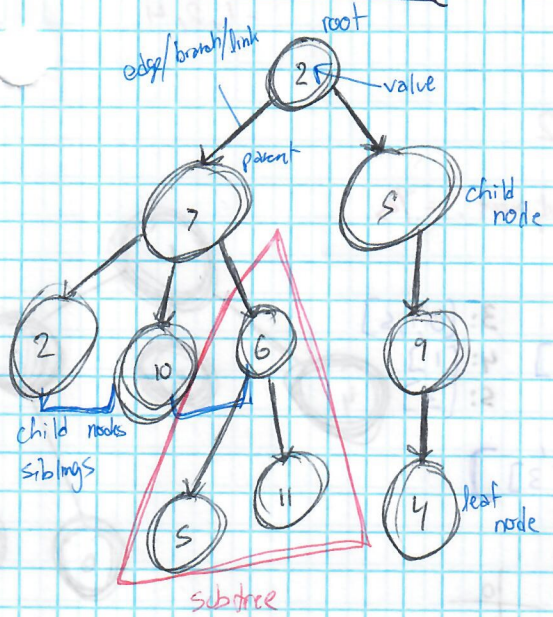
# Exam 2 Review:




* height = # of edges along longest path from root to leaf
  - ↳ deepest level in tree = index from zero

root = 2 (top node) — subtrees = 9

children of root = 2 — height = 3

# of nodes = 10

**private variables:** (getter methods)

getValue()

getChildren()

**private instances:**

value T (int, String, boolean, etc.)

children List<TreeNode<T>>

sumTree — take Tree Integers; return sum of all values within tree

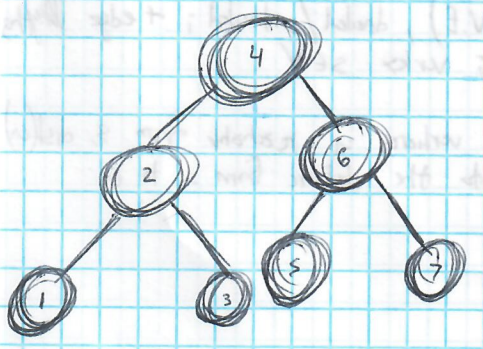
*height — take Tree; returns number of nodes within longest path

maxOfTree — take Tree of Integer; return maximum value of Tree (in)

```
if (tree.isLeaf()) { return tree.getValue; }
int maxSoFar = tree.getValue();
for (TreeNode<Integer> child : tree.getChildren()) {
    maxSoFar = Math.max(maxSoFar, maxOfTree(child));}
return maxSoFar;
```

Queues: First In, First Out (FIFO)

Stacks: Last In, First Out (LIFO)

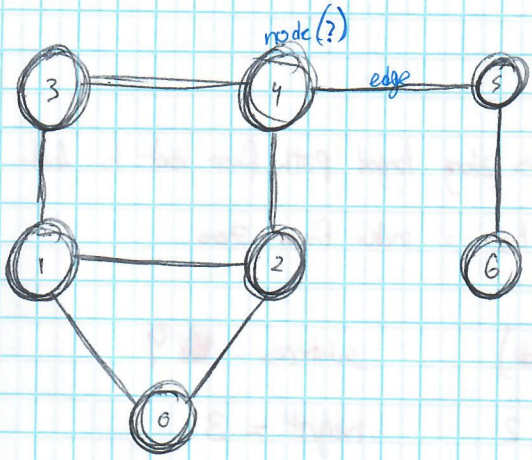



Pre-Order: 4 2 1 3 6 5 7 — top → bottom, left → right
↳ print node; print left; print right

In-Order: 1 2 3 4 5 6 7 — left → node → right
↳ print left child; print node; print right child

Post-Order: 1 3 2 5 7 6 4 — bottom → top, left → right
↳ print left; print right; print node

BST Add, Remove, Searching: Best = O(1) Worst = O(n) Average = (Random Tree) ↳ usually O(h)

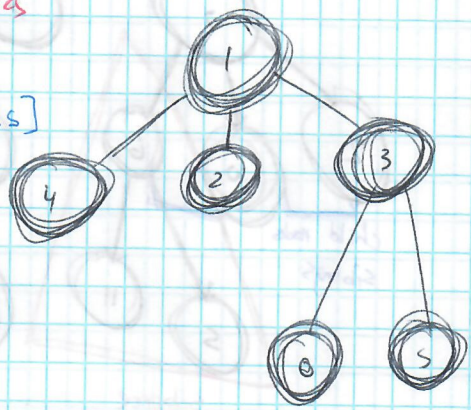



- vertices = 7
- edges = 8
- neighbors of 0 = 2
- verts 3 & 4 adjacent? = Yes
- most neighbors? = 1, 2, 4

List: 0: [3] 3: [0,1,5]
1: [2,3,4] 4: [1]
2: [1] 5: [3]
[ [3], [2,3,4], [1], [0,1,5], [1], [3] ]

Matrix:

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |




0 = Matrix: | 0 | / 0 | 0 | List: [ ]

DFS (Graph, node)
u node visited = true
For each node v in adj. graph[node u]
if visited = false
DFS(Graph, node v)

BFS - create queue Q
v = visited; put v in Q
while Q is not empty
remove head u of Q
mark & enqueue all v neighbors of u

- Undirected = no arrows
- Directed = yes arrows
- Cycle = yes loops
- Acyclic = no loops

[ - Source = no arrows pointing at node
- Sink = no arrows pointing out of node ]

Directed Acyclic Graph (DAG)

→ weighted graph = edges have values!

Priority Queue: add = O(log(n)) deletemin = O(log(n)) decrease key = O(log(n))

## Dijkstra Pseudocode:

for all node $u \in V$,
dist(u) = ∞, prev(u) = nil
dist(s) = 0

H = makeQueue (V) using dist-value keys
while H != empty
u = deletemin(H)
for all edges (u,v) ∈ E:
if dist(v) > dist(u) + l(u,v):
dist(v) = dist(u) + l(u,v)
prev(v) = u
decreasekey(H,v)

* pick the next unvisited node closest to s based on dist & process its neighbors

- Input: Graph G = (V,E), directed/undirected; + edge lengths $(l_e : (e \in E))$; vertex $s \in V$
- Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u