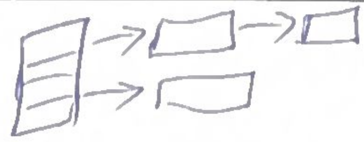

①

Hash Tables

key → hash(key) → mapping in array (hash(key) % len) → linked list

- $O(1)$ access ; $O(N)$ worst case
- can also be implemented with balanced BSTs. $O(\log n)$ lookup. uses less space. in-order traversal is useful for frequent range queries and ordered data access.

ArrayLists

- automatically resizable. $O(1)$ ~~lookup~~ access. $O(1)$ insertions $O(N)$ in worst case (doubling)

StringBuilder (concatenate list of strings)

- say size x strings and n strings.
  $x + 2x + 3x \ldots\ldots nx = x(1+2+\cdots n) = x\frac{n(n+1)}{2} = O(xn^2)$
  on each concatenation, a new copy of string is created and ~~one~~ two strings are copied over e.g. sentence = sentence + w
- StringBuilder can avoid this problem by creating resizable array of all strings.

Linked List

- add/remove items from beginning in constant time
- in implementation, we wrap a Node class inside a LinkedList class so that if the head changes, all objects referencing it know the update.
- in implementation, ① check for Null pointer and ② head and/or tail pointer as necessary.
- "The Runner Technique" (second pointer)
  - fast node might be ahead by a fixed amount or it might be hopping multiple nodes for each one node that slow node hops through

- Recursive Problems: a number of linked list problems involve recursion. recursive algos take at least $O(n)$ space where n is the depth of recursive call. All recursive algos can be implemented "iteratively".

Stacks

- LIFO ordering.
- pop(remove top item), push(add item to the top), peek(return the top of the stack), isEmpty().
- constant-time adds and removes but no constant time access to ith item.
- can be implemented like a linked list
- $\boxed{\text{Python}}$: list with append(push) and pop operations $O(N)$
  collection deque with append(push) and pop operations $O(1)$
  queue LifoQueue with put(push) and get(pop) ops $O(1)$
- useful in recursive algos. E.g. when you need to push temp data onto stack as a recurse but then remove them as you backtrack (bc recursive check failed).
- stack can also be used to implement recursive alogorithms iteratively.

Queue

- FIFO ordering
- add(add to the end), remove(remove first item), peek, isEmpty
- easy to mess up the updating of first and last nodes in a queue.
- often used in BFS and implementing a cache.

BFS ↓
① store list of nodes we need to process.
② each time we process a node, we add its adjacent nodes to the back of the queue.

Big O

→ time and space complexity

→ expected and worst cases of a routine are important.

→ big O just describes the rate of increase. $O(N)$ can run faster than $O(1)$. therefore, we drop the constants and non-dominant terms

→ $O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2)$ $< O(2^N) < O(N!) < O(N^N) < O(2^N * N!)$

→ add runtimes when there're no nested loops. if there are, multiply runtimes

→ Amortized time when worst case is not frequent e.g. insertion in ArrayList is $O(1)$

→ log N: # of elements in problem space get halved. base of log doesn't matter.
$N, N/2, N/4, \dots 1 \Rightarrow 2^K = N \Rightarrow K = \log N$

→ recursive runtimes: fibonacci → twice # of calls at each level → $1 + 2 + 4 + 8 + \dots 2^N$ → $2^{N+1} - 1$ → $\log(2^N)$

$O(N)$ space complexity since $O(N)$ nodes exist at any time

• in general, for recursive calls: $O(\text{branches}^{\text{depth}})$

• Recursive Problems: a number of ... algos take at least $O(n)$ space

→ give special attention to variables especially when there's more than 1. Don't use variable "N".

→ input of array of strings, sort the strings (s) and then sort the array (a).

$$O(a*s(\log a + \log s))$$

→ just bc there's a binary tree, doesn't mean there's a log in it. use the $O(\text{branches}^{\text{depth}})$ approach or think how many nodes we touch.

~~sometimes we have to check if~~

→ prime ~~factors~~ number check $O(\sqrt{n})$, compute factorial $O(n)$, fibonacci $O(2^n)$

→ generally, an algo with multiple recursive calls is exponential