Data Structures

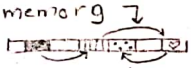
Basics
- Memory ↓ (bytes)
  - ↳ boolean (1)
  - ↳ char (2)
  - ↳ int (4)
  - ↳ float (4)
  - ↳ long (8)
  - ↳ double (8)
  - ↳ pixel (3)
  - ↳ arrays → (type)*n + 24
  - ↳ [][] → (type)*m*n + 24
  - ↳ obj overhead → 16
  - ↳ referance → 8
  - ↳ padding

Run Time
- tilde (~) biggest
  - ↳ 5log n + 33 → 5log n
- big O worst case
  - ↳ 5log n + 33 → O(log n)
- $1+2+\ldots+n = \frac{n}{2}(n+1)$
- $2^0+2^1+\ldots+2^k = 2^{k+1}-1$

Stacks (LIFO)
- pop (@.length - 1)
- push (@.length)

Queue (FIFO)
- enqueue (@.length)
- dequeue (@ 0 index)
  - ↳ resizing arrays
    - ↳ full? 2x size
    - ↳ ¼ full? ½ x size

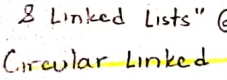
Linked Lists
memory ↓



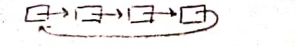
visualized ↓

- memory = (address + ref + item)n + overhead
- no limit on space
- no need to resize
- no random access
- extra 8 bytes for ref
- big O on "Stacks, Queues, & Linked Lists" @ bot

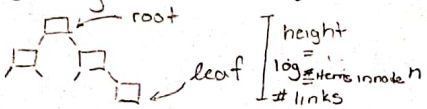
Circular Linked List
- last.next = first



Double Linked List



Binary Search Trees



- binary search trees
  - ↳ max 2 children / node
  - ↳ left is smalle, right is greater
  - ↳ if median is root → optimal
  - ↳ O(log n) search avg
  - ↳ inorder (left, root, right)
    - ↳ pre- (root, l, r)
    - ↳ post - (l, r, root)
    - ↳ lvl - ⇒ top down, left → right
- hibbard deletion (see "binary search")
- 2-3 tree

  - ↳ ≤ 2 items in node & ≤ 3 children
  - ↳ will always be balanced b/c insertion sys
- Red Black BST
  - ↳ 2-3 tree w/ internal nodes marked as "red"
  - ↳ every path root → null has same # black links
  - ↳ For insertion (left leaning)
    - ① child & gchild red → rotate right
    - ② right child red → rotate left
    - ③ both childs red → color flip

    (see "Binary Search")
- B trees
  - ↳ general 2-3 tree w/ M-1 key pairs
  - ↳ stores m-1 items in 1 node
  - ↳ ≥ 2 key link pairs in root
  - ↳ ≥ $\frac{m}{2}$ key link pairs in other nodes
  - ↳ usefull to store large amounts of comparable items

Priority Queue
- queue where items are given "priority" & higher priority is moved to the front
- implemented w/ binary heaps often

Big O →	order array	unorder array	binary heap
insert	O(n)	O(1)	O(log n)
delMin	O(1)	O(n)	O(log n)
findMin	O(1)	O(n)	O(1)

Binary Heap
- complete binary tree (all lvls full except last) & order invariant
- parent @ k → children @ 2k & 2k+1
- array implementation (leave 0 empty)
- insert @ .length & swim
- delete Min / Max swap w/ .length - 1 & delete & sink new root

for min:
- ↳ swim - if [k] < [k/2] swap, keep moving up
- ↳ sink - if [k] > min(2k, 2k+1) swap w/ smallest child, keep moving down (children ↓)

"priority queue" gives max Queue code.

- heapify (k) bottom up

```
if (x is leaf) return x
l = heapify (x.left)
if can
    r = heapify (x.right)
    if (r < x && r < l)
        // r smallest
        swap r & x
if (l < x)
    swap l & x
return x
```

- ↳ uses sinking

Hash Tables
- uses a hash function to sort items into an array
- index = hashcode % table size
- 2 ways to avoid chaining

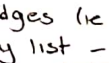
① Seperate chaining
- collisons are added @ head of a linked list
- array of M linked lists
- resizing (arbitrarily will be 10)
  - $\frac{N}{M} > 8$ → 2x
  - $\frac{N}{M} < 2$ → ½x
- resize, rehash all

② linear probing
- array of size M, M ≥ N
- collisions are added in the next open index [(# % 10) + 1]
- loop back to start if end reached w/o insert
- same resizing as ①
- avg # probes = $\frac{\text{probs / key}}{\text{\# probs}}$ (α = % table full)
- avg search hit $\sim \frac{1}{2}(1+\frac{1}{1+\alpha})$
- avg miss $\sim \frac{1}{2}(1+(\frac{1}{1+\alpha})^2)$

- load factor = #keys ÷ table size
  - ↳ % of table used
  - ↳ ideally < 70%

Graphs
- ∅ - V^2 # of edges
- 3 implements

① adjacency matrix - matrix of boolean values (from row → col?)
- ↳ space inefficient



② set of edges (ie {(1,2), (2,1), (3,2)}

③ adjacency list - represents edges
- O(E) space →

- Depth First Search (DFS) O(V+E)

```
mark v as visited
for (all adjacent to v)
    if (unmarked)
        dfs (adjacent vertex)
        edgeTo v → adj vertex
```

recursive

OR

```
push v onto stack S
mark v as visited
while (S is not empty)
    w = S.pop
    for (all adj to w)
        if (adj not visited)
            S.push (adj)
            mark adj as visited
```

iterative & stack

↳ can be used to find connected components (see bot of "graphs")

- Breadth First Search (BFS) O(V+E)

```
push v onto queue
mark v as visited
while (queue is not empty)
    w = dequeue
    for (all adj to w)
        if (adj unvisited)
            enqueue (adj)
            mark adj as visited
```

↳ often used to find shortest path btwn nodes
↳ can also find distance (dist = prev dist + 1)
(see "graphs" @ above directed Graphs)

- directed graphs have a max $\frac{n(n-1)}{2}$ edges
- adj matrix col gives in or out degree based on set up
- directed graph max n! paths
- Preorder DFS → print into queue b4 dfs call on encounter
- Postorder DFS → go to end & print when back tracking
  - ↳ into queue after dfs call
- Topological Sort O(n+e)
  - ↳ use DFS
  - ↳ assign numbers when back tracking, starting w/ n-1, then n-2, ... 0
  - ↳ all arrows "point up"
- reverse post order - into stack after dfs call
- Strong Connectivity O(E+V)
  - ↳ path u → v & v ← u
  - ① Find reverse post order of G reverse
  - ② Run DFS on G visiting unmarked nodes in the reverse post order

Sorting
- selection sort $O(n^2)$ starting @ 0 exchange w/ min/max (move 1 value)
- insertion sort $O(n^2)$ moving right, move smallest value left
  - ↳ left side will always be sorted
- merge sort O(n log n) keep dividing array in half until size = 1, then merge smaller arrays until back to 1 array
- quick sort O(n log n)
  - ① shuffle array
  - ② partition
  - ③ repeat
- heap sort O(n log n)
  - ① build heap w/ array
  - ② x = delMin/max
  - ③ insert x into array in order