Week 3 - 21

DATA STRUCTURES

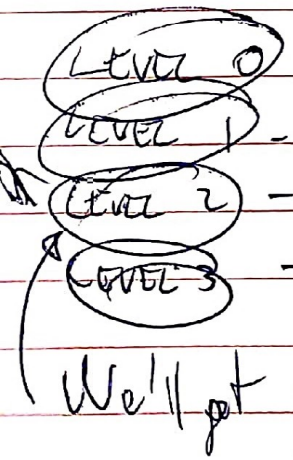
Point: organize data so it can be accessed quickly and usefully

Examples: lists, stacks, queues, heaps, search trees, hashtables, bloom filters, union find etc.

Why so many?
They all support different sets of operations with different times.
So they are suitable for different kinds of tasks.

Rule of Thumb: choose the "minimal" data structure that supports all the operations that you need.

TAKING IT TO THE NEXT LEVEL



- What's a data structure?
- Cocktail party knowledge, head of stuff, can't use in tech interviews
- "This problem calls for a heap"
- "I only use data structures I wrote myself"

Week 3 L2

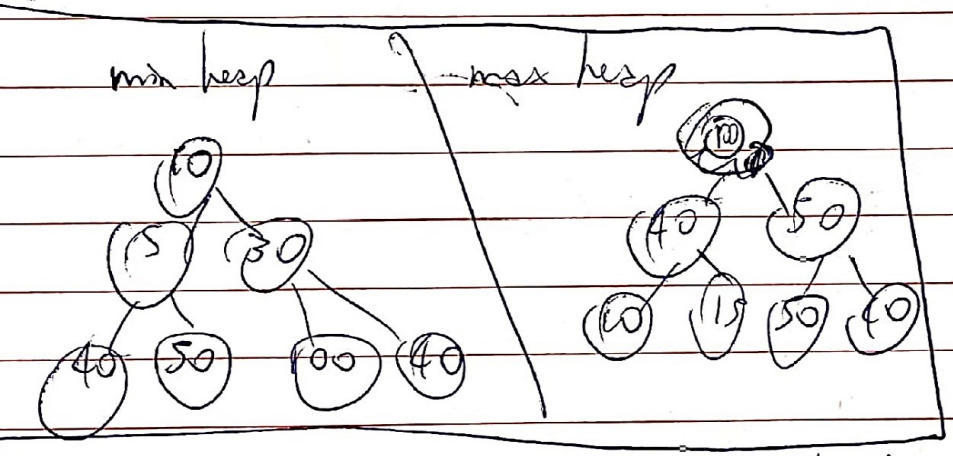
Heap HEAP: SUPPORTED OPERATIONS

- a container for objects that have keys
- employee records, network edges, events etc

INSERT: add object to a heap Runtime $O(\log n)$

EXTRACT-MIN (or MAX): remove an object with a minimum key value (ties broken arbitrarily) Runtime $O(\log n)$

quite good contents, 6/2



MAKEHEAPIFY: (n batched inserts in $O(n)$ time).

You don't need to create an empty heap and put n objects in. n times $\log n$ time needed for insertion $O(n \log n)$, We can do it in a slick $O(n)$ way

DELETE an arbitrary element from middle of a heap in $O(\log n)$

~~Coolest uses of heaps~~

CANONICAL USES OF HEAP

Canonical use: fast way to do repeated minimum computations.

Example: Selection Sort = $\Theta(n)$ linear scans, $\Theta(n)$ time on array of length n

Heap Sort:

① Insert all n array elements into a heap
② Extract-Min to pluck out elements in sorted order

Running time = $2n$ heap operations = $O(n \log n)$ time

Can we do better? NO. $n \log n$ is lower bound for any sort

ANOTHER APPLICATION: EVENT MANAGER.

"Priority queue" — synonym for a heap

Imagine videogame simulation:

- objects = event records, event-triggering events added with time stamp key
- Extract-min → next scheduled event.

APPLICATION: MEDIAN MAINTENANCE

I give you an intensely amount of numbers, one by one.
I want the median each time step:
Need to do it in $\log n$ (-) time

WEEK 3 L4

BALANCED SEARCH TREES

Sorted Array: supported operations

3	6	10	11	17	23	32	36

OPERATIONS	RUNNING TIME
SEARCH (bin search)	$O(\log n)$
SELECT in statistic	$O(1)$
MIN / MAX	$O(1)$
PREDECESSOR / SUCCESSOR	$O(1)$
RANK (rank of 23 is 6)	$O(\log n)$
OUTPUT IN SORTED ORDER	$O(1)$ n times, $O(n)$
INSERTION	$O(n)$
DELETION	$O(n)$

We want insertion and deletion

That's a lot for insertion and deletion unless you barely ever do them

That's why we make:

BALANCED SEARCH TREES (BSTs)

Recipe of the: sorted array + fast (logarithmic) inserts and deletes

OPERATIONS	RUNNING TIME
SEARCH	$O(\log n)$
SELECT	$O(\log n)$
MIN / MAX	$O(\log n)$
PRED / SUCC	$O(\log n)$
RANK	$O(\log n)$
OUTPUT IN SORTED	$O(n)$

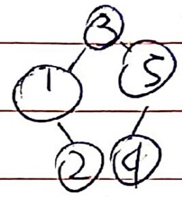
Up from $O(1)$

SAME FOR HEAP or MIN/MAX $O(1)$ adorable

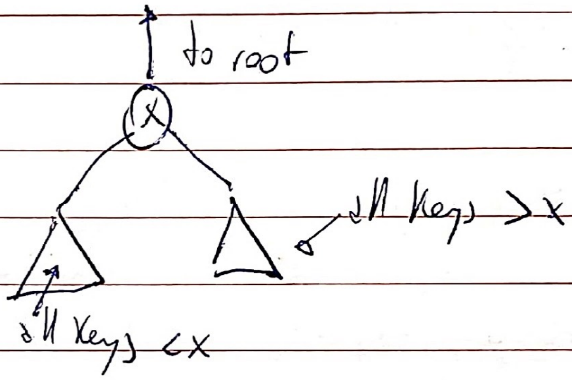
WEEK 3 L5

~~Raison d'être: like sorted array + fast logarithmic~~

BSTs



- exactly one node per key
- most basic version:
  each node has:
  - left child pointer
  - right child pointer
  - parent pointer

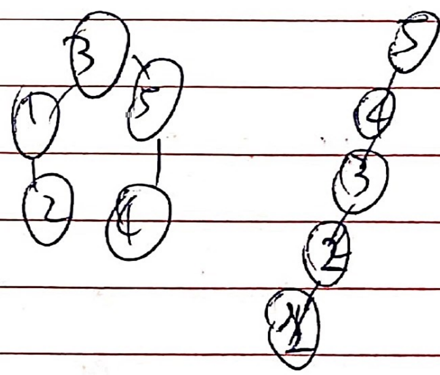


HEIGHT OF BST

best case, perfectly balanced — worst case: chain

Roughly $\log_2(n)$ usually, up to n

Many possible trees for a set of keys:



SEARCH FOR KEY IN TREE T

- start at the root
- traverse left/right child pointers as needed
- return node with key k or NULL as appropriate

TO INSERT

- search for k (unsuccessfully)
- put new node with key k at the end of pointer

Week 5

Search or Insert operations in a binary tree take at most Θ(height)

MIN, MAX, PRED AND SUCC

To compute min: start at root, you follow left pointer until the end Θ(height) again

PREDECESSOR OF KEY K

- easy case: if left subtree to K, non empty, return K's max in left subtree
- otherwise: if left subtree empty, follow parent pointers until you get to a key less than K

PRINT KEYS IN INCREASING ORDER

- Let r = root of search tree with subtrees T_L and T_R
- recurse on T_L
- print r
- recurse on T_r

$O(1)$ time per call, n recursive calls. $O(n)$

DELETE A KEY FROM A SEARCH TREE

- SEARCH FOR KEY

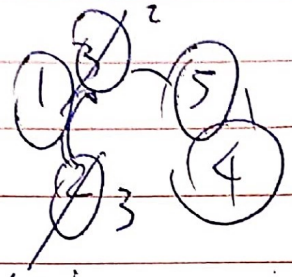
Then: three cases

(easy) no children. Just delete

(medium) splice out's K's node. Its one child takes its p

DIFFICULT CASE (x's node has 2 children)

- Compute K's predecessor & until you get to top, then swap top



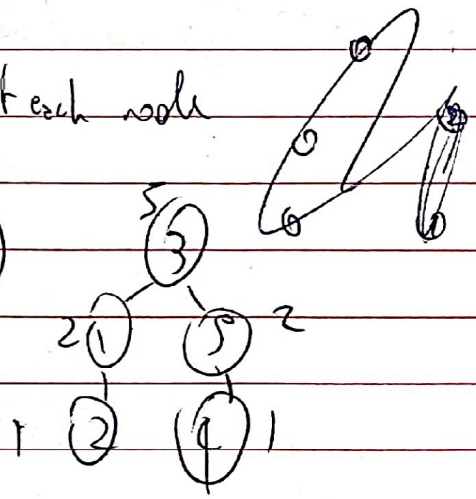
Then delete 3.

The running time $\Theta(\text{height})$

SELECT AND RANK

Idea: Store a little bit of extra info at each tree node.
about the tree itself

Example: store # of nodes in subtree rooted at each node



Note: if x has children y and z, $(\text{size}(y) + \text{size}(z) + 1)$

Also: Easy to keep sizes up to date in $\Theta(\text{height})$

TO SELECT:

- Start at root x, with children y and z
- let $a = \text{size}(y)$ [$a = 0$ if x has no left child]
- if $a = i-1$ return x's key
- if $a \geq i$ recursively compute i^{th} on root y
- if $a < i$ recurse $(i-a-1)^{th}$ on z

WEEK 3 L7

Idea: Ensure height always $\log(n)$

Example: Red-Black Trees

RED BLACK INVARIANTS

1. Each node red or black
2. Root is black
3. No 2 reds in a row
4. Every root-NULL path has the same number of black nodes

~~EXAMPLE~~

NO CHAIN CAN BE AN RB TREE

HEIGHT GUARANTEE:

$$\text{height} \leq 2\log_2(n+1)$$