# STACKS, QUEUES, PRIORITY QUEUES (slides)

Stack

- LIFO: last in first out
- abstract ➝ push, pop
- java.util.Stack class ➝ adds and removes from end of ArrayList
    - push() ➝ add element to stack, O(1)
    - pop() ➝ get and remove last element in, O(1)
- could write your own with ListNodes
    - can write peek() ➝ return next value without removing

Queue

- FIFO: first in first out (fair)
- abstract ➝ enqueue (add element to queue), dequeue (remove first in element)
- java.util.Queue interface
    - add() ➝ at end, O(1)
    - remove() ➝ from front, O(1)
    - can implement with LinkedList
- could write your own with ListNodes
    - can write peek() ➝ same code as diy Stack

PriorityQueue

- keeps values in sorted order or by priority, smallest out first
    - objects must be Comparable or provide Comparator
- abstractly, is a sorted list

## BINARY SEARCH TREE (slides)

priority queue

- inefficient diy design
    - invariant → keep list sorted
    - remove() → always remove first and update myFirst, O(1)
    - add() → O(N) because have to search for insertion
    - OR keep list unsorted → peek/remove() O(N), add O(1)
- java API uses binary heap → balanced best of both worlds
    - peek() → O(1)
    - remove() → O(log(N))
    - add() → O(log(N))

binary heap at a high level

- sorted list of nodes → ordered binary tree of nodes
- heap property: every node is less than or equal to its successors
    - values are smaller at top, smallest is the root
- shape property: tree is full (each has 2 children) except rightmost positions on the last level

binary tree comparisons

- ArrayList → fast (access) not very dynamic (changes)
    - O(1) get but O(N) add/remove except at end
- LinkedList → dynamic, not very fast
    - O(N) get, O(1) add/remove (once you get there)
- binary search tree → fast and dynamic
    - O(log(N)) search and O(log(N)) add/remove, assuming tree is balanced
- TreeSet/Map
    - O(log(N)) add, contains, put, get (not amortized)

        - stored in sorted order, can get range of values in sorted order efficiency
    - HashSet/Map
        - O(1) add, contains, put, get (amortized)
        - unordered, cannot get range efficiently

binary tree nodes

- has value and right+left child nodes

terminology

- root: top node, has no parent, node you pass for the whole tree
- leaf: bottom nodes, have no children (both null)
- path: sequence of parent-child nodes
- subtree: nodes at and beneath
- depth: number of edges (lines) from root to node (can have different conventions)
- height: maximum depth

binary search tree if

- left subtree values are less than this nodes value, right subtree values are all greater
- enables efficient search like binary search

## TREE RECURSION (slides)

ways to recursively traverse a binary tree

- inOrder: left, root, right
- preOrder: root, left, right
- postOrder: left, right, root

ex: TreeCount

- base case → tree == null, return 0
- recursive case → 1 + count(t.left) + count(t.right)

complexity of tree traversal

- create recurrence relation and solve $(T(N) = a*T(g(N)) + f(N))$
- tree traversal → $O(n)$ regardless of balance
    - balanced → $T(n/2) + O(1) + T(n/2) = O(n)$
        - $T(n/2)$ because same number left and right
    - unbalanced → $T(0) + O(1) + T(n-1) = O(n)$
        - if every node has one child on one side
- search → balanced $O(\log(n))$, unbalanced $O(n)$
    - balanced → $T(n/2) + O(1) = O(\log(n))$
    - unbalanced → $T(n-1) + O(1) = O(n)$
- a binary tree (a,b) is approximately balanced if
    - for every node rooting a subtree of size n >= a, left and right subtrees have at most b(n/2) nodes
- approximately balanced tree is good enough for $O(\log(N))$

## GREEDY ALGORITHMS FOR DISCRETE OPTIMIZATION (slides)

optimization

- find solution that maximizes or minimizes an objective

greedily searching

- start with partial solution, take a step toward a complete solution in each iteration
- greedy principle: in each iteration, make the lowest cost or highest value step
- does not always guarantee the best overall solution (global optima)
- sometimes is optimal or not provably optimal but works in practice
- ex: machine learning → in a neural network, make a small change to best improve performance

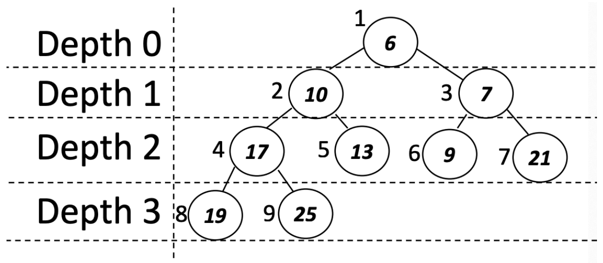
## BINARY HEAPS (slides)

binary heap

- java API PriorityQueue is implemented as a binary heap
- heap property: every node is less than or equal to its successors
    - values are smaller at top, smallest is the root
- shape property: tree is full (each has 2 children) except rightmost positions on the last level

using an array

- PriorityQueue actually implements with an array
    - minimizes storage (no nodes)
    - simpler to code (no explicit tree traversal)
    - faster (constant, not asymptotic) → children are located by index in array
- for a Heap
    - index positions into the tree level by level, left to right



1 indexed (not 0)

    - last node in heap is at the max index, min node is at index 1
    - peek returns first value
    - with 1 indexing, for node with index k
        - left child index = 2k
        - right child index = 2k+1
        - parent index = k/2
    - heap is complete → complete binary tree has $2^d$ nodes at depth d

heap add

- add to first open position at last level of tree (end of array)
- swap with parent if heap property violated → stop when parent smaller or root reached

heap remove

- returns root value
- replace root with last node in heap
- while heap property violated (while node is larger than current node), swap with smaller child

## BALANCED BINARY SEARCH TREES (slides)

binary search tree invariant

- binary tree that for every node ➝ left subtree values < node, right subtree values > node

diy TreeSet

- stores no duplicates in sorted order ➝ prints the tree values in-order
- add() ➝ wrapper for recursive helper insert()
    - if less, insert or recurse left; if greater, insert or reuse right; if equal, nothing (duplicate)
- contains() ➝ wrapper for recursive search
    - base cases ➝ no more tree to search, did not find target; found target
    - if target < node, search left; if target > node, search right
- runtime for add/contains on balanced tree
    - $T(N) = T(N/2) + O(1)$, $O(\log(N))$ ➝ same as binary search
- runtime for add/contains on (perfectly) unbalanced tree
    - $T(N) = T(N-1) + O(1)$ ➝ $O(N)$, search in linked list

runtime

- peek() ➝ return first value of array, $O(1)$
- add(), remove() ➝ traverse one root-leaf path, max is log N
    - complete binary tree always has height $O(\log(N))$
- runtime of add/contains is proportional to height of the binary search tree

red-black tree

- criteria (for binary search tree)
    - root is black, red node can’t have red children
    - all paths from a node to null nodes must have same number of black nodes
- TreeMap ks a red-black tree based in NavigableMap implementation
- TreeSet is a NavigableSet implementation based on a TreeMap

- not all binary search trees can be colored as a red-black trees → if not approximately balanced
- properties guarantee that approximate balance (good enough for O(log N))
    - any red-black tree with N nodes has height O(log N)
- functionality
    - contains/search → same as bst
    - add/insert → run regular bst add/insert, color new node red, recolor tree
        - sometimes won’t work → need to rotate (change structure, change root)

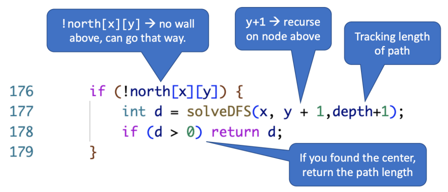
## GRAPHS, DFS, BFS (slides, slides)

graph

- data structure for representing connections among items, vertices connected by edges
- vertex (node) represents item, edge represents connection between 2 vertices
- undirected graph: edges go both ways (ex: webpage and links)
- directed graph: edges go one way (social media networks)
- simple graph: at most one (undirected) edge between nodes (or 2 directed)
- size of graph
    - N or |V| ➝ number of edges
    - M or |E| ➝ number of edges
    - in a simple graph, M <= $N^2$
- path: sequence of unique vertices connected by edges (or edges with unique vertices)

recursive depth-first search (DFS) in grid graphs

- 2d grid is a graph with implicit structure
- maze is a grid graph (2d with not all edges present)
    - edge = no wall, no edge = wall
- depth first search for solving maze
    - always explore (recurse on) a new (univisited) adjacent vertex if possible
    - if not, backtrack to the most recent vertex adjacent to an unvisited vertex and continue
    - base cases ➝ searching off the grid, already visited here (keep in array to avoid infinite)


!north[x][y] → no wall above, can go that way.
y+1 → recurse on node above
Tracking length of path
If you found the center, return the path length

```
if (!north[x][y]) {
    int d = solveDFS(x, y + 1,depth+1);
    if (d > 0) return d;
}
```

    - recursive case ➝ 3 more symmetric cases for other 3 directions
    - N width x height nodes ➝ runtime O(N)

        - each node will be recursed on <= 4 times (4 neighbors that can recurse on it)
        - each recursive call is O(1)
- vs recursive tree traversal ➝ tree traversal assume 2 adjacent nodes and no cycles

iterative graph search data structures

- have an adjacency list for the graph ➝ HashMap<Vertex, HashSet<Vertex>> aList
- keep track of visited nodes ➝ Set<Vertex> visited
- keep track of the previous node ➝ Map<Vertex, Vertex>

iterative DFS

- stack to store unexplored nodes (LIFO)
- keep going on one path until it can't anymore
- iterative DFS loop structure
    - while toExplore not empty, explore from the recently discovered node (current)
    - look at all the neighbors and check if unvisited (not in visited)
    - note how we got to this neighbor (put in previous), note we've seen this (add visited), mark to explore (push to toExplore)
- DFS search tree ➝ can find paths from one point to another by going backwards from the end point, the particular paths found in this graph
- DFS complexity (for N vertices, M edges)
    - pop each of N nodes at most once
    - loop over neighbors of each node exactly once, consider each edge twice (each relationship is recorded twice)
    - N+2M ➝ O(N+M)

iterative breadth-first search (BFS)

- queue to store unexplored nodes (FIFO)
- explore all your neighbors before visiting neighbors' neighbors
- iterative BFS loop structure
    - while toExplore not empty, explore from the closest discovered node (current)

    - look at all the neighbors and check if unvisited (not in visited)
    - note how we got to this neighbor (put in previous), note we’ve seen this (add visited), mark to explore (push to toExplore)
- BFS guarantees the shortest path, DFS does not ➝ BFS does not explore all paths, only shortest

## WEIGHTED GRAPHS, DIJKSTRA (slides)

weighted graphs

- each edge has an associated weight representing cost, distance, etc

Dijkstra data structures

- have an adjacency list for the graph → stores neighbors for each vertex (aList)
    - HashMap<Vertex, HashSet<Vertex>> aList → O(1) to get neighbors with good hashCode
- ~~keep track of visited nodes in a set (visited)~~
- keep track of the previous node → how did i get here (previous)
- priority queue to store unexplored nodes (toExplore)
- hashmap of distance of the shortest path so far, to a given node

Dijkstra's algorithm

- search loop structure
    - while toExplore not empty, explore from the closest unexplored node (current)
    - get weight on current to neighbor edge
    - for each neighbor, if no path found yet (!distance.containsKey(n)) or the path through current is shorter (distance.get(n) > distance.get(curr) + weight)
        - put this path (distance.get(curr) + weight) for neighbor in distance
        - note how we got to this neighbor (put in previous), note we've seen this (add visited), mark to explore (add to toExplore)
- might add the same node to the priority queue multiple times
    - adding neighbor to toExplore usually updates the priority of neighbor, not add
    - ^ use add because most standard library binary heaps don't have efficient update
- not guaranteed to be correct because greedy
- runtime complexity
    - consider each node (N) once, each edge (M) twice, log(N) for each → O((N+M)log(N))
- heap duplicates

- while loop may loop more than N times
- in graphs with constant degree (constant number of neighbors), will still just be O(N), but maybe not N
- worst case provable O((N+M)log(N)) needs an efficient priority update

## MINIMUM SPANNING TREE (MST), DISJOINT SETS, UNION FIND (slides, slides)

minimum spanning tree (MST) problem

- given N nodes and M edges, each with a weight/cost
- find a set of edges that connect all the nodes with minimum total cost ➝ tree

Prim's algorithm (greedy)

- initialize ➝ choose an arbitrary vertex
- parial solution ➝ MST connecting subset of vertices
- greedy step ➝ choose cheapest edge that connects a new vertex to the partial solution
- algorithm greedily grows by choosing closest unconnected vertex

Kruskal's algorithm (greedy)

- initialize ➝ all nodes in disjoint sets
- partial solution ➝ forest of spanning trees in disjoint sets
- greedy step ➝ choose cheapest edge that connects 2 disjoing sets/trees
- algorithm greedily grows by cheapest edge that connects disjoint sets/trees
- pseudocode (input of N nodes, M edges, M edge weights)
    - let MST to an empty set
    - let S be a collection of N disjoint sets, one per node
    - while S has more than 1 set:
        - let (u, v) be the minimum cost remaining edge
        - find which sets u and v are in; if not equal:
            - union the sets
            - add (u, v) to MST
    - return MST
- runtime ➝ O(M(log(M)+C)
    - while loop ➝ worst case, over all M edges

- removing from binary heap within loop → $O(\log(M))$
- C is time for union/find