  - subtraction (relative complement):  $A \setminus B$ : all elements in  $A$  and not in  $B$
  - can be easily implemented using a binary tree

- Quick Select:**
- algorithm to find the  $k$ th smallest element
  - algorithm that's the same as quicksort, except it only recurses onto the section containing the element we're looking for
  - recurrence relation:  $T(n) = O(n) + T(\frac{n}{2})$ 
    - \*  $O(n)$  for the partitioning step
    - \* this solves to  $O(2n)$  which is  $O(n)$

- Sorting:**
- slow sorts:**
- selection, insertion, bubble
  - all  $O(n^2)$  average
  - insertion and bubble are  $O(n)$  best-case
  - selection is  $O(n^2)$  always

- Heap Sort:**
- $O(n \log(n))$  all cases
  - in-place
  - can use the bottom-up heap building to be faster, but the removal stage still limits it to  $O(n \log(n))$
  - non-recursive

- Merge Sort:**
- $T(n) = 2T(\frac{n}{2}) + O(n)$ 
    - \* splitting is  $O(1)$ , merging is  $O(n)$
  - always  $O(n \log(n))$
  - not in-place

- recursive
- Quick Sort:**
- $T(n) = 2T(\frac{n}{2}) + O(n)$
  - recursive
  - pivot is randomly selected
  - $O(n \log(n))$  average and best,  $O(n^2)$  worst
    - \* worst-case: pick worst element every time. This makes it  $T(n) = T(1) + T(n + 1) + O(n)$ .
    - \* Unlikely for random numbers

- Graphs:**
- path: sequence alternating vertexes and edges. must begin and end on a vertex.
    - \* simple path has all vertexes and edges unique (does not cross itself)
  - cycle: a loop of nodes
    - \* simple if it doesn't intersect itself (except beginning/end)
    - \* non-simple otherwise
  - total in-degree of a graph is equal to it's total out degree
  - total degree of graph is double the number of edges
  - connected graph: there is a path between every pair of vertexes
  - tree: connected graph with no cycles
  - spanning tree: tree that includes all vertexes in graph
  - TODO expound this

|                   | Edge List  | Adjacency List            | Adjacency Matrix |
|-------------------|------------|---------------------------|------------------|
| space             | $O(n + m)$ | $O(n + m)$                | $O(n^2)$         |
| endVertexes()     | $O(1)$     | $O(1)$                    | $O(1)$           |
| opposite()        |            |                           |                  |
| incidentOn(v)     |            |                           |                  |
| v.incidentEdges() | $O(m)$     | $O(deg(v))$               | $O(n)$           |
| v.adjacentTo()    | $O(m)$     | $O(\min(deg(v), deg(w)))$ | $O(n)$           |
| insertEdge(u,v,w) | $O(1)$     | $O(1)$                    | $O(1)$           |
| eraseEdge(e)      |            |                           |                  |
| insertVertex(x)   | $O(1)$     | $O(1)$                    | $O(n^2)$         |
| eraseVertex(v)    | $O(m)$     | $O(deg(v))$               | $O(n^2)$         |

- adjacency matrix is usually a bad choice
- DFS: Depth First Search:**
- visits each child node before visiting adjacent nodes
  - can be used for maze traversal
    - \* each position is a vertex, edges are places you can get to
  - similar to preorder tree traversal
  - labels vertexes visited or not, labels edges as discovery or back edges
    - \* no cross edges because those would have been discovery edges
  - $O((m + n))$  for adjacency list?
  - DFS and BFS are both good for finding connected components

```
DFS(G: (V,E), s: starting v):
    v.label = VISITED
    for e in v.incidentEdges():
        if e.label == UNEXPLORED:
            w = e.opposite(v)
            if w.label = UNEXPLORED:
                e.label = DISCOVERY
                DFS(G, w)
            else:
                e.label = BACK
// then repeat for other connected components
```

- BFS: Breadth First Search:**
- visits all nodes in order of their distance from the starting node. (distance in hops, not counting edges)
  - labels vertexes to keep track of whether visited
  - labels edges as discovery or cross edges
    - \* no back edges because those would have been discovery edges (assuming undirected)
  - uses a (non-priority) queue to keep track of the edges to check next
  - $O((m + n))$  for adjacency list

BFS( $G: (V,E)$ ,  $s$ : starting  $v$ ):

```
Q.enqueue(s)
while !Q.empty():
    v = Q.dequeue()
    v.label = VISITED
    for e in v.incidentEdges():
        if e.label == UNEXPLORED:
            w = e.opposite(v)
            if w.label == UNEXPLORED:
                e.label = DISCOVERY
                w.label = VISITED
                Q.enqueue(w)
            else:
                e.label = CROSS
// then repeat for other connected components
```

### MST: Minimum Spanning Tree:

- minimum total weight spanning tree
- partition property: if you partition the MST into two subsets, there must be exactly one edge connecting the two, and it must be the minimum possible edge connecting the two subsets

### Prim-Jarnik's Algorithm:

- $O((m+n)\log(n))$  for adjacency list
- start at a given (or arbitrary) node as our MST
- put all the vertexes in a PQ keyed with their shortest edge connecting them to the MST
- add the closest vertex  $v$  to the MST
- update the vertexes adjacent to  $v$  with their new distance (use PQ.replaceKey())
- repeat until the PQ is empty

```
Prim_Jarnik(G: (V,E), s):
    for each v in V:
        D[v] = inf, P[v] = NULL
        PQ.add(v, key=D[v])
    while (!PQ.empty()): // O(n)
        u = PQ.removeMin()
        for e in u.edges(): // O(m)
            z = e.opposite(u)
            if e.weight() < D[z]:
                D[z] = e.weight(); P[z] = u
            PQ.replaceKey(z, D[z]) // O(log(n))
```

### Kruskal's Algorithm:

- $O((m+n)\log(n))$  for adjacency list
- initialize a PQ with all edges keyed by weight
- make clouds of mini MST's by adding each minimum edges
  - \* adding edge joining non-cloud vertex to cloud is easy
  - \* if an edge connects two vertexes in the same cloud, ignore it
  - \* if an edge connects two vertexes in different clouds, add it and merge the clouds (merging these clouds is complex depending on the implementation)
- does not start at any particular node
- cluster merging:  $merge(u,v)$  is  $O(\min(|C_u|, |C_v|))$ . We assume that merging doubles the size of the sets, therefore we preform max  $\log(n)$  merges
- complexity:
  - \* PQ:  $m$  removals:  $O(m\log(n))$ ; cluster merges:  $O(n\log(n))$
  - \* total:  $O((m+n)\log(n))$

```
kurskals(G: (V,E)):
    T = (V, NULL) // all vertexes, no edges
    for v in V: { define cluster C(v) = {v} }
    for e in E: { PQ.add(e, key=e.weight()) }
    while ( P.size() < V.size() - 1 ): // O(m)
        (u,v) = PQ.removeMin() // O(log(n))
        if C(u) != C(v): // nodes from different clusters
            T.insertEdge(u,v)
            MergeClusters(C(u), C(v) )
            // ~- O(min(|C(u)|, |C(v)|))
```

### SSSP: Single Source Shortest Path:

- SSSP is a spanning tree where the path from every node to the root is the shortest possible path.
  - \* not necessarily the same thing as the MST
- a subpath of a shortest path is itself a shortest path
- if there is no path between two vertexes, we generally represent the path length as  $\infty$
- **Dijkstra's Algorithm**
  - \*  $O((m+n)\log(n))$  for adjacency list
    - $(m+n)$  because it must look at every node
    - $\log(n)$  for the PQ operations

- \* assumes: graph is connected, all edges are undirected, all weights are  $\geq 0$
- \* is a greedy algorithm
- \* very similar to Prim-Jarnik's Algorithm, main difference is that we care about the distance to the root node, not the distance to the cloud
- \* store all distances in a map keyed with the vertex;  $\text{map}<\text{dist}, \text{vertex}> D[]$
- \* also store vertexes in a PQ, keyed with their total distance from the root (also stored in  $D[]$ )
- \* edge relaxation:

- for vertex  $v_0$  not yet in the cloud: check if edge  $(v_b, v_0)$  provides a shorter path than the current edge  $(v_a, v_0)$
- if  $D[v_b] + (v_b, v_0).weight < D[v_a] + (v_a, v_0).weight$ : use the new edge  $(v_b, v_0)$  to connect  $v_0$  to the cloud, and update  $D[v_0]$  accordingly
- \* does not work for negative edge weights because it is greedy, doesn't go back and check for the ways negative weighs could change things

```
dijkstras(G: (V,E) P: ParentMap, s: start vertex):
    D[v] = infinity for each v in V
    D[s] = 0
    P[s] = NULL
    Q = PQ of all v in V keyed with D[v]
    while !Q.empty(): // O(n)
        u = Q.removeMin() // O(log(n))
        for e in u.edges(): // relax each adj. edge
            z = e.opposite(u)
            if D[u] + e.weight() < D[z]:
                D[z] = D[u] + e.weight()
                Q.updateKey(z) // O(log(n))
                P[z] = u // update the parent of this node
```

### • Bellman-Ford Algorithm

- \*  $O(nm)$
- \* works for negative edge-weights (therefore must assume directed graph, otherwise there are negative weight cycles)
- \* doesn't work if there are negative weight cycles, but can be extended to detect them
- \* iteration  $i$  finds all shortest paths of length  $i$ , therefore the last iteration finds the maximum length shortest-path, of length  $|V| - 1$ 
  - detect negative weight cycles: relax again at the end. If any edge can be relaxed, there is a shortest path with length  $|V|$ , and therefore there must be a negative weight cycle

```
bellman_ford(G: (V,E), s: starting vertex, P):
    D[v] = infinity for each v in V
    D[s] = 0; P[s] = NULL
    for i = 1:(V.size() - 1): // O(n)
        for each e in E: // O(m)
            // relax edge
            u = e.source(); z = e.target()
            if D[u] + e.weight() < D[z]:
                D[z] = D[u] + e.weight()
                P[z] = u
```

### Directed Graphs:

- also just called digraph
- graph is **strongly connected** if every vertex can be reached from every other vertex
- strongly connected components: subsets of a graph which are themselves strongly connected.
- determine if graph  $G$  is strongly connected:
  - \* do DFS on  $G$ . if there are any nodes not visited, graph is not strongly connected.
  - \*  $G' = G$  with all directed edges reversed
  - \* do DFS on  $G'$ . if there are any nodes not visited, graph is not strongly connected.
  - \* otherwise, graph is strongly connected
- Directed Acyclic Graph: directed graph with no directed cycles
- DFS and BFS make sense on a digraph. MST doesn't really make as much sense on a digraph