COMPSCI 201 MIDTERM 3 REFERENCE SHEET

ArrayList

- get() = O(1), contains() = O(n), size() = O(1)
- add() = O(1) at end, O(n) at beginning
- remove() = O(N), have to shift all elements

HashSet

- unordered but efficient; internally creates a HashMap to store elements of set
- add(), remove(), contains(), size() = O(1)

HashMaps

- calculates hash (int) of a key with .hashCode() to see where to store values (diff keys can have same hash)
- put() adds to hash index bucket, updates if present = O(1)
- get() gets value at computed location to get hash bucket iterates over hashBucket keys to find .equals() key = O(1)
- containsKey() checks if location empty = O(1)

TreeSet/TreeMap

- ordered set/map, slightly less efficient than HashSet/HashMap but can easily get range
- add(), contains(), get() = O(log n))

LinkedList

- add() and remove() at front = O(1) → change pointers
- get() = O(N), loops through each element until null
- contains() = O(N), loops through list until you find it
- null pointer exc if you call method/variable on null node

Big O

- outer(inner(n)) → runtime of inner + runtime of outer on return value of inner

$O(2^N)$	Exponential	Calculate all subsets of a set
$O(N^3)$	Cubic	Multiply NxN matrices
$O(N^2)$	Quadratic	Loop over all *pairs* from N things
O(N log(N))	Nearly-linear	Sorting algorithms
O(N)	Linear	Loop over N things
O(log(N))	Logarithmic	Binary search a sorted list
O(1)	Constant	Addition, array access, etc.

recursive runtime formula → T(N) = a * T(g(N)) + f(N)

- T(N): runtime of method with input size N
- a: number of recursive calls (relative to N)
- g(N): how much input size decreases on each recursion
- f(N): runtime of non-recursive code on N

Recurrence	Algorithm	Solution
`T(n) = T(n/2) + O(1)`	binary search	`O(log n)`
`T(n) = T(n-1) + O(1)`	sequential search	`O(n)`
`T(n) = 2T(n/2) + O(1)`	tree traversal	`O(n)`
`T(n) = T(n/2) + O(n)`	qsort partition ,find k^{th}	`O(n)`
`T(n) = 2T(n/2) + O(n)`	mergesort, quicksort	`O(n log n)`
`T(n) = T(n-1) + O(n)`	selection or bubble sort	$O(n^2)$

binary heap - ordered binary tree of nodes

- heap property: every node <= 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
- peek = O(1), remove = O(log N), add = O(log N)

binary tree 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
- height: maximum depth

binary search tree

- left subtree values are less than this nodes value, right subtree values are all greater → efficient search
- tree traversal → O(n) regardless of balance
- search → approx balanced O(log(n)), unbalanced 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

tree recursion

- inOrder (l this r), preOrder (this l r), postOrder (l r this)

Stack - LIFO (can be implemented with ArrayList, add on end)

- push() = O(1), pop() = O(1), search = O(N)

Queue - FIFO (can be implemented with LinkedList)

- add() at end = O(1), remove() from front = O(1)

greedy algorithm

- optimization: find solution that maximizes or minimizes an objective
- greedy principle: in each iteration, make the lowest cost or highest value step
- no guarantee best overall solution (global optima)

PriorityQueue - keeps values in sorted order, smallest first

- in theory is binary heap, actually array for memory
- binary heap → peek() = O(1), remove() = add() = O(log(N))
- array indices store nodes L-R on each depth
- with 1 indexing, for node with index k
  - left child index = 2k
  - right child index = 2k+1
  - parent index = k/2
- add → add to heap at end of array, swap with parent if heap property violated, stop when parent is smaller or root is reached
- remove → returns root value, replace root with last node, swap with smaller child while heap property violated (node > current)

red black trees → TreeMap, TreeSet in java

- root is black
- red node can't have red children
- from root, all paths from a node to null nodes must have same number of black nodes
- satisfies approximate balance, height = O(log N)
- not all binary search trees can be colored as rb
- contains/search → same as bst
- add/insert → run regular bst add/insert, color new node red, recolor tree (sometimes won't work and need to rotate to change root)
- not more efficient due to maintenance

graph

- data structure for representing connections among items, vertices connected by edges
- vertex: item, edge: connection between 2 vertices
- undirected: edges go both ways, directed: go both ways
- simple graph: at most one directed edge between nodes
- path: sequence of unique vertices connected by edges

recursive DFS

- check if exceeding boundary, check if node already visited
- if neither then recurse in order of directions in all directions
- return if goal found for each direction, or recurse other directions

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 stores unexplored nodes; go down one path fully

- 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)
- for N vertices M edges: N+2M → O(N+M)

iterative BFS → queue stores unexplored nodes; check all your neighbors

- BFS guarantees the shortest path, DFS does not
- BFS does not explore all paths, only shortest
- N+2M → O(N+M)

weighted graphs

- each edge has an associated weight representing cost/distance

Dijkstra → priorityqueue (priority = distance of shortest path so far)

- 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)
- O((N+M)log(N))