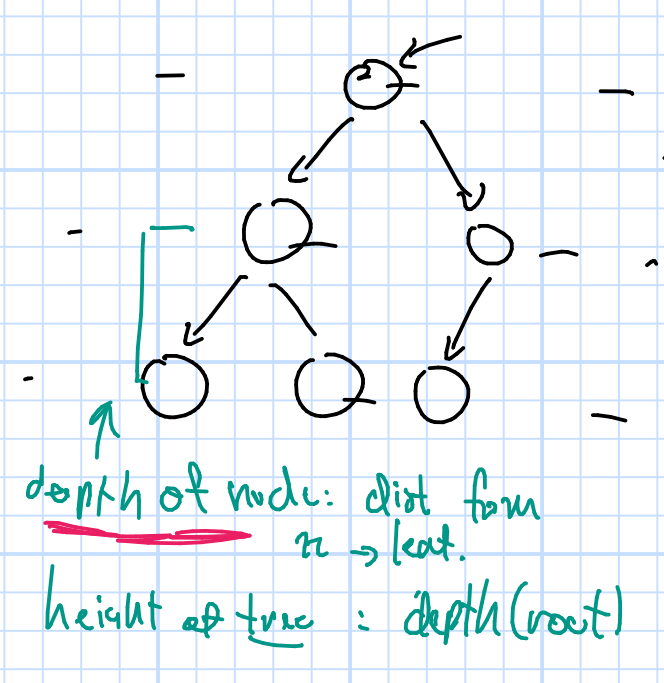


- directed graph w/ no cycles.
- n children/node binary → 2 trees.
- node w/ no children: leaf node.
- complete: every level filled last from L→R
- full: last filled as well.

depth of node: dist from n → leaf.
height of tree: depth(root)

3 types of tree traversals.

```
typedef node {
    int val;
    node* left;
    node* right;
} node;
```

PRE

```
void pre_order(node* n){
    printf("%d," n->val);
    if (n->left != NULL)
        pre_order(n->left);
    if (n->right)
        pre-order(n->right);
}
```

- copying a tree
- something that will go thru everything.

POST

```
void post_order(node* n),
    if (n->left != NULL)
        post_order(n->left);
    if (n->right)
        post-order(n->right),
    printf("%d," n->val);
}
```

- free-ing a tree

IN

```
void in_order(node* n){
    if (n->left != NULL)
        In_order(n->left);
    printf("%d, "n->val
    if (n->right)
        in-order(n->right);
}
```

- will, for a BST, traverse inorder of the elem.

Priority Queue

- insert
- peek - view min elem
- pop() - always return min. value in PQ

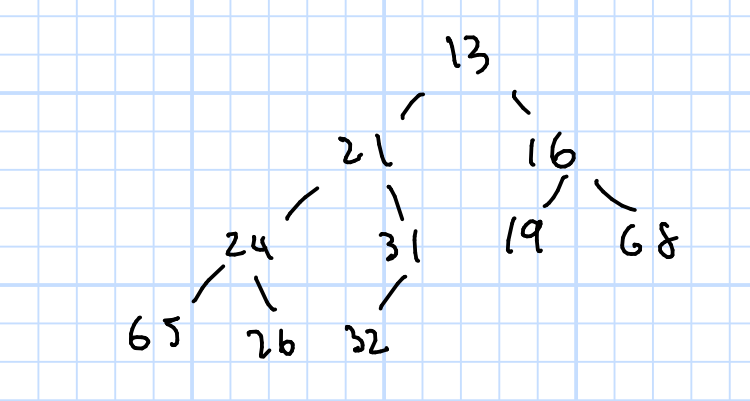
	insert	peek	pop
1) array	O(1)	O(n)	O(n)
2) sorted arr	O(n)	O(1)	O(1)
3) heap	O(log n)	O(1)	O(log n)

heap

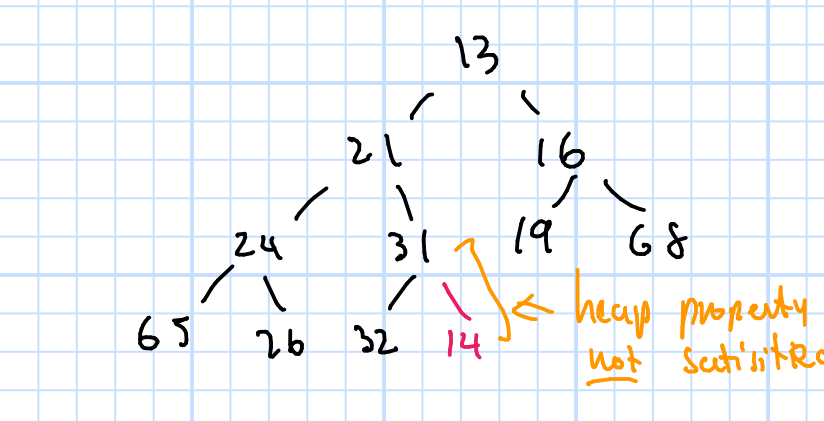
"heap property" / "heap invariant"

1) For all nodes that are not the root, the parent_val ≤ n. val } min-heap; max-heap ≥
2) min element is at the root.

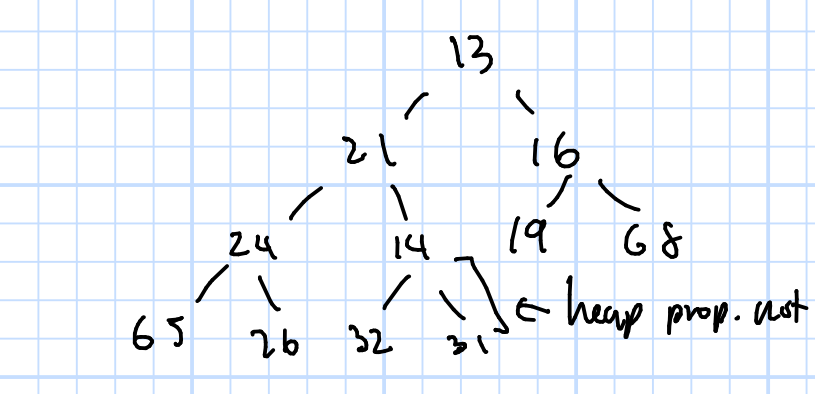
```
int peek(node *n){
    return n->val;    // check if n null too.
}
```



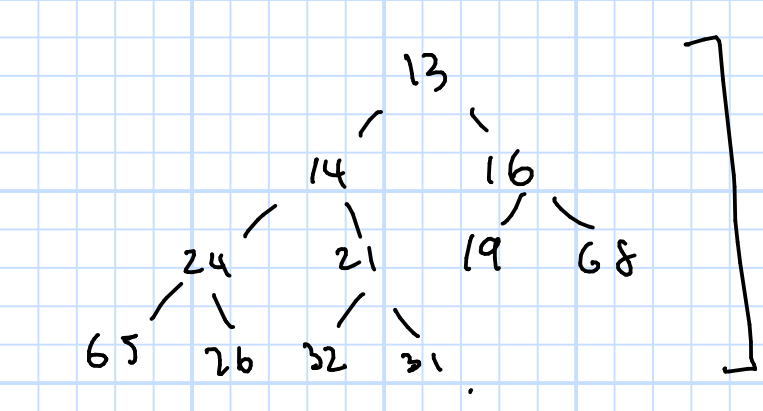
insert(14)



1) put it at the first ava. leaf node.



- swap 14,31



(swap 14,21)

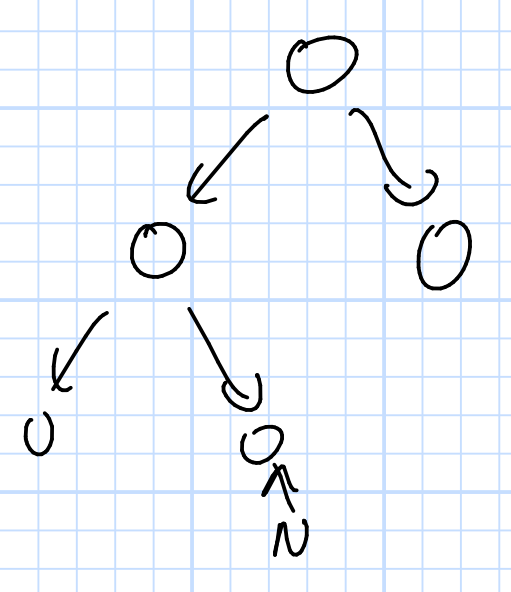
heap property is satisfied!

height ≈ $\log_2$(# of tree elem)

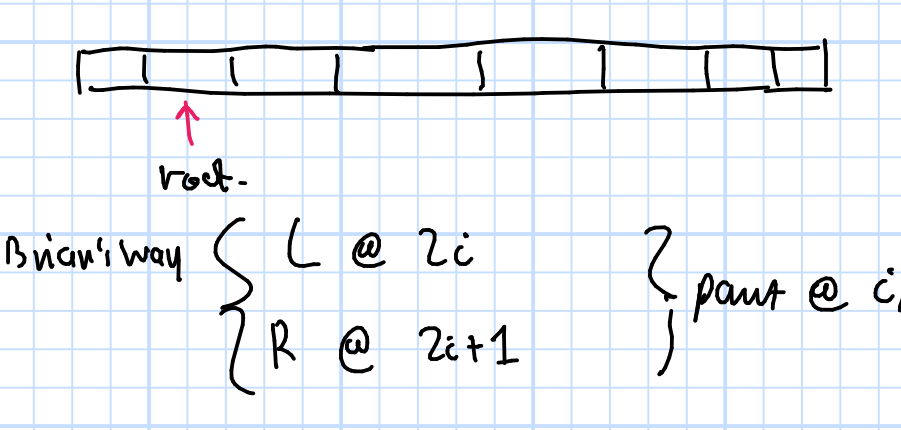
max # of swaps = height.

∴ insert is O(log(n)).

```
insert(heap, x):
// assuming arr. impl. of heap
// get first empty slot.
    k = len(heap)+1          // in arr impl, last elem is the last leaf node.
    heap[k] = x
    while (k>1 and heap[k/2] > pq[k]):
        swap(pq[k], pq[k/2])
        k /= 2
```

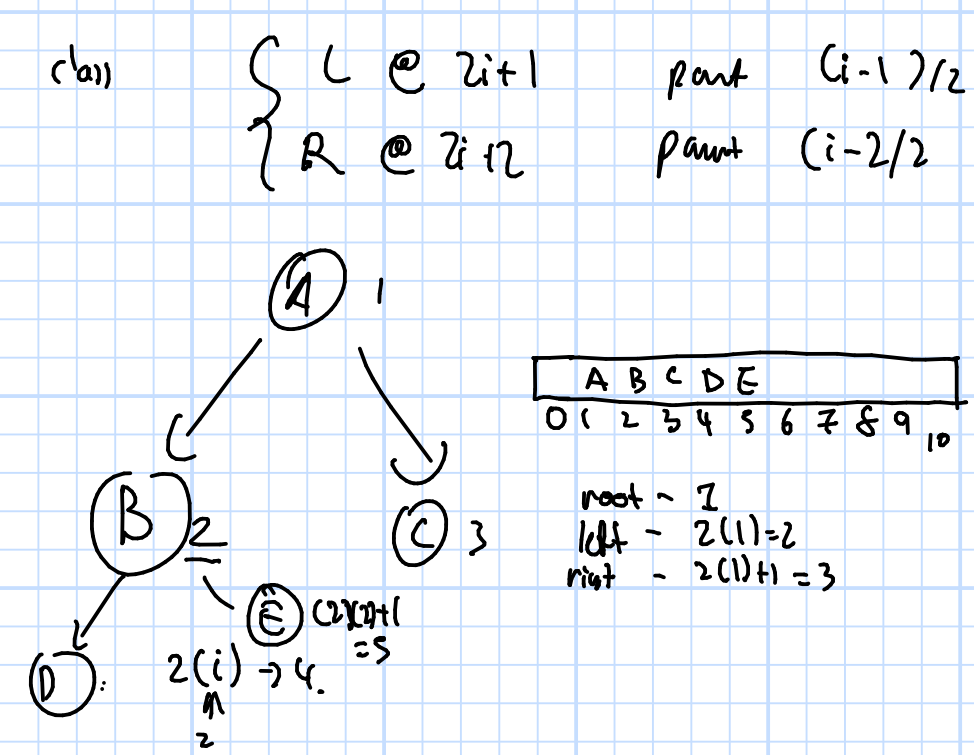


* in pointer impl, nodes typically don't know who their parent is.



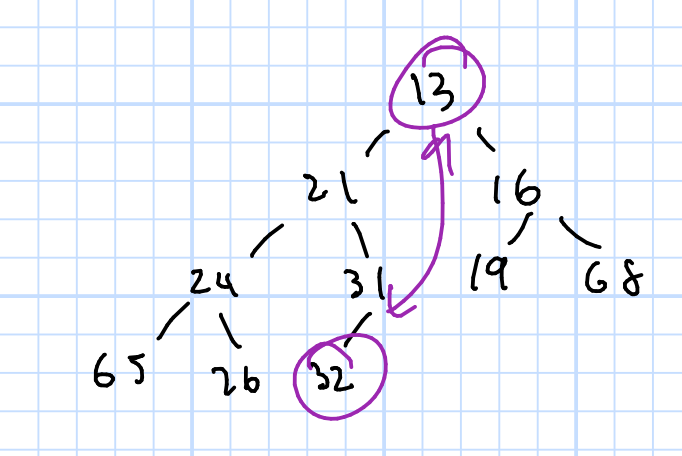
Brian's way { L @ 2i ; R @ 2i+1 } parent @ i/2   * root @ 1

class { L @ 2i+1 parent (i-1)/2 ; R @ 2i+2 parent (i-2)/2 }   * root @ 0.

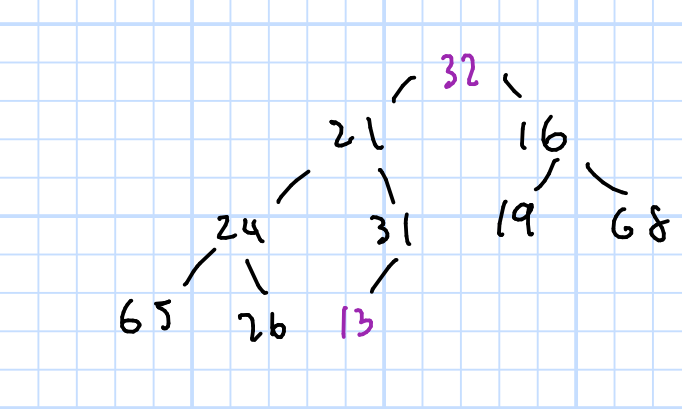


pop()

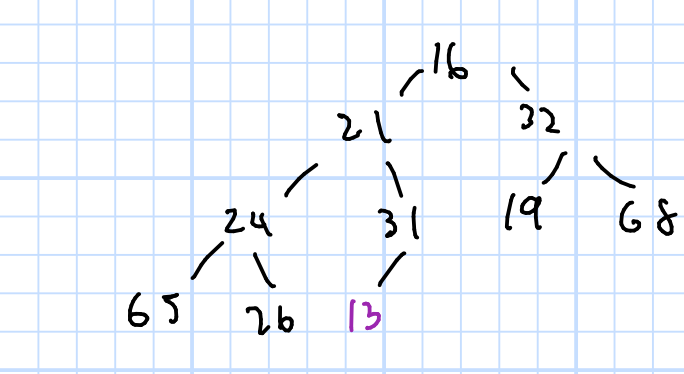
1) swap min elem w/ last elem.
2) bubble down the swapped element until heap property is satisfied
3) remove min elem.



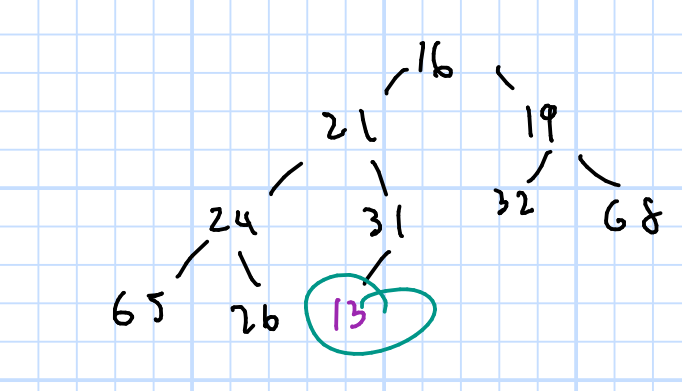
1) swap



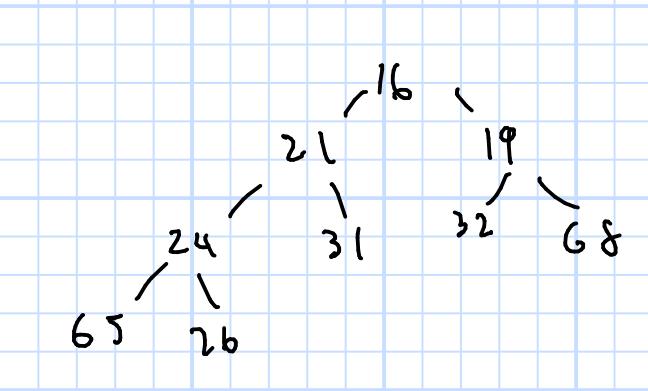
swap w/ 16.



on which side can we swap?
R: cause parent > child ∴ no.
L: OK. after swap parent < child.



swap 19,32



remove min elem & return.

(13)

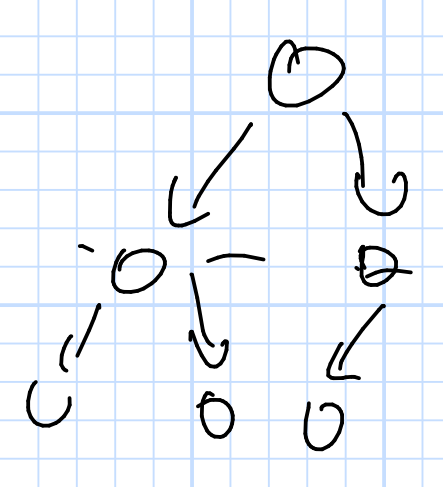
Building a heap

1) use insert() → (n calls to log(n) insert) → n log n.

2) heapify O(n)

```
heapify(list)
    i = len(list)/2
    while i>0
        bubble-down(list, i)   // exact same as pop.
        i -= 1
```

Upper bound on $\frac{n}{2}\log(n)$.



$h \times \frac{n}{2^{h+1}} \times C$ @ a level h.

$$\sum_{h=1}^{\infty} ch \frac{n}{2^{h+1}} = c\frac{n}{2}\sum_{h=1}^{\infty}\frac{h}{2^h} = c\frac{n}{2} \xrightarrow{\text{stack overflow.}} (2) = cn$$

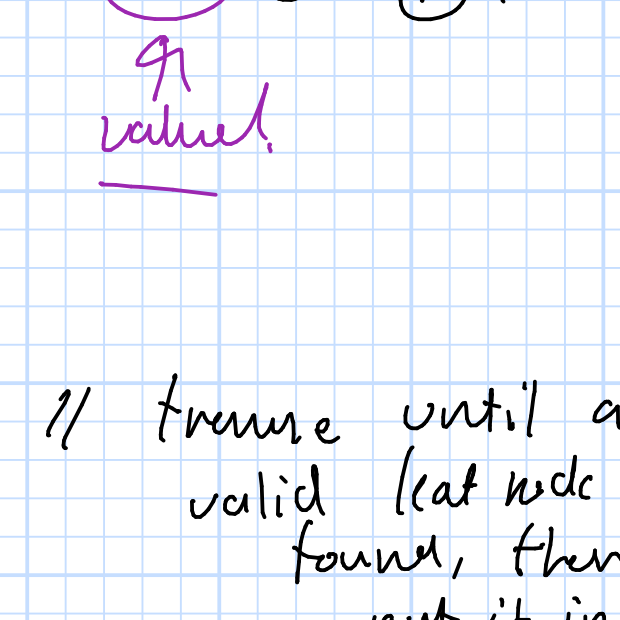
$O(cn) = O(n)$ — we don't care

BST

Property:
@ a node n
left child < n
right child > n. } implies no dup. values.

- traversal: in-order as per before.

```
int search(node* root, x){
    if (!root)
        return (-1);
    if (root->val == x)
        return (1);
    if val < root->val
        return search(root->left, val)
    if val > root->val
        return search(root->right, val)
```



search(5)

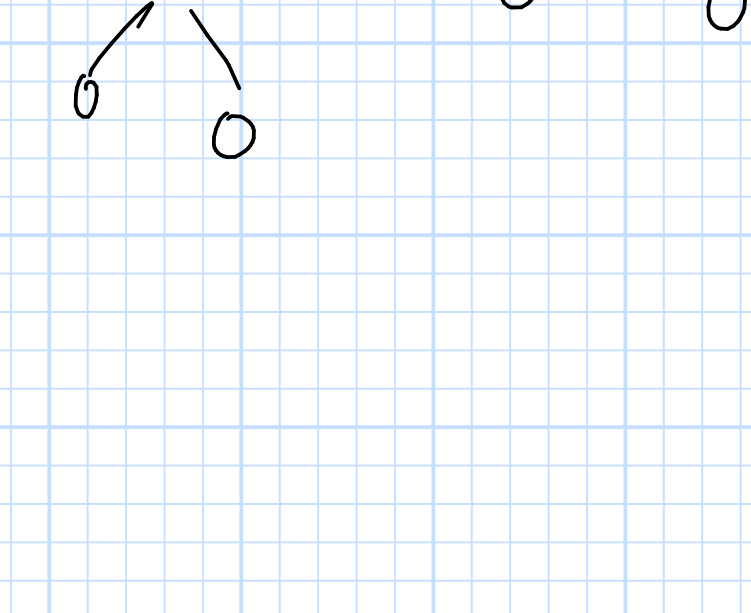
height ≈ log n (# nodes)
~ O(log(n)).

```
int insert(node* root, x)
    if (!root)
        - create a new node & link it up, etc.
    if val < root->val
        return search(root->left, val)
    if val > root->val
        return search(root->right, val)
```

// traverse until a valid leaf node is found, then put it in.

deletion

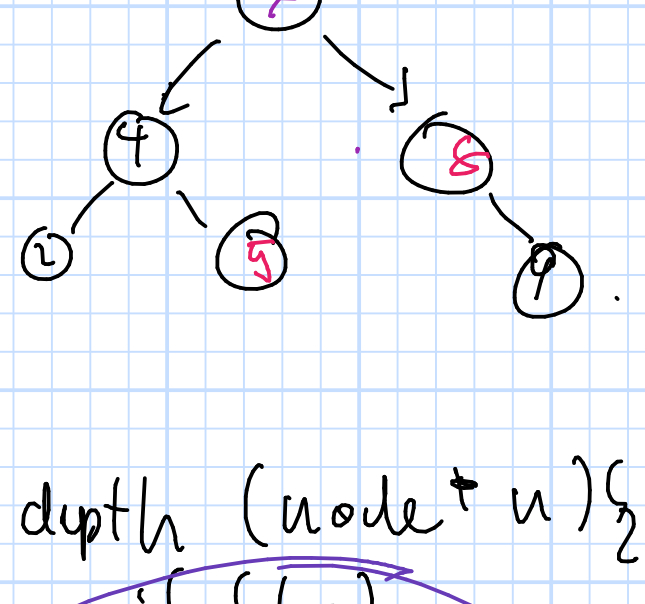
1) no children
→ free & remove/unlink it same as LL delete.



2) 1 child.
→ free & link grandparent w/ child.

2) 2 children
1) leave node in
2) find the "successor"
3) swap w/ successor
4) delete node.

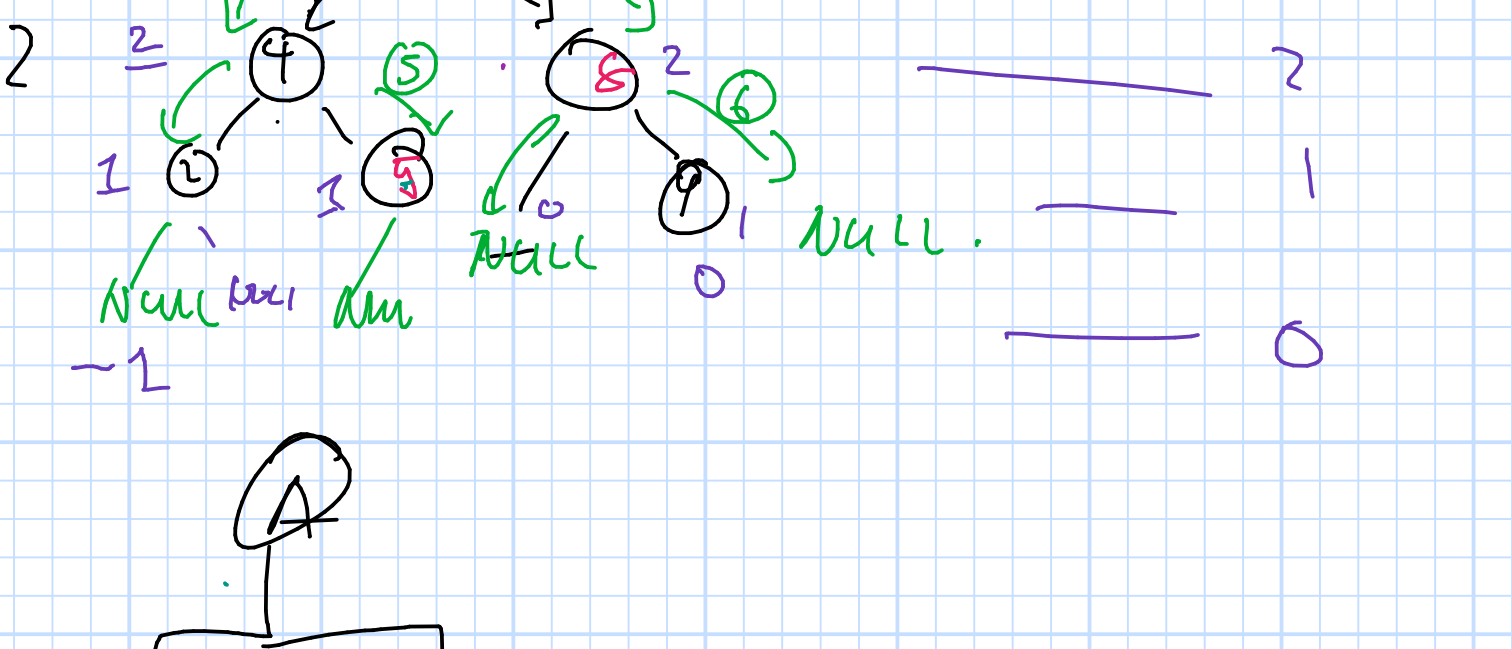
successor — next-largest value in the tree.
→ smallest val. in Right sub-tree.



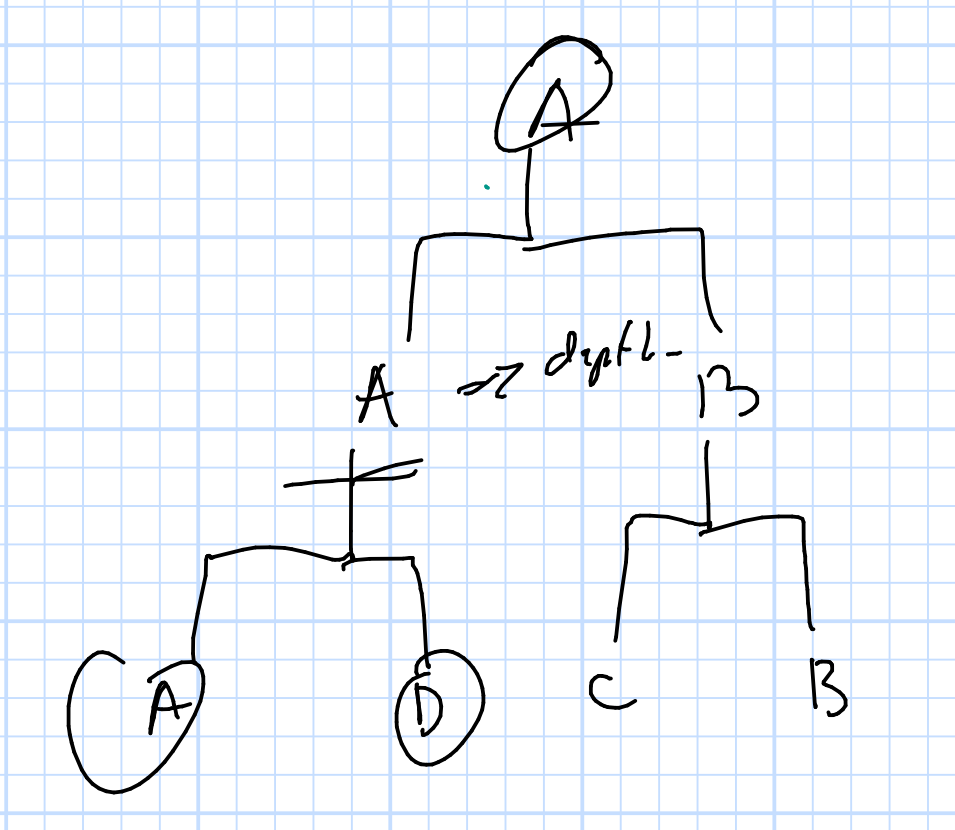
delete (6)

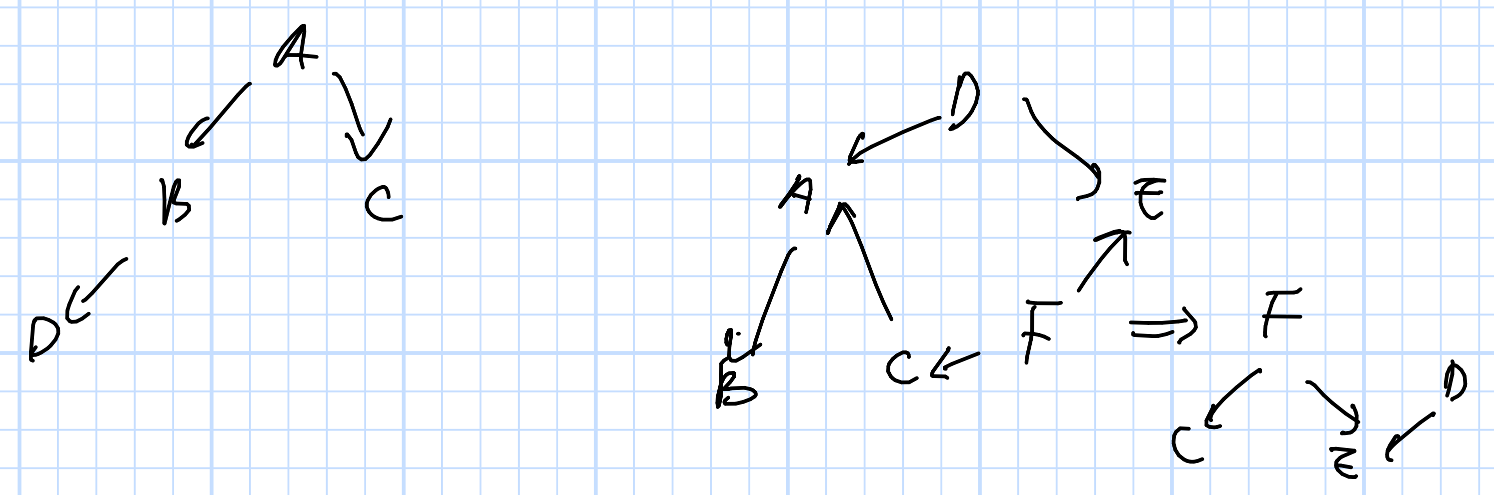
```
int depth(node* n){
    if (!n)
        return 0;
    int L = depth(n->left);
    int R = depth(n->right);
    if (L > R)
        return (L+1)
    else
        return (R+1)
},
```

depth(root) = ?

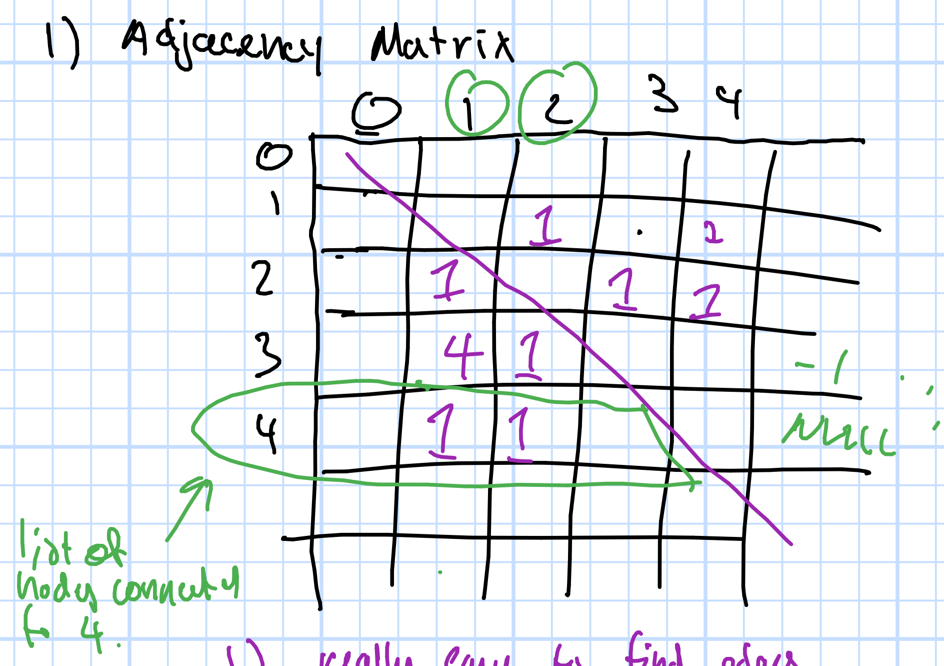


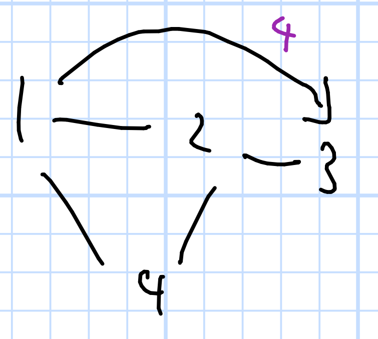
{ ① find top node w/id
② call depth on player one
③ get rank: (h+1) - d





1) Adjacency Matrix





arr[1][3]
arr[3][1]

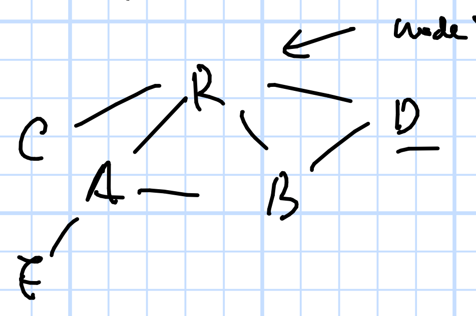
- can represent a directed graph by ommiting a side of the diagonal

1) really easy to find edges
→ O(1) lookup for edge weight.

2) Space complexity scales O(n²)

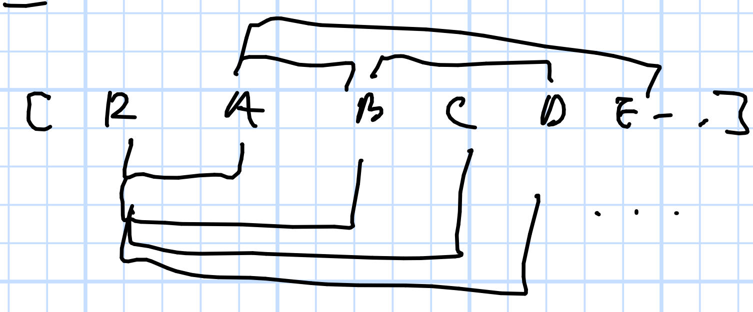
2) Pointers

2a) Everything relative to a root node & store a pointer to the root node



ex: add node in E & D.

2b)



→ some arr/LL/etc. of nodes w/ edge information encoded in the pointers.

→ Advantage: can access nodes more easily.

1) Add a node
2) Remove a node
3) Add an edge
4) Remove an edge
5) Traverse a graph
- Depth-first
- Breadth-first
6) Shortest path

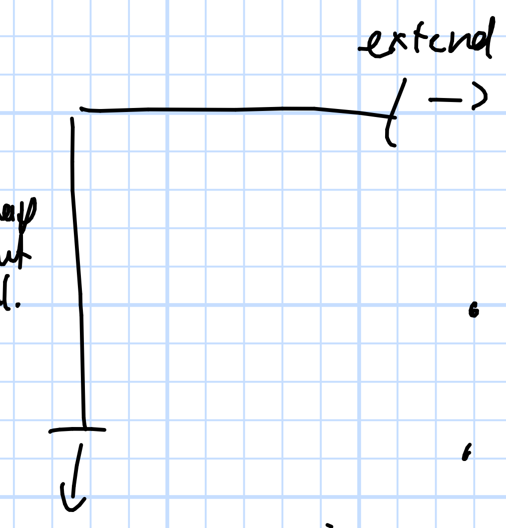
```
struct node {
    int val;
    node*[] neighbours;
    node* next;
};
```

[. . node3 , . .]

node →(next) node →(next)
& neighbours by the neighbour arr.

1) Adding node

ARR:



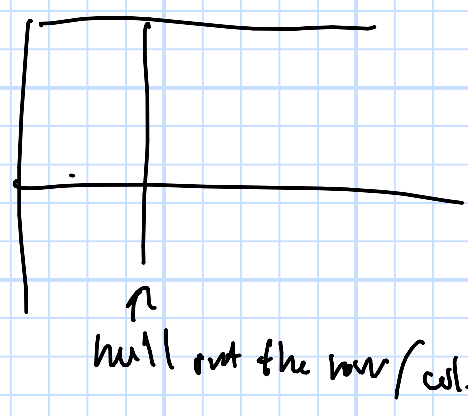
* can use dict/hashmap to implement as well.

Pointer-only
- we would have to add nodes w/ the edges

ARR + Ptr:
append a new node

2) deleting a node

ARR repr.



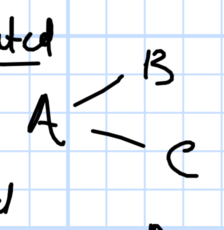
null out the row/col.

- you just null it out
- if we have to add/delete a lot reconsider if ARR is the right way to go

Pointer-only

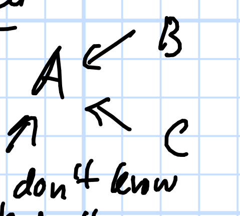
1) Find the node.
→ Traversal algo
- until find right node
- unlink from graph
- delete it.

undirected



→ go to B, C, unlink A, then free. A.

directed



we don't know about the references to A, from A

- as traverse graph
- keep track of ref → A
- then can free A & remove ref.

inserting an edge

ARR

ARR[E1][E2] = weight
" [E2][E1] . . .

Ptr

1) - get ref to the two nodes we want
2) - update their neighbours accordingly.
→ append a ptr.

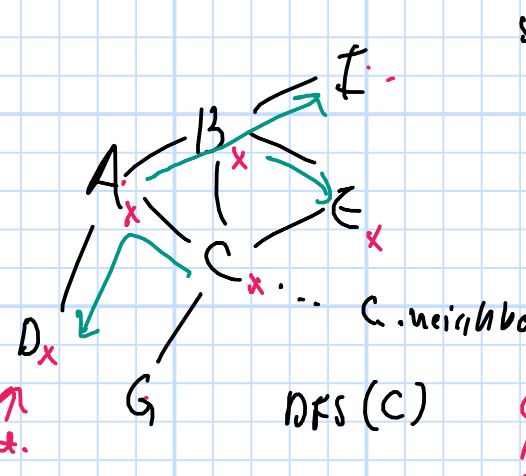
1) Ptr only
→ may have to traverse graph to get ref.

2) Arr repr
→ can just go along array to get ref.

Traversals

Depth-first

```
DFS(v)
    v.explored = true
    ∀ w ∈ v.neighbours :
        if !w.explored:
            DFS(w)
```




```
struct node {
    int val;
    node*[] neighbours;
    int explored;
};
```

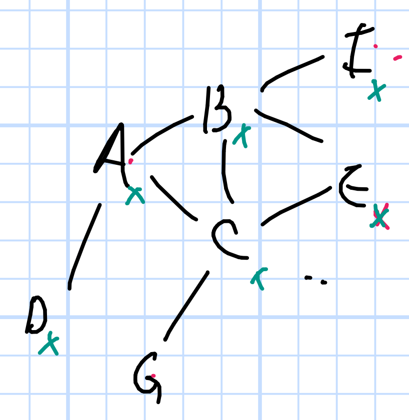
C.neighbours . . [A, B, E, G]

DFS (C)

C
A
D
B
F
E

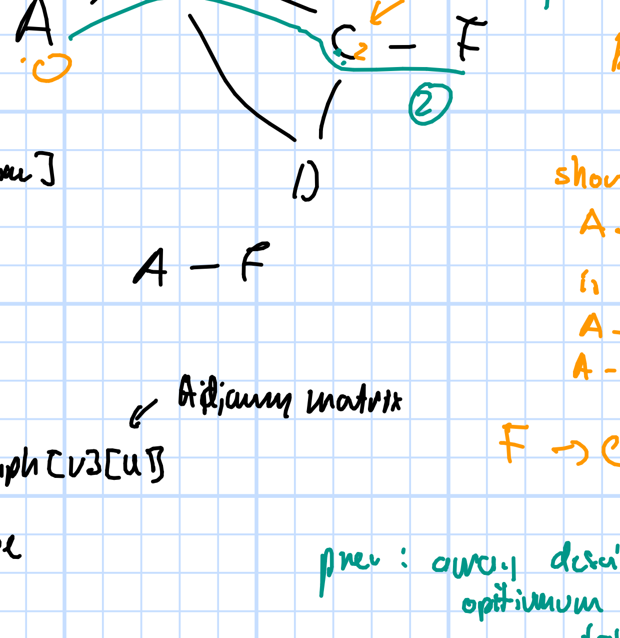
Breadth First

```
BFS(v)
    Q = Queue()
    Q.push(v)
    while Q:
        v = Q.pop()      ← do stuff w/ v here.
        ∀ w ∈ v.neighbours:
            if !w.explored     ←
                w.explored = true
                q.push(w)
```



BFS(A)		Queue
		[A]
A	pop	[B, C, D]
B	pop	[C, D, F, E]
	repeat.	

Dijkstra



A - F

```
dijkstra (v)
    prev - [NULL .. n_nodes]
    dist - ARR [INF .. n_nodes]
    q: PQ
    dist [v] = 0

    while q:
        v = q.popmin()
        ∀ u ∈ v.neighbours
        → alternative = dist[v] + graph[v][u]     ← Adjacency matrix
            if alternative < dist [u]
                dist [u] = alternative
                prev [u] = v
    return dist, prev
```

idea: are any of the alternative ways better? If so use it.

intuition: break into subcases

shortest path from A - F
is shortest path
A - C then C - F
A - B, then B - F . . .

F → C → B - A.

prev: array describing the optimum previous node to take.

dist: distance from v → a node . . .

Heap
- popmin - O log N
enqueue - O log N

$O(E) \times O(\log V)$

$\rightarrow O(E \log V)$

2B(i)

int *p;

*p = malloc . . .

deref → *p

func(*p)

invalid read + write

int i;

p = ____ ← here

*p → ____

**p → ____

} here-not.