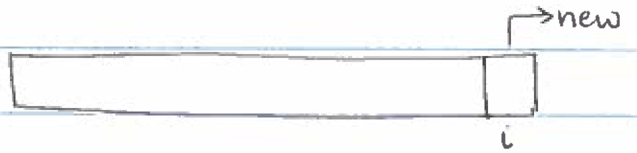
1/25/18

HEAPS

- Claim: If start with valid heap, increase any node i and ru
  decrease
  Heapify-Down(i), then get valid heap.
  Heapify-Up(i)
  $\underbrace{}_{\Theta(\log n) \text{ time}}$

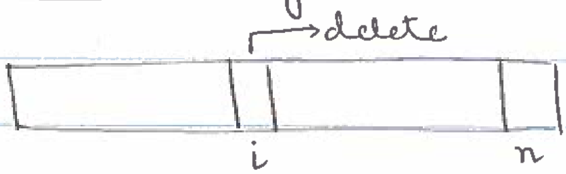
→ Insertion: $\Theta(\log n)$

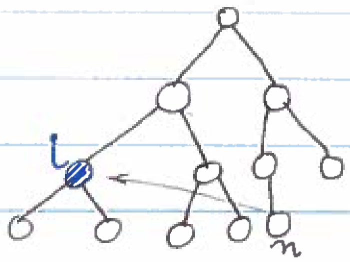
Say a new element is tobe added → add to end of array and call Heapify-Up (i) → Taken $\Theta(\log n)$ time



- Can think there was ∞ there before and we decreased it

→ Deletion: $\Theta(\log n)$



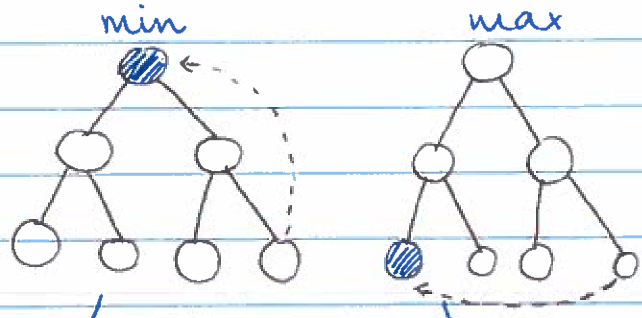


- Easy to delete and insert at the end of the array.
- what we do is: Override value of i with value of n.
  value(i) = value(n)

If value (i) is larger than before, call Heapify-Down(i)
If value(i) is smaller than before, call Heapify-Up(i)
delete n

? → Data structure such that we find min and max in $\Theta(1)$ time, insert/delete in $\Theta(\log n)$.

A



In addition to value, store index in max-heap

with value, store index in min-heap.

↳ If not stored, can't know where element to delete is.

→ <u>Naive Way to Build a Heap</u>:

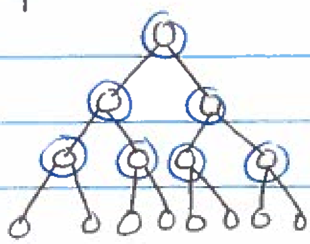
- Build a heap from existing array A.
  - ↳ Repeatedly call insert: $\Theta(n \log n)$
  - ↳ Can do better! $\Theta(n)$

- For simplicity assume:
  $|A| = 2^k - 1$ for some $k \geq 1$, i.e., will be a full binary tree.

- Build-Heap(A): ← array in arbitrary order

  for $i = \left\lfloor \frac{|A|}{2} \right\rfloor$ to 1

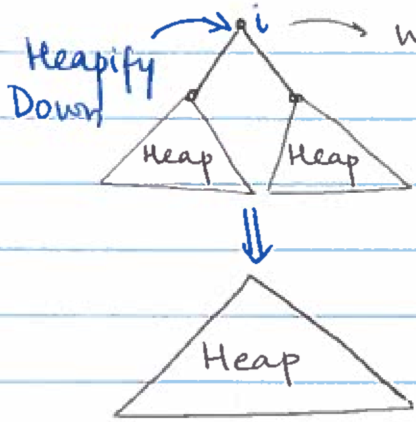
  Heapify-Down (A, i)



? Why backward order?

? Why Heapify-Down?

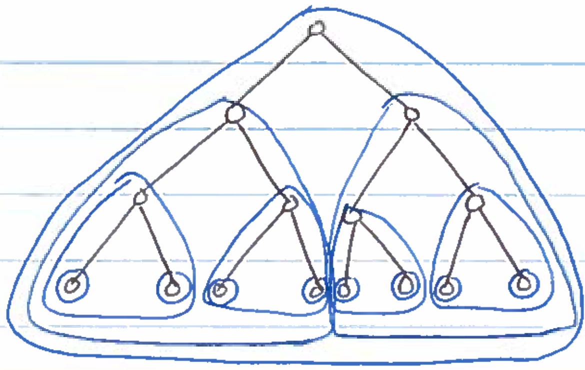
→ Say we have a subtree,



→ whole thing might or might no be a heap.

- Why Heapify-Down?
Assume at i, it was $-\infty$
now we increase the val

Similarly, we have a base case that each leaf is a hea



Recursive

∴ We go backward & use Heapify-Down.

? Why is this $\Theta(n)$?

- Seems to be $\underbrace{n}_{\text{Nodes looked at}}\underbrace{\log n}_{\text{Heapify-Down}}$

→ Total Work (# of swaps that might be needed)

	# nodes · # swaps
$l=0$	$1 \cdot 3$
$l=1$	$2 \cdot 2$
$l=2$	$4 \cdot 1$
$l=3$	$8 \cdot 0$

Generally, for layer l work done is

$$(h-l)\,2^l$$

$(h-l)$ → height

$\therefore$ Total work for all layers $= \sum_{l=0}^{h} 2^l (h-l)$

Let ~~~~. $j = h-l$,

$$\sum_{l=0}^{h} 2^l (h-l) = \sum_{j=h}^{j=0} 2^{h-j} \cdot j = 2^h \sum_{j=0}^{h} j/2^j$$

$$\le 2^h (2) = 2^{h+1}$$

$$\sum_{j=0}^{\infty} j/2^j = 2$$

For a full-binary tree, $n = \sum_{l=0}^{h} 2^l = 2^{h+1} - 1$

$\therefore$ Total work of all layers $\le 2^{h+1} = n+1$ $\Theta(n)$

⇓

$\boxed{\Theta(n)}$

GRAPH ALGORITHMS

→ DFS, BFS, Topological Sort

⇒ Representation of Graphs:

→ Adjacency Matrix – Good to find neighbours in constant time; Directed Graphs not symmetric Ma

→ Adjacency Lists – Array of vertices in graph and each array element has a linked list. (need not be sorted linked list)

To find neighbours, worst case need to traverse entire linked list.

- Check if $i \to j$?

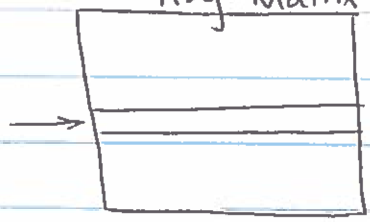
  Adj. Matrix: $\Theta(1)$

  Adj. Lists: $\Theta(m)$ → # of edges.

  (n = # of node
  m = # of edg

- We prefer adjacency lists because it is easier to make a walk on a tree with them (discussed later in BFS, DFS

Adj. Matrix



To go to next adjacent node need to scan whole row & look out for a 1.

In adj lists, list has this info.

⇒ DEPTH FIRST SEARCH (DFS):

– Find all nodes reachable from u (BFS also solves this)

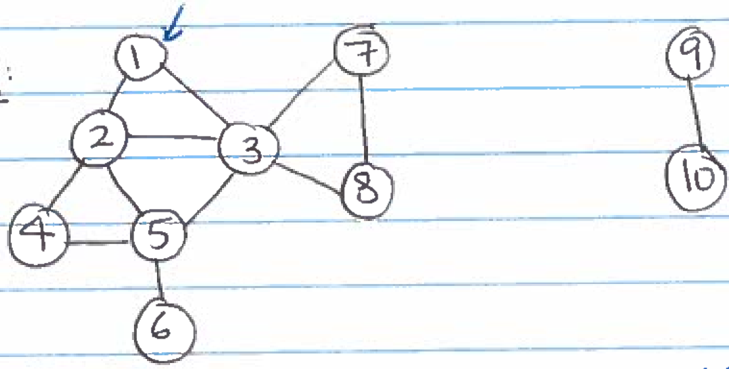
- BFS and DFS are for both directed and undirected graphs.
  ↓
  we look at this in class.

⟹ DFS(u) :

```
    mark u as explored
    add node u to T
    for each edge(u,v):
        if v is not explored
            add edge (u,v) to T
            DFS(v)
```

Return all explored nodes

G:



(Say adj lists have smaller value first)

DFS Tree : Starting at 1