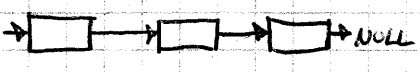
Analysis algorithm/datastructure
memory/CPU/operations $O(n)$ Notation

Queues FIFO add-to-rear / take-from-front enqueue/dequeue $O(1)$
array-based (circular array) / linked-list-based

Stack LIFO add/take-from-top array-based/linked-list-based
⇒resize array when full, doubling-strategy

Singly-Linked-List 

⇒Sequence ADT
⇒List

Doubly-Linked-List 

Vectors extended concept of array, resizes as needed

Trees traversal: pre-; in-; post-order root, nodes, leaves
depth→node, height→tree subtrees

- Binary Tree: 2 children per node, ordered (left/right)
- Heap: binary tree, internal nodes store keys, $Key(n) \geq Key(parent(n))$
  also called Heap-Order, complete binary tree, log(n)

Insertion⇒Upheap: restore order, swap val upward till root or n> val

Removal⇒ replace root with last node, make last node leaf
Downheap: restore order, swap with child and repeat

Max-Heap: ≥, large num at root, Min-Heap: ≤, smallest num at root

- BST (Binary Search Tree): binary tree where $Key(l) \leq Key(p) \leq Key(r$
  log(n) find/search, needs to be balanced for efficiency
- (2,4)-Tree: Multi-Way Search Tree, internal nodes store (d-1) ele
  where d is the num. of children, ordered as BST, split/fusion,
- AVL-Tree: balanced BST, height of all subtrees may only differ by -1,

~~[illegible]~~
if root.bal ≤ -2: (if R.bal<0: (RL(root); if ==-1: RL(R);) elif ==1: RR(R); RL(root);)

TreeNode rotations: RIGHT(r):

```
p = r→left;
r→left = p→right;
p→right = r;
reset parent to p;
```

LEFT(r):

```
p = r→right;
r→right = p→left;
p→left = r;
reset parent to p;
```

if root.bal: analog, but inverting directions, signs and

Sorting:
- Heap-Sort: $O(n\log(n))$ use a priority queue implemented as heap, add all you want to sort, remove min() (the root, max(
- Quick-Sort: $O(n\log(n))$ expected, $O(n^2)$ wc divide-and-conquer, divide by picking a pivot, putting everything less before, bigger after, and repeat/recurse on the left and right partitions, at the end; join back together
- Merge-Sort: $O(n\log(n))$ divide-and-conquer, divide until small and easy to sort, then join the pieces back, merging them
- Bucket-Sort: $O(n)$ if "spread" of elements known, we just prepare a sorted list of buckets and throw in the elements, then take them out again (counting-sort similar for integ
- Radix-Sort: $O(n)$ lexicographic-sort using bucket-sort

Priority-Queues: queue where elements have a priority, ordered by priority (maintaining FIFO order too), heap OK, usually remove Mi

Hash-Tables: array, map into it with hash-function, $O(1)$ lookup if good, random hash-function, either closed-addressing ⇒ chains (Linear Probing / Double Hashing) of max size, then double array, or open-addressing ⇒ re-hash or try

Dictionary: lookup of elements, BST (balanced) or HT or Skip-List

Graphs: Vertices, Edges; directed/undirected; degree, path, cycle

Digraph: directed graph, DFS/BFS, topological sorting

DFS (Depth-first-Search)

BFS (Breadth-first-Search):

Shortest Paths: Dijkstra (not-negative-weights), Bellman-Ford (also negative), DAG-based, All-Pair SP

Minimum Spanning Tree: Prim-Jarnik, Kruskal