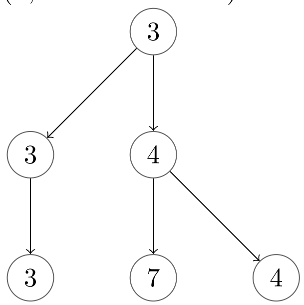
1 Quake Heaps

Quake heaps are a collection of **tournament trees**, where the value at the parent node is the minimum of tha values at child nodes. Each node only has 1 or 2 children, and we can have more than one tree side by side at the same time. $\forall$ node, $h(x) = dis(x,$descendant node$)$.



Since every node has at most two children, then the height of each level n_{i+1} will be $\leq \alpha n_i$, where $\alpha \in (1/2, 1)$. Basically, every level will have 1/2-1 times more nodes than the level directly downward. Its potential function is defined as $N+T+B/(2\alpha-1)$, n=nodeNum, T=treeNum, B= deg-1 nodeNum. **Insert**$O(1)$, **Dec-Key** $O(1)$,**Extract-Min** $O(\log n)$. When we Dec-Key, we remove the min path from the top of tree to leaves in $O(\log n)$, then, we merge all the trees with height i.

2 Disjoint Sets

A data structure that maintains collection of disjoint sets. With **Make-Set(x)**, we create a singleton set $\{x\}$. With **Union(b,d)**, we make a single set out of the two sets containing b and d respectively **Find-Set(x)** returns the **representative element** of set S_i. If DS doesn't change, then the RE will stay the same.

2.1 Linked Lists

We can use LLs to maintain the data structure. We maintain a pointer from avery element to the representative, so $O(1)$ Find-Set(x). To merge two sets, we would need $O(min(S1, S2))$. It takes $O(n \log n + m)$ to perform seq. of m Make-Set(x), Find-Set(x), Union(x,y), in which ther eare n Make-Set(x) operations, due to the fact that we can update the pointers at least $O(\log n)$.

2.2 Forest Implementation

With this approach, we implement all the sets as a tree with elements pointing at the head. Here, Union(x,y) takes $O(1)$ since we only update the root pointers when switching elements. Just compare the size of the trees, adn append the larger one to the smaller one. However, Find-Set(x) takes $O(h)$, where h is the height of the tree.

3 Graph Traversal

We can represent a graph by by using **adjacency list** or **adjacency matrix**. There are differences in how much time it takes to check if there is a connection, vs. how much space it takes to store a graph. The adjecency list might make it harder to iterate and delete nodes, since it would have TC $O(\deg(u))$

Algorithm 1: BFS(G,S)

visited[1..n] ={no};
dis[1..n] = {inf};
parent[1..n] = $\{NIL\}$;
EnQueue(Q,s);
while *Q≠0* **do**
  u = DeQueue(Q);
  for $v \in Adj[u]$ **do**
    if *visited[v]=no* **then**
      EnQueue(Q,v);
    end
  end
end

For this algorithm, all the nodes are enqueued at most once, since their visited field is changed when enqueued. Therefore, it takes $O(n+m)$

3.1 Applications

The **diameter** of a tree is the maximum distance between two nodes in the graph. If we run BFS(T,c) and record the node with largest d: this is o_1 or o_2. There are 4 different cases.

3.2 DFS

For DFS, we keep visiting any previously unvisited nodes in G as soon as we find them. Then, we start to work on the node itself. We mark them **white** if they are not discovered yet; **grey** if they have been discovered but are not finished; and **black** if they have been completely explored. However, it will only discover the nodes reachable from the source node.

Proper DFS algo will call DFS-Visit() for multiple starting nodes, usually those that might not be reachable from previous starting nodes.

Algorithm 2: DFS-Visit(G,u)

time = time + 1;
u.d = time;
u.color = gray;
for *all nodes* $v \in Adj[u]$ **do**
  if *v.color == White* **then**
    v.parent = u;
    DFS-Visit(G,v);
  end
end
u.color = Black;
time = time + 1;
u.f = time;

3.3 Parenthesis Theorem

For any two nodes u and v, exactly one of the following holds:

- The intervals [u.d, u.f] and [v.d, v.f] are disjoint neither is a descendant of the other.
- The interval [v.d, v.f] contains the interval [u.d, u.f] :v is an ancestor of u
- The interval [u.d,u.f] contains the interval[v.d,v.f]: u is an ancestor of v.

There are four different types of edges:

- **tree edges**: v discovered while v.color == white
- **back edges**: while exploring (u,v), v.color == Gray
- **forward edges**: while exploring (u,v), v.color == black and u.d ¡ v.d
- **cross edges**: while expl.(u,v), v.color == black and u.d¿v.d