Data Structures

		Array/Vector	L. List	(Deque)
insert / remove	front	$\Theta(n)$	$\Theta(1)$	Am. $\Theta(1)$
	"middle"	$\Theta(n)$	$\Theta(n)$ → $\Theta(1)$ if have a ptr	$\Theta(n)$
	back	Am. $\Theta(1)$	$\Theta(1)$	Am. $\Theta(1)$
search		$O(n)$	$O(n)$	$O(n)$

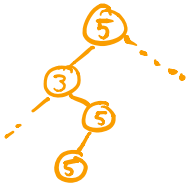
Binary Heap

(STL)	Hash Table (unordered map/set)	Trie	BST	AVL Trees	Red-Black Trees (map/set)
insert	Am. $\Theta(1)$, $O(n)$	$\Theta(L)$	$O(n)$	$\Theta(\log n)$	
remove	Am. $\Theta(1)$, $O(n)$	$O(L)$	$O(n)$	$\Theta(\log n)$	
search	Am. $\Theta(1)$, $O(n)$	$O(L)$	$O(n)$	$O(\log n)$	

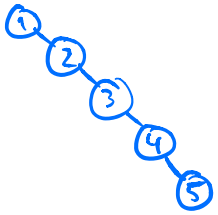
Avg. good hash table / hash fn

$O(n)$ ← worst case

L = length of key



1, 2, 3, 4, 5

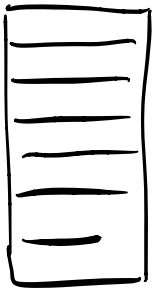


T/F In a BST, worst-case time complexity of insert, remove, search is $\Theta(h)$ where h = height of tree

→ In the worst case, $h = n$.

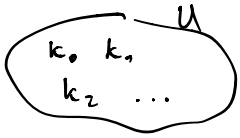
Hashing

Array of size m, store n elements



find **slot / bucket** using a hash fn

① Use hash fn to map keys to integers (hash codes)



② Compress hash code to $\{0, 1, 2, \ldots, m\}$

Simple way: hashCode mod m

Problem: collisions

Can occur when ① create hash codes
② compression
(③ during collision resolution)

Ex.: m = 10

hashCode = s[0] ← BAD

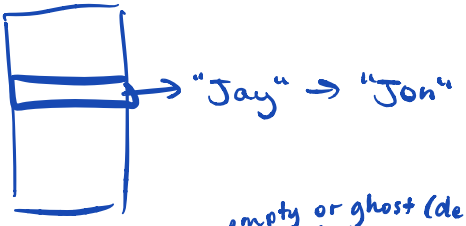
compress with h % m

s_1 = "Jay", s_2 = "Jon"

Collision resolution

① Open addressing (probing)

② Separate chaining



empty or ghost (deleted)

Linear probing: go to (next slot mod m) until available OR stop if full

Quadratic probing: $h(k) + p^2$ — $h(k) \xrightarrow{\text{taken}} h(k)+1 \to h(k)+4 \to h(k)+9$ (p = probe index)

Double hashing

$h_1(k) + p \cdot h_2(k)$

stop after m iteration
good probing will visit each slot only once
1A
1B
1C

Removing elements

1. Separate chaining → easy (remove from linked list) → constant if I have a ptr otherwise $O(\alpha)$
2. Open addressing
   replace with DELETED (aka 👻)
   → insert: treat DEL/GHOST same as EMPTY
   → search/find: treat same as TAKEN
   remove

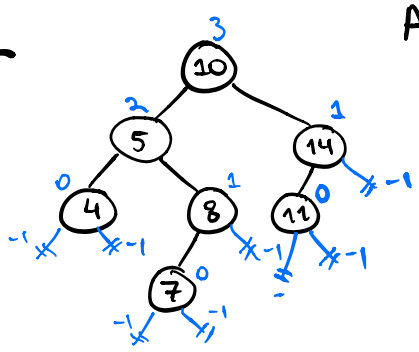
Load factor

$\alpha = \frac{n}{m}$ In sep. chaining $O(1+\alpha)$

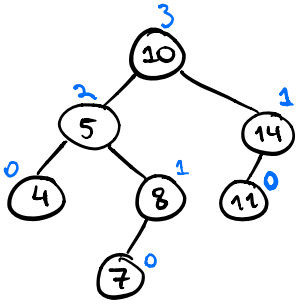
Dynamic hashing

if $\alpha > \alpha_{max}$ → resize/grow array and rehash
α_{max}: 0.7

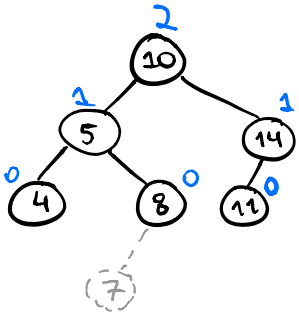
AVL Trees



AVL-tree property
heights of children can differ by at most 1.

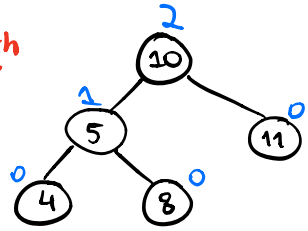


- Remove 7

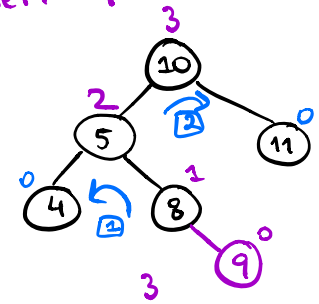


REMOVE operation
- remove node
  - CASE 0: no children ⇒ easy!
  - CASE 1: 1 child → child takes the node's place
  - CASE 2: 2 children ↪ replace node with in-order successor
- walk up the tree
  - update height
  - rebalance if necessary

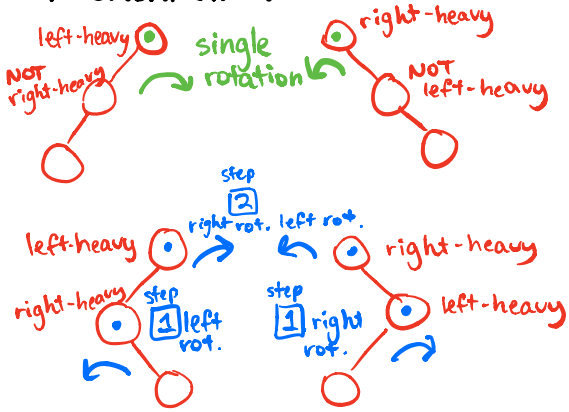
- Remove 14

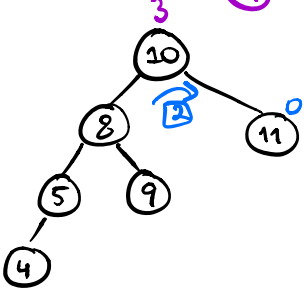


- Insert 9

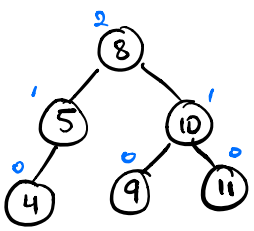


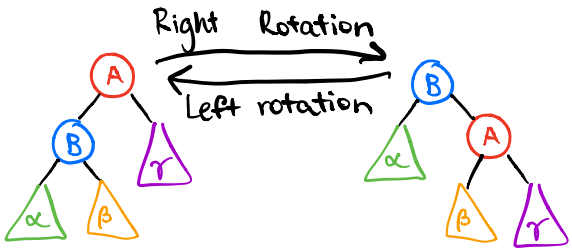
REBALANCING



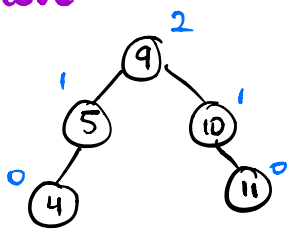


ANSWER

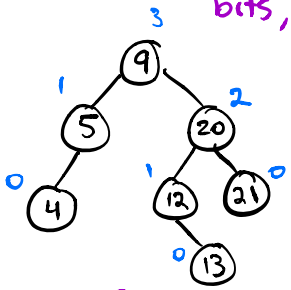




- Remove 8

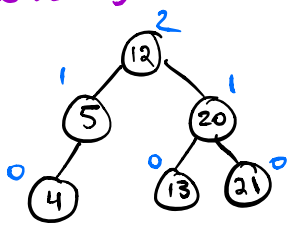


- Particles got attacked, mixed up bits, tree changed



2-minute break

- Remove 9



Tries

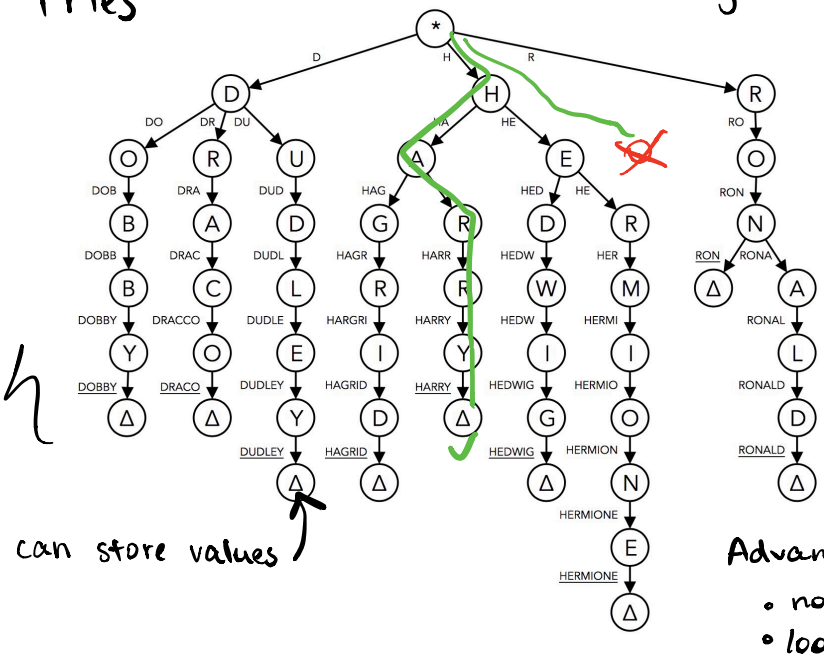
keys must have a string-like structure

"MAXIM"

2017

"characters" come from a fixed Alphabet

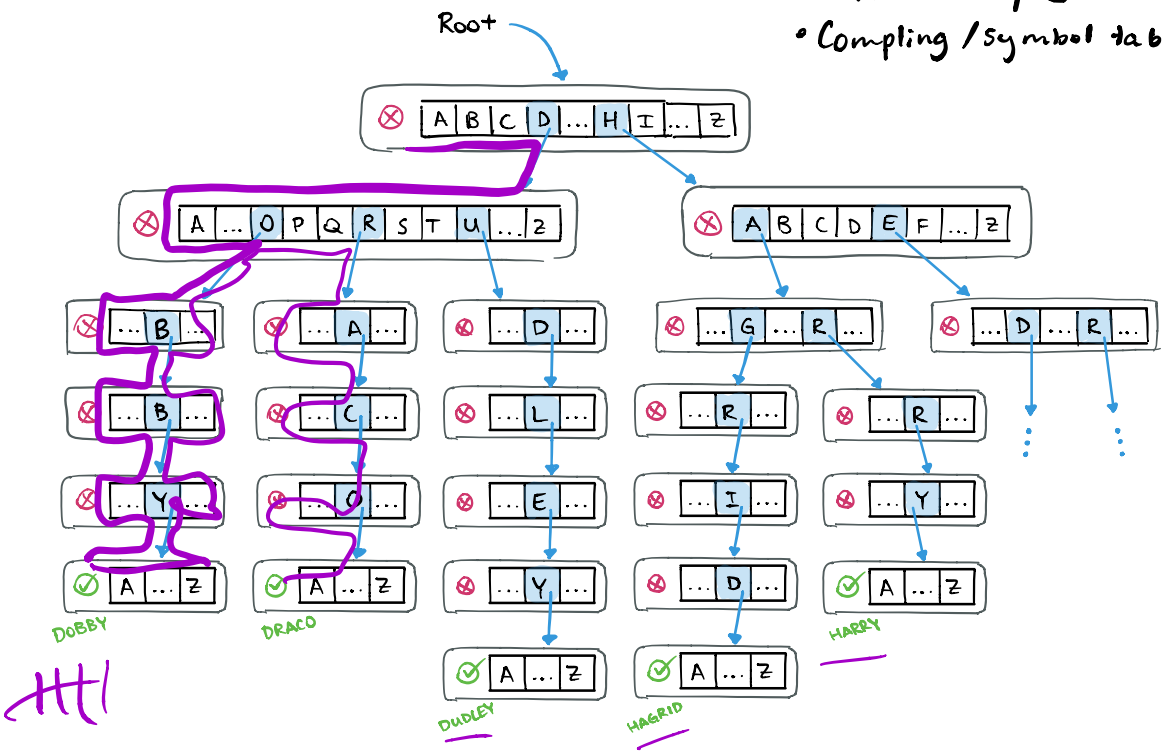
A-Z,
a-z A-Z
0-9
0-9 A-F



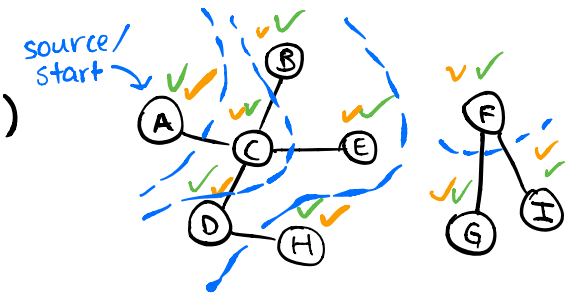
can store values

Advantages:

- no collisions
- look up by prefix
  - Google
  - Autocomple
- Compling / Symbol table



Graphs



DFS (pre-order depth-first search) A C B D H E F G I

visit node
mark visited
dfs on all unvisited neighbors

Post-order depth-first search B H D E C A G I F

mark visited
dfs on all unvisited neighbors
visit node

Breadth-first search A C B D E H F G I

visit all nodes on level 0
level 1
level 2

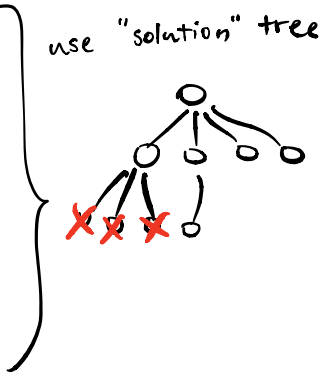
Algorithms

- **Brute-force:** min in unsorted array
  every room in every building to find your class
  guess passcode on someone's phone

- **Divide and conquer:** count ppl in room
  0) stand up
  1) think of #1
  2) find someone else
     2a) add up #s
     2b) one sit down
         other go to step 2

- **Greedy:** counting change
  Dijkstra's, Prim's, Kruskal's

- **Backtracking:**
  → a solution
  → all solutions
  N-Queens, maze, Coloring map, Sudoku



- **Branch-and-bound:**
  → optimal solution
  maximize benefit
  minimize cost
  Knapsack
  TSP (P4)

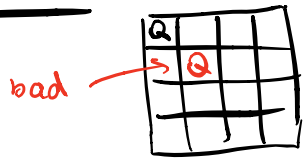
- **Dynamic programming**
  solving subproblems that have optimal substructure
  Often uses memoization (array or hash table) to remember solutions to subproblems

  Top-down or **bottom-up**

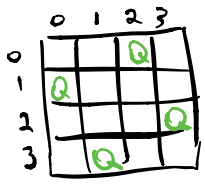
recursively	iteratively
solve only "required" subproblems	solve all subproblems

N-Queens

N by N chessboard (N is 4, 8, 16, ...)



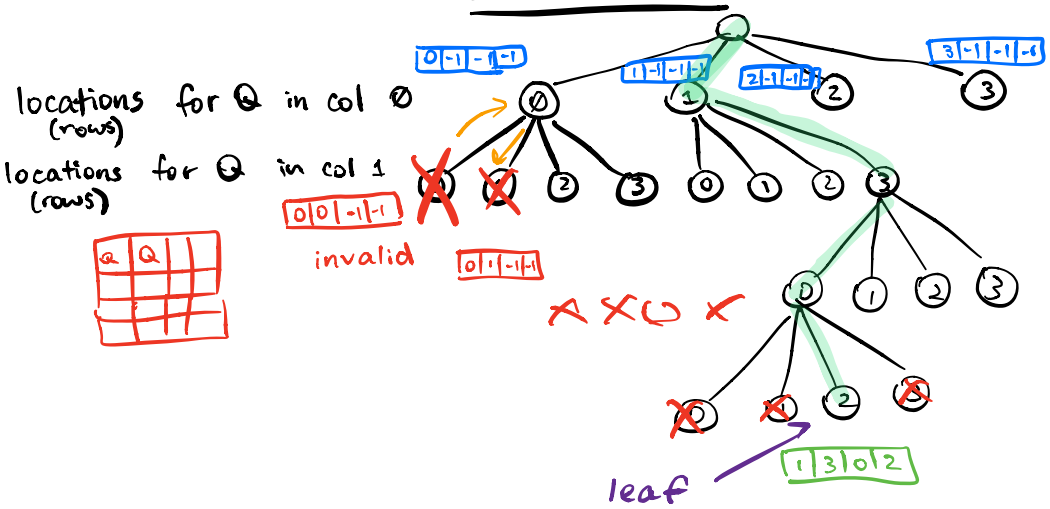
Place N Queens so that they can't attack each other.



Representation

Array of size N

Solution Tree



Knapsack Problem:

	0	1	2	3
weights	2 lb	1 lb	5 lb	6 lb
values	$10	$2	$7	$4

W=8 lb

Q Want to know max value

Subproblem Max value starting at item i with remaining size of backpack X — $K(i, X)$

Need to solve

$$K(0, W) \rightarrow \max\begin{pmatrix} \text{choose 0} \rightarrow & \text{values}[0] + K(1, W - \text{weights}[0]) \\ \text{!choose 0} \rightarrow & K(1, W) \end{pmatrix}$$

Basecase

$K(N, X) = 0$ $\quad K(4, X) = 0$

General

$$K(i, X) = \max\begin{pmatrix} \text{choose i} \rightarrow & \text{values}[i] + K(i+1, X - \text{weights}[i]) \\ \text{!choose i} \rightarrow & K(i+1, X) \end{pmatrix}$$

only if $\text{weights}[i] \leq X$

Approach to DP

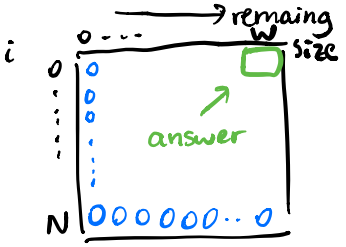
- Know how to represent solution
- Break into subproblems
- Find recursive def. + base cases
- Top-down or bottom-up
  - recursive
  - compute ans when unknown
  - memo
  - table for intermediate answers
  - answer in one of the corners

Knapsack

Total value (int)

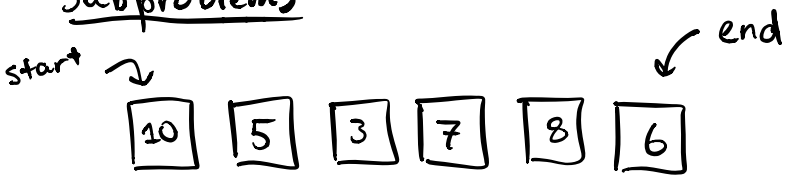
$K(i, X)$

$K(i, X) = \max(\_, \_)$



CARD GAME

- Want to know highest value you can get by playing agains opponent who uses greedy strategy
- Subproblems



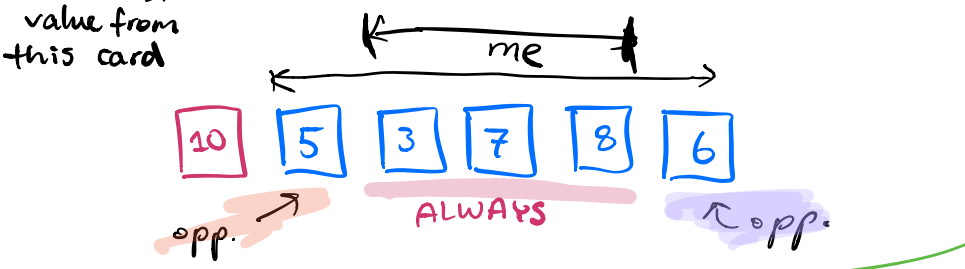
G(start, end) : what subarray to look at

- Actual problem
  G(0, N-1)
- Recursive def.

G(s, e) = max(choosing Left Card, choosing Right Card)

cards[s] + min(G(s+2, e), G(s+1, e-1))

value from this card



cards[e] + min(G(s+1, e-1), G(s, e-2))