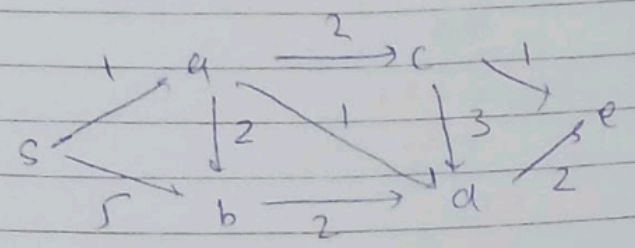
~~Lec-11~~ Lec-11

# Single source shortest path




Que: shortest path of s from all other nodes

Naive soln → break 5 into 5 × 1 edges & then use BFS

↓

(i) - max wt can be huge (Google maps)

↳ t.c. inc.

(ii) - Real wts

(Fractional, Negative ....)

---

Assumption: non-negative wts

(but can be real)

| | |
|---|---|
| a | 1 |
| b | 3 |
| c | 3 |
| d | 2 |
| e | 4 |
| s | 0 |

~~a - 1~~

~~d - 2~~

~~b - 3~~

~~c - 3~~

~~e - 4~~

Pf (By contradiction)

Assume this is not true

$\exists$ a path to

$s \rightsquigarrow y^* \rightsquigarrow v^*$ s.t. dist is <

of $y^* \in V/A$

~~it chosen P at some time~~

~~the one of the intermediate~~ if it would have been argmin in

$\rightarrow$ dw

hypothetical smaller path < path for algo

not path

~~Since~~ since

$\arg\min \; d(u^*) + wt(u^*, v^*)$

$\wedge\!\!|$

$d(y) + wt(x, y)$ ~~+ wt(y, v*)~~

$\Rightarrow d(u^*) + wt(u^*, v^*) \nless d(x) + wt(x, y) + wt(y, v^*)$

$\Rightarrow\Leftarrow$

Procedure SSSP(G, ~~w~~ s) Dijskhra's algo ~~(in networks)~~

// G = (V, E)

d ← empty array of size |V|

for each v ∈ V

d[v] ← ∞

d[s] ← 0

π ← empty array of size |V|

π[s] ← NULL

~~Inct~~

Include s to A ~~A ←~~ A is "seen" set

PQ ← priority queue of items = V, key defined by array d.

while (A ≠ V)

u ← PQ.extract-min() // d(s,u) is known

for each (u,x) ∈ E

PQ.update.key(x, d(u) + wt(u,x))

only if d(u) + wt(u,x) is smaller than original key

or

if (d(x) > d(u) + wt(u,x))

π[x] ← u

PQ.update-key(x, d(u) + wt(u,x))

endif

endfor

endwhile

or ~~while~~ run a counter from 1 to |V|

like dequeue or pop

similar to enqueue or push in DFS

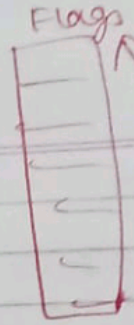
really used in networks

PQ will use the d array & thus do updations to d array itself.

PQ has d array inside it

Naive PQ
↳

Flags & go over PQ & find min



for seen or not

∵ each vertex visited exactly once

while loop ← $\Theta(n)$

extract_min ← $\Theta(n)$

for each $(u,x) \in E$ ← m total ⇒ $\Theta(n^2+m)$

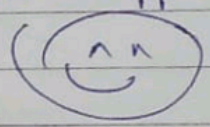
∵ ~~each~~ for visiting edge again

& satisfy > condⁿ

would need to visit a vertex twice

which doesn't happen

(Think!)



Linked List ← necessary

- ~~Sim~~ Procedure SSSP $(G, s)$ Dijkstra's
(Assumes non-neg wts)

```
// G = (V,E)
d ← empty array of size |V|
for each v ∈ V
    d[v] ← ∞
endfor
d[s] ← 0

π ← empty array of size |V|
π[s] ← NULL
PQ ← priority queue with items V and keys defined by d

Include s to A
while (A ≠ V) // or run a loop from 1 to |V|
        u ← PQ.extract-min()
        Include u to A

        for each (u,w) ∈ E
                if (d[w] > d[u] + wt(u,w))
                        PQ.update-key(w, d[u]+wt(u,w))
                        π[w] ← u

                endif
        endfor
endwhile
```

Note PQ contains the d array

- A näive PQ can be a simple array storing key-value pair, and to store min we iterate over each element of array ☺

- Time of naive implementation : $\Theta(n^2+m)$ n: no. of nodes
m: # of edges
if extract-min() is $\Theta(n)$
& update key is $\Theta(1)$

We will store it as an array

| 5 | 7 | 8 | 12 | 17 | 10 | 11 | 19 | 13 | |
|---|---|---|---|---|---|---|---|---|---|

~~dist~~ 0 dist 1 . . .

heap-size = 9

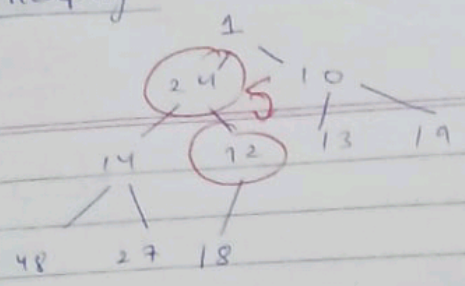
Procedure Left(x)
// x is index of node.
// returns index of the left child ~~of the~~
return $2x$

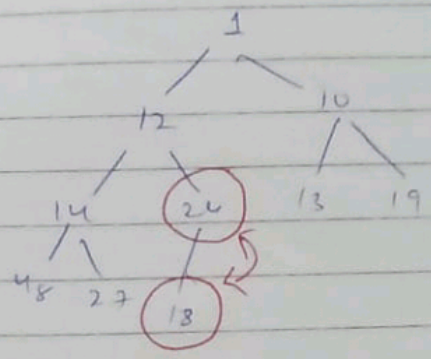
Procedure Right(x)
return $2x+1$

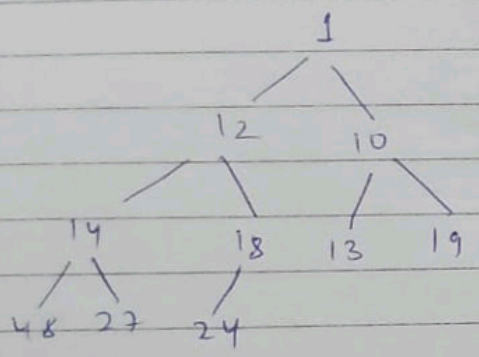
Procedure Parent(x)
return $\left\lfloor \frac{x}{2} \right\rfloor$

{if 0 is returned we ~~get~~ know $x = 1$} $\rightarrow$ we are at ~~x is~~ root.

Min-Heapify




we can make an array of structs

Procedure Min Heapify (A, x)

// A has a violation of heap property at index x.

while (1) {

minIndex ← x;

if (left (x) ≤ heap-size and A [left (x)].key < A[x].Key)

minIndex ← left(x)

end if

if (right (x) ≤ heap-size and A[right(x)].key < A[minIndex].key)

minIndex ← right (x)

end if

$T(l) = T(l-1) + \Theta(1)$

$\Rightarrow \Theta$

if (minIndex ≠ x)

tmp ← A[x]

A[x] ← A[minIndex]

A[minIndex] ← tmp

minHeapify (A, minIndex)

Recursive

else

return

endif

endwhile

T.C.

Q: ~~How ma~~ ht of a heap with n items?

Let $ht = l \Rightarrow l-1$ levels are full.

$$1 + 2 + 2^2 + \cdots + 2^{l-1} + [1, 2^l]$$

$$= 2^l - 1 + [1, 2^l]$$

$$= [2^l, 2^{l+1} - 1]$$

$$\Rightarrow \quad l \le \log_2 n$$

$$\log_2(n+1) - 1 \le l$$

$$l = \Theta(\log_2 n)$$

$\therefore$ Min Heapify $\quad T(l) = T(l-1) + \Theta(1)$

$$\Rightarrow T(l) = \Theta(l)$$

$$= \Theta(\log_2 n)$$

T.C. of Build Heap $\Rightarrow$ crude estimate ~~$\Theta(n$~~ $O(n \cdot \log_2 n)$

# Heap Sort

Procedure HeapSort (A, n)

B ← BuildHeap (A, n) // O(n)  correctness is obvious

~~heap size ← n~~

for j = 1 to n

print (ExtractMin (B)) // O (n log n)  reason why not O(n)?

- stability?
- space?
- etc.

log n + log n-1 + log (n-2) ... log(1)

= log (n!) ~~≤ log (n^n) = n log n~~

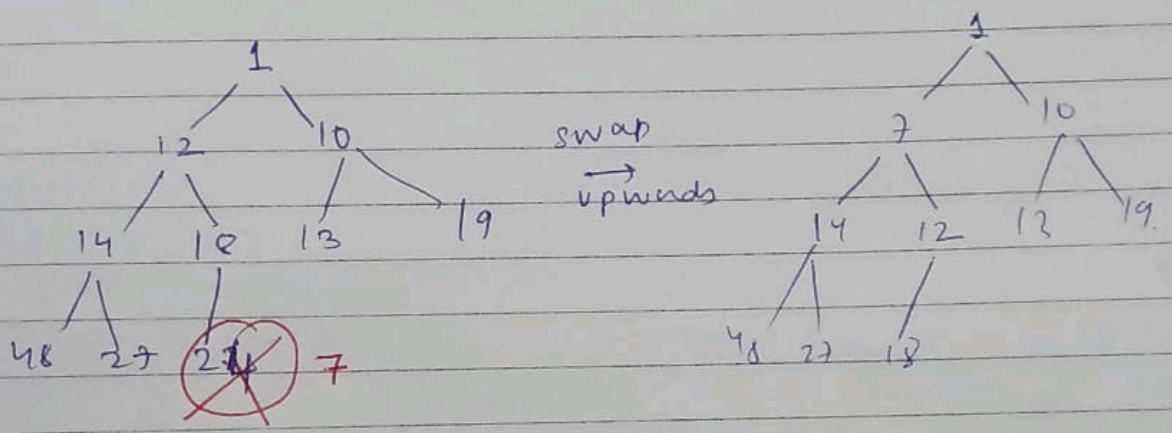
// Stirling's approx

n log(n) − n + O(log n) ⇒ log (n!) = O (n log n)

~~Min log(n!) / n→∞ n log(n^n) =~~

(15)

Decrease-key operation.




Procedure Decrease-Key (A, x, newKey)

~~A[x] ← newkey~~

if x > heap-size return error endif.

// check if A[x].key is already less equal a.

A[x].key ← newkey

while (1)

if (parent(x) = 0) break endif

~~tmp ← A[x]~~

if (A[parent(x)].key < A[x].key) break endif

tmp ← A[x]

A[x] ← A[parent(x)]

A[parent(x)] ← tmp.

x ← parent(x)

endwhile

ity Queue — heap
— mapping: vertex → heap index

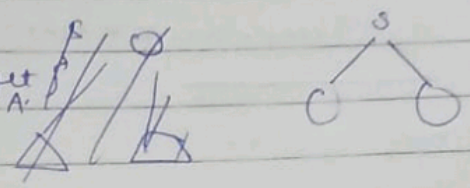
$\infty \equiv \sum w(e) + 1$ (in shortest path algo)

d:

| | | | | |
|---|---|---|---|---|
| ∞ | ∞ | | 0 | ∞ |
| 1 | 2 | | s | n |

(source node)

we'll call build heap on d. ⇒ to get heap A.
(ofc. s will be at top)
∵ d[s] = 0



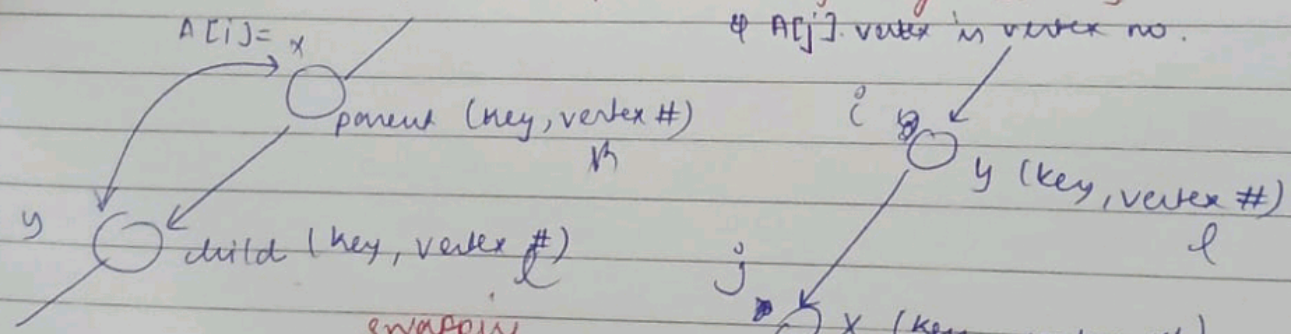
heap is array of structs: {key, vertex #}

mapping:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| H | j | 5 | 4 | | 1 | | |
| vertex # | 1 | 2 | 3 | 4 | . . . | 8 . . . | n |

store index in heap

gives the pos° in heap say j then A[j].key is key & A[j].vertex is vertex no.




swapping happens in ~~Build~~ Min-Heapify & Decrease-key

mapping

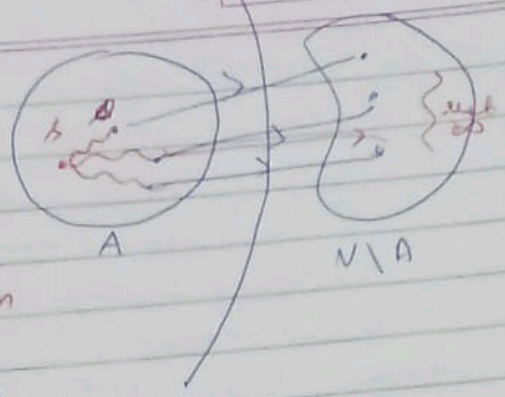
| | |
|---|---|
| i | j |
| k | l. |

mapping

| | |
|---|---|
| j | i |
| k | l |

T.C. of Dijkstra

Procedure SSSP ( . . . . . )




while i=1 to |V|

$\Theta(\log n$ times) → u ← extract-min(A) } n log n

for each (u,w) ∈ E

If (d(w) > d(u) + wt(u,w))

(m time only as u only visited once)

π(w) ← u

~~d(w) ←~~

$\Theta(\log n)$ → decrease_key(A, —, —)

A × V\A

left endpt  right endpt

Time Complexity

$O((m+n)\log n)$  $O(m+n^2)$

$m \le \left(\frac{n}{2}\right) = \Theta(n^2)$

$O((m+n)\log n) \ll O(n^2)$

unless. $m = \omega(n^2/\log n)$

$O(m + n\log n) \ll O(n^2)$

↓

Fibonacci Heap

can also use π array here

Procedure SSSP (G, s)

d ← empty array of size |V| initialised with ∞.

A ← array of ~~structs~~ pairs made of (idx, d[idx])

PQ ← heap ← BuildHeap(A, n)

PQ.mapping ← empty array of size |V|

for i = 1 to n

PQ.mapping[PQ.heap[i].vertex] ← i  inverse

Decrease-key(PQ, s, 0)

for i = 1 to n

u ← Extract-min(PQ)

for each (u,w) ∈ E do

if (wt(u,w))

{ PQ.mapping[u].key

PQ.mapping[w].key

Decrease-key(PQ, u, s)

mapping stores current position of vertex j within heap A.

Ask sir, + verify d implementation PQ

A[j].key

.vertex# → ~~mapping[A[j].vertex#] ← i~~

extract-min : call extract min of underlying heap.

(after implementing ~~this~~ updating ~~the~~ of mapping after ~~swapping~~ swaps)

decrease-key : call decrease-key of heap.

Ask

~~Procedure~~

PQ → heap A , mapping array

n = |V| , A ← ~~struct~~ array of structs ~~made~~ = {v, d(v)}

PQ.heap ← Buildheap (A, n)

~~mapping array~~ = PQ.mapping ← empty array of size n.

for (i = 1 to n)

PQ.mapping[A[i].vertex number].idx ← i

~~endfor~~

PQ.mapping[A[i].vertex number].key ← A[i].key

endif

Procedure Extract-min ()

if (heap-size = 0) return error endif

x ← A[1].~~key~~ vertex#

key ← A[1] // A[1] ← A[heap-size] // A[heap-size] ← key

PQ.mapping[A[1].vertex no.].idx = 1 // PQ.mapping[A[heap-size].vertex no] = —

~~key = A[1].vertex no.key~~

heap-size --

Min-heapify (A, 1)

return x

In min-heapify & Bubble whenever swap remember to update in PQ mapping

↳ PQ.mapping[A[x].vertex no].idx = x

PQ.mapping[A[y].vertex no].idx = y

Procedure Decrease-key (i, a) at vertex i

//check i in bounds & val greater than a.

PQ.mapping[i].key = a // A[PQ.mapping[i].idx].key = a

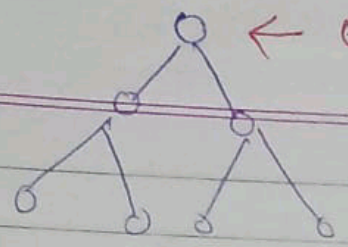
Bubble up (A, PQ.mapping[i].idx)

T.C ← O(log (n))

as ~~goes above each~~

at each iteration it goes up one level.

Let's do calcⁿ more carefully:- $S - 2S = 1 + \sum_{j=2}^{l} 2^{j-1} - 2^l \cdot l - 1$



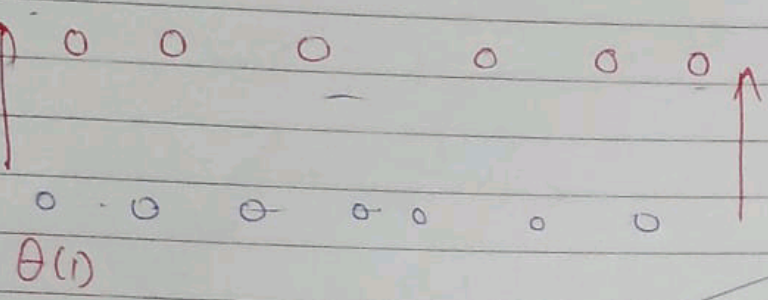
$\leftarrow \Theta(\log_2 n)$

$\sum_{j=1}^{l} 2^{j-1}(l+1) - \sum_{j=1}^{\log n} 2^{j-1} \cdot j$

$(l+1) \cdot \frac{(2^l - 1)}{1} - (2^l - 1 - 2^l \cdot l)$

$= (l+1)(2^l - 1) - (2^l(l-2)+1)$

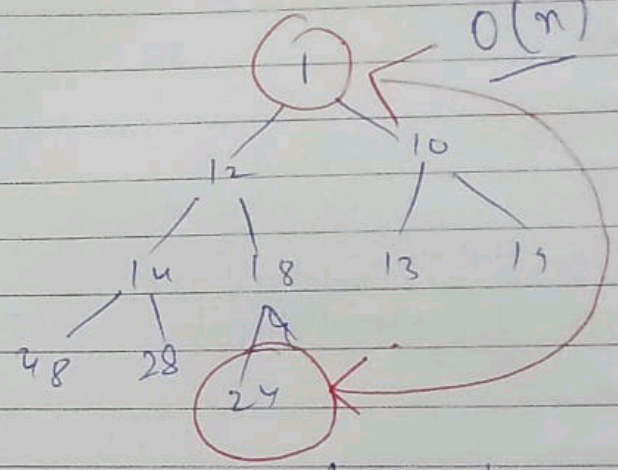
$(l+1)(2^l - 1) - 1$




Ex.

$$C = \sum_{j=1}^{\log n} 2^{j-1}(l-j+1) = \Theta(n)$$

By Product : Heapsort




$O(n)$

heap-size --
call min heapify

Procedure ExtractMin(A)

// Heap A, return min

tmp ← A[1]

A[1] ← A[heap-size]

A[heap-size] ← tmp

heap-size --

Min Heapify(A, 1)

return A[heap-size +1]

$\leftarrow O(\log n)$

~~if (heap-size = 1)~~

~~heap-size --~~

~~return~~

if (heap-size == 0) return error endif

Procedure Build-Heap (A, n)

// Array A of size n. (assuming A has valid structure)

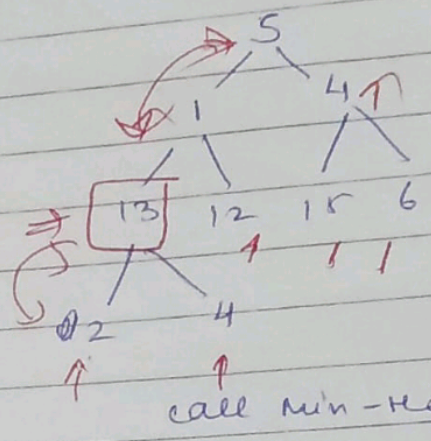
maybe a global variable { heap-size ← n

for int i = n to 1

Minheapify(A, n)

endfor

| 5 | 1 | 4 | 13 | 2 | 15 | 6 | 2 | 4 |



~~Wouldn't extract min run for $O(n^2)$ or $O(n\log n)$ (Hag.) for naive implementation of PQ (cuz how to check where in A)~~

- Accessing int* then why log(n) isn't size of address fixed?

Date :
Page :

Lec-12

- Lec 5 summary

Lec-> summary exp. Induction for T.C.

Priority Queue — key / data

- extract-min: removes item with smallest key & returns it

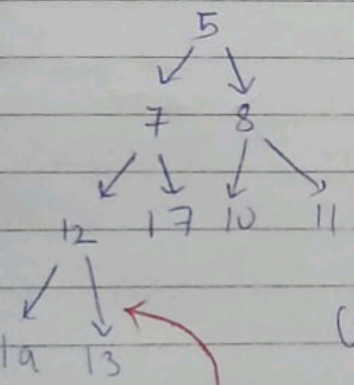
- d(v): stores a vertex → key mapping & when queried, returns the key of the queried vertex.

- decrease-key: reduces key of a ~~an~~ an item.

(Min)

(Binary) Heap → ~~Based on~~ Graph (Binary Tree)

↳ ~~Just an~~ array stored as — but is a




binary Tree

- each vertex is a key
- Every parent has 2 children and every node has got a unique parent — 3 if ternary allowed to violate last level & its parents

Complete Binary Tree

- All but last levels are full (Last can be full too) — if agree with
- Last level is filled ~~to an~~ left to right (no gaps)

If arrows not mentioned they are understood to pt downwards.

- Parent's key is smaller that that of the children (Min)

⇓

root is smallest

Lec-11

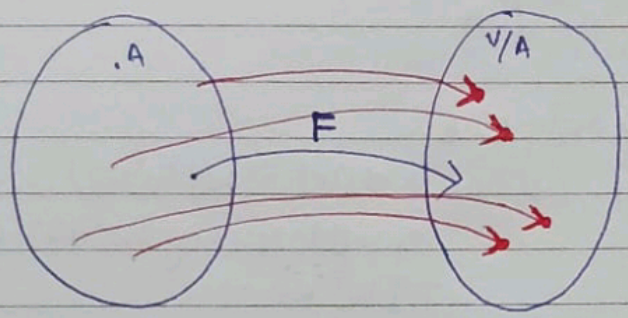
[SUMMARY]

SSSP

- It is possible to use BFS for SSSP after ~~writing~~ converting w edge with wt w to w edges.

(This clearly doesn't work for ~~neg~~ -ve & fractional wts and hurts the space & time complexity in case of large wts)

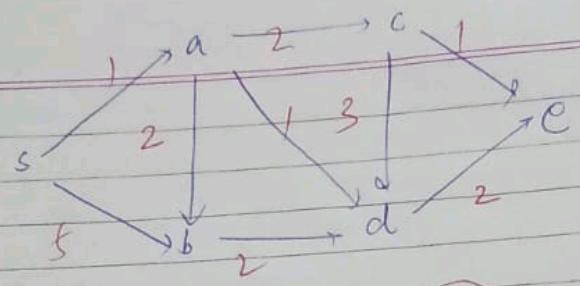
-




$\underset{(u,v)\in F}{\operatorname{argmin}}\{dist(u) + wt(u,v)\} = (u^*, v^*)$

$A = \{w \mid$ shortest path from $s$ to $w$ has been determined$\}$

Then, $s \rightsquigarrow u^* \rightsquigarrow v^*$ is a shortest path from $s$ to $v^*$




| | |
|---|---|
| ~~0~~ | 0 |
| ~~1~~ | 1 |
| ~~2~~ | 2 |
| ~~3~~ | 3 |
| ~~2~~ | 2 |
| ~~4~~ | 4 |

Let's say naive PQ

A

s, a, d, c

b, e

choose key-value in an array and scan every time to find minimum

~~Naive~~ ~~Naive~~

DFS — stack
BFS — queue
SSSP — priority queue

the item to be dequeued is the minimum.
(w.r.t some key)

Database

Roll No.
. . . . etc.
Key = Roll No.
will get smallest Roll No.

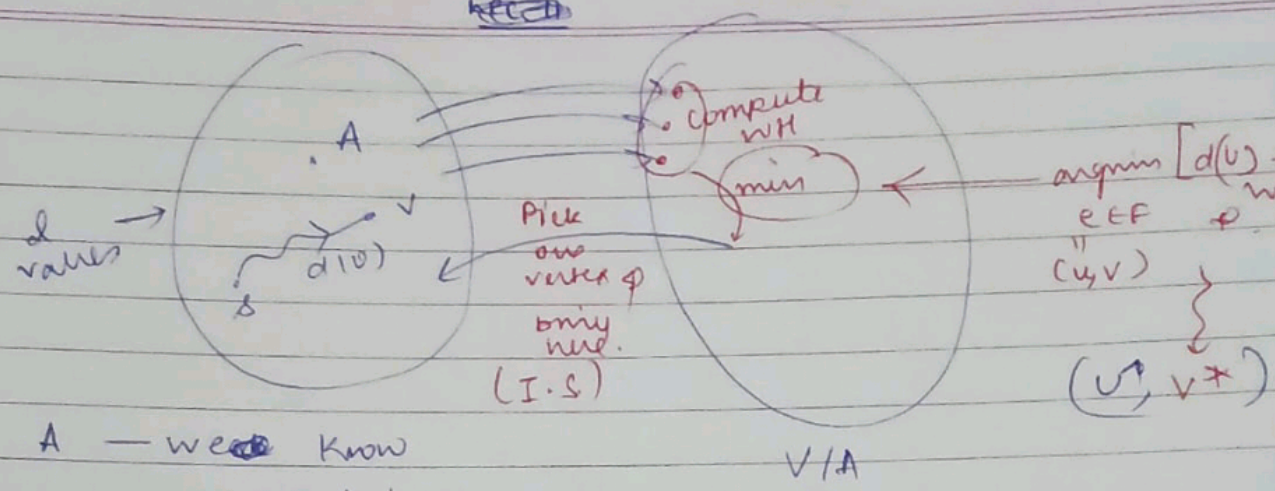
- sorting works?
  - not good for updates
  - suppose ROLL No. i.e. key value is dec.

the key would be currently known shortest path to this node.

## Priority Queue Data Structure

- query a particular item — get key of this item.
- update key to smaller value
- extract the min$^m$ key item

# none - else
# Accessing vertex
# $\Sigma$ outdeg.



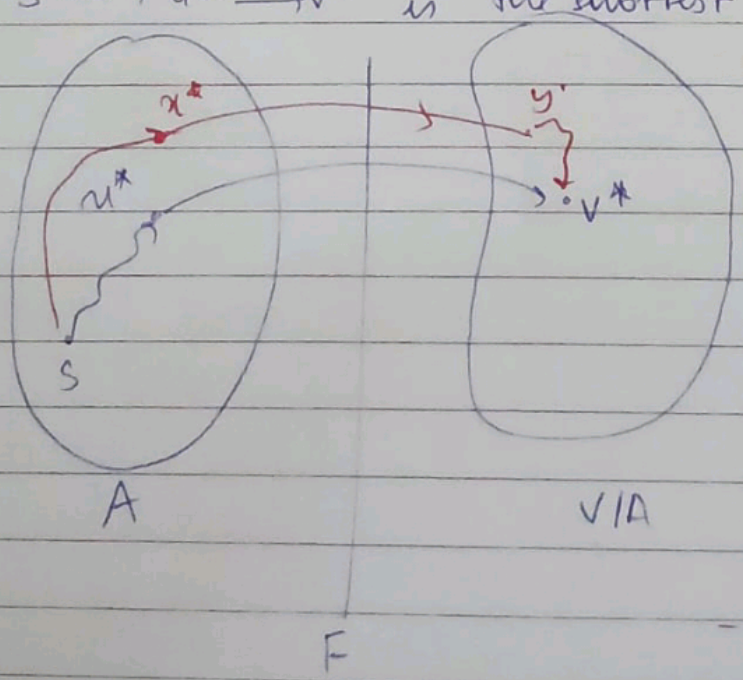

A — we know the shortest path from s.

$F \subseteq E$

edges from A to V/A

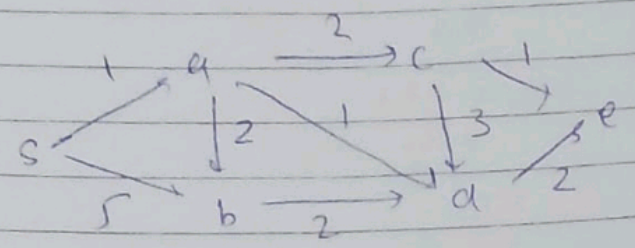
$F \subseteq A \times \{V \setminus A\}$

Claim: $s \rightsquigarrow u^* \longrightarrow v^*$ is the shortest path to $v^*$



~~Lec-11~~ Lec-11

# Single source shortest path




Que: shortest path of s from all other nodes

Naive soln → break 5 into 5 × 1 edges & then use BFS

↓

(i) - max wt can be huge (Google maps)

↳ t.c. inc.

(ii) - Real wts

(Fractional, Negative ....)

---

Assumption: non-negative wts

(but can be real)

| | |
|---|---|
| a | 1 |
| b | 3 |
| c | 3 |
| d | 2 |
| e | 4 |
| s | 0 |

~~a - 1~~

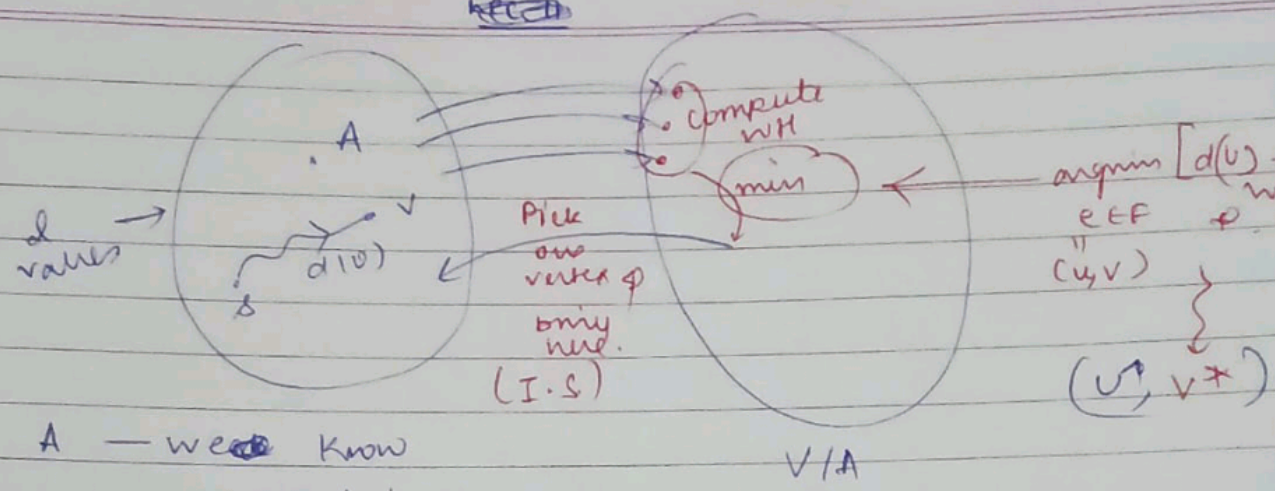
~~d - 2~~

~~b - 3~~

~~c - 3~~

~~e - 4~~

# none - else
# Accessing vertex
# $\Sigma$ outdeg.



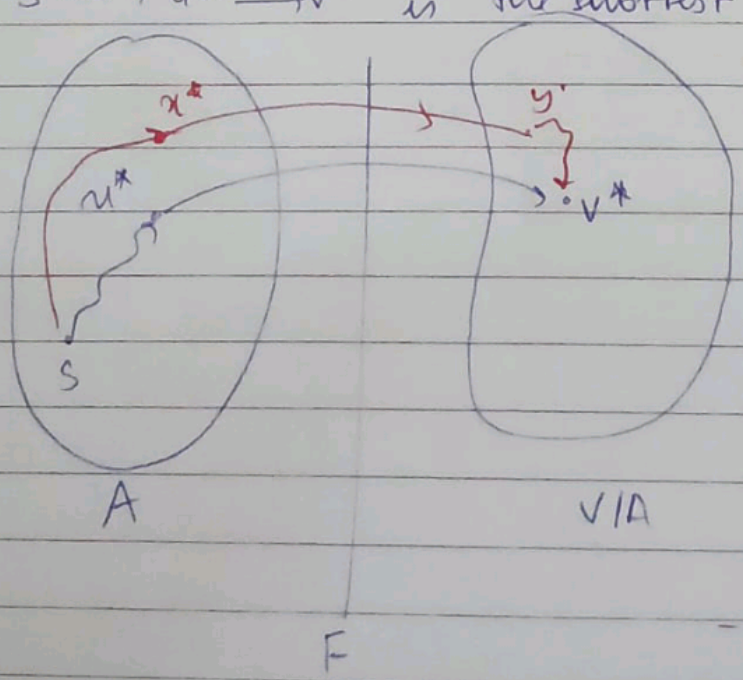

A — we know the shortest path from s.

$F \subseteq E$

edges from A to V/A

$F \subseteq A \times \{V \setminus A\}$

Claim: $s \rightsquigarrow u^* \longrightarrow v^*$ is the shortest path to $v^*$



Pf (By contradiction)

Assume this is not true

$\exists$ a path to

$s \rightsquigarrow y^* \rightsquigarrow v^*$ s.t. dist is <

of $y^* \in V/A$

~~it chosen P at some time~~

~~the one of the intermediate~~ if it would have been argmin in

$\rightarrow$ dw

hypothetical smaller path < path for algo

not path

~~Since~~ since

$\arg\min \; d(u^*) + wt(u^*, v^*)$

$\wedge\!\!|$

$d(y) + wt(x, y)$ ~~+ wt(y, v*)~~

$\Rightarrow d(u^*) + wt(u^*, v^*) \nless d(x) + wt(x, y) + wt(y, v^*)$

$\Rightarrow\Leftarrow$

DFS — stack
BFS — queue
SSSP — priority queue

the item to be dequeued is the minimum.
(w.r.t some key)

Database

Roll No.
. . . . etc.
Key = Roll No.
will get smallest Roll No.

- sorting works?
  - not good for updates
  - suppose ROLL No. i.e. key value is dec.

the key would be currently known shortest path to this node.

## Priority Queue Data Structure

- query a particular item — get key of this item.
- update key to smaller value
- extract the min$^m$ key item

DFS — stack
BFS — queue
SSSP — priority queue

the item to be dequeued is the minimum.
(w.r.t some key)

Database

Roll No.
. . . . etc.
Key = Roll No.
will get smallest Roll No.

- sorting works?
  - not good for updates
  - suppose ROLL No. i.e. key value is dec.

the key would be currently known shortest path to this node.

## Priority Queue Data Structure

- query a particular item — get key of this item.
- update key to smaller value
- extract the min$^m$ key item

Procedure SSSP(G, ~~w~~ s) Dijskhra's algo ~~(in networks)~~

// G = (V, E)

d ← empty array of size |V|

for each v ∈ V

d[v] ← ∞

d[s] ← 0

π ← empty array of size |V|

π[s] ← NULL

~~Inct~~

Include s to A ~~A ←~~ A is "seen" set

PQ ← priority queue of items = V, key defined by array d.

while (A ≠ V)

u ← PQ.extract-min() // d(s,u) is known

for each (u,x) ∈ E

PQ.update.key(x, d(u) + wt(u,x))

only if d(u) + wt(u,x) is smaller than original key

or

if (d(x) > d(u) + wt(u,x))

π[x] ← u

PQ.update-key(x, d(u) + wt(u,x))

endif

endfor

endwhile

or ~~while~~ run a counter from 1 to |V|

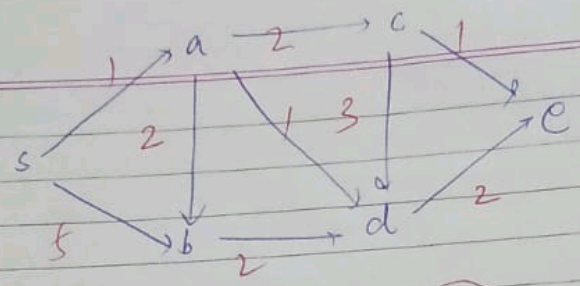
like dequeue or pop

similar to enqueue or push in DFS

really used in networks

PQ will use the d array & thus do updations to d array itself.

PQ has d array inside it




| | |
|---|---|
| ~~0~~ | 0 |
| ~~1~~ | 1 |
| ~~2~~ | 2 |
| ~~3~~ | 3 |
| ~~2~~ | 2 |
| ~~4~~ | 4 |

Let's say naive PQ

A

s, a, d, c

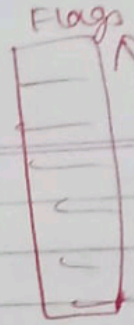
b, e

choose key-value in an array and scan every time to find minimum

~~Naive~~ ~~Naive~~

Naive PQ
↳

Flags & go over PQ & find min



for seen or not

∵ each vertex visited exactly once

while loop ← $\Theta(n)$

extract_min ← $\Theta(n)$

for each $(u,x) \in E$ ← m total ⇒ $\Theta(n^2+m)$

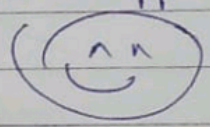
∵ ~~each~~ for visiting edge again

& satisfy > condⁿ

would need to visit a vertex twice

which doesn't happen

(Think!)



Linked List ← necessary

Date :
Page :

Lec-11

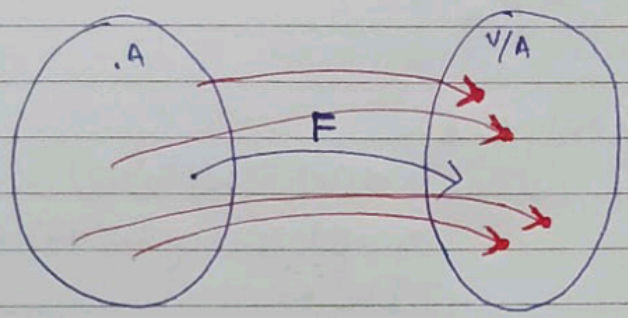
[SUMMARY]

SSSP

- It is possible to use BFS for SSSP after ~~setting~~ converting w edge with wt w to w edges.

(This clearly doesn't work for ~~neg~~ -ve & fractional wts and hurts the space & time complexity in case of large wts)

-




$\underset{(u,v)\in F}{\operatorname{argmin}}\{dist(u) + wt(u,v)\} = (u^*, v^*)$

$A = \{w \mid$ shortest path from s to w has been determined$\}$

Then, $s \rightsquigarrow u^* \rightsquigarrow v^*$ is a shortest path from s to $v^*$

- ~~Sim~~ Procedure SSSP $(G, s)$ Dijkstra's (Assumes non-neg wts)

```
// G = (V, E)
d ← empty array of size |V|
for each v ∈ V
    d[v] ← ∞
endfor
d[s] ← 0

π ← empty array of size |V|
π[s] ← NULL
PQ ← priority queue with items V and keys defined by d

Include s to A
while (A ≠ V) // or run a loop from 1 to |V|
    u ← PQ.extract-min()
    Include u to A

    for each (u,w) ∈ E
        if (d[w] > d[u] + wt(u,w))
            PQ.update-key(w, d[u]+wt(u,w))
            π[w] ← u

        endif
    endfor
endwhile
```

Note PQ contains the d array

- A näive PQ can be a simple array storing key-value pair, and to store min we iterate over each element of array ☺

- Time of naive implementation : $\Theta(n^2+m)$ n: no. of nodes
  if extract_min() is $\Theta(n)$ m: # of edges
  & update key is $\Theta(1)$

~~Wouldn't extract min run for $O(n^2)$ or $O(n\log n)$ (Hag.) for naive implementation of PQ (cuz how to check where in A)~~

- Accessing int* then why log(n) isn't size of address fixed?

Date :
Page :

Lec-12

- Lec 5 summary

Lec-> summary exp. Induction for T.C.

Priority Queue — key / data

- extract-min: removes item with smallest key & returns it

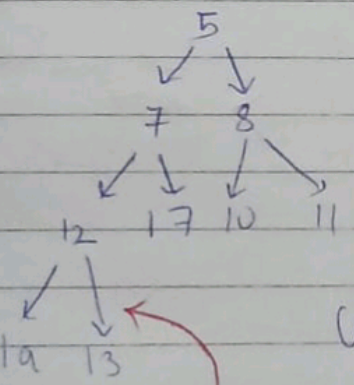
- d(v): stores a vertex → key mapping & when queried, returns the key of the queried vertex.

- decrease-key: reduces key of a ~~an~~ an item.

(Min)

(Binary) Heap → ~~Based on~~ Graph (Binary Tree)

↳ ~~Just an~~ array stored as — but is a




binary Tree

- each vertex is a key
- Every parent has 2 children and every node has got a unique parent — 3 if ternary allowed to violate last level & its parents

Complete Binary Tree

- All but last levels are full (Last can be full too) — if agree with
- Last level is filled ~~to an~~ left to right (no gaps)

If arrows not mentioned they are understood to pt downwards.

- Parent's key is smaller that that of the children (Min)

⇓

root is smallest

We will store it as an array

| 5 | 7 | 8 | 12 | 17 | 10 | 11 | 19 | 13 | |
|---|---|---|---|---|---|---|---|---|---|

~~dist~~ 0 dist 1 . . .

heap-size = 9

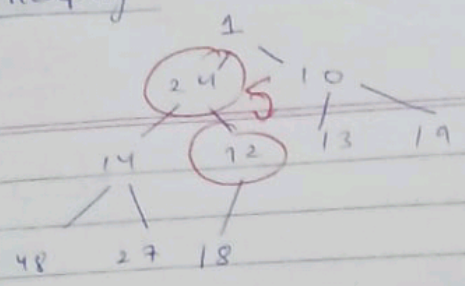
Procedure Left(x)
// x is index of node.
// returns index of the left child ~~of the~~
return $2x$

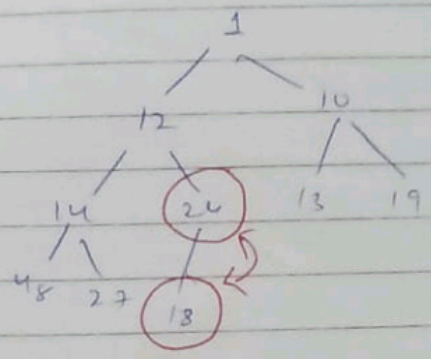
Procedure Right(x)
return $2x+1$

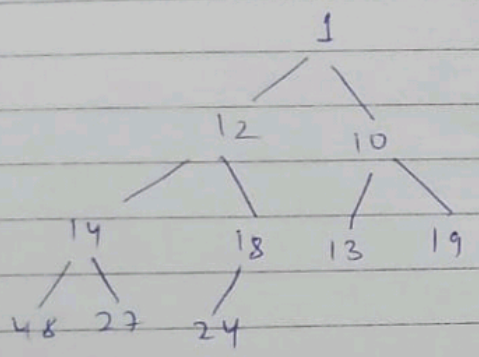
Procedure Parent(x)
return $\left\lfloor \frac{x}{2} \right\rfloor$

{if 0 is returned we ~~get~~ know $x = 1$} $\rightarrow$ we are at ~~x is~~ root.

Min-Heapify




we can make an array of structs

Procedure Min Heapify (A, x)

// A has a violation of heap property at index x.

while (1) {

minIndex ← x;

if (left (x) ≤ heap-size and A [left (x)].key < A[x].Key)

minIndex ← left(x)

end if

if (right (x) ≤ heap-size and A[right(x)].key < A[minIndex].key)

minIndex ← right (x)

end if

$T(l) = T(l-1) + \Theta(1)$

$\Rightarrow \Theta$

if (minIndex ≠ x)

tmp ← A[x]

A[x] ← A[minIndex]

A[minIndex] ← tmp

minHeapify (A, minIndex)

Recursive

else

return

endif

endwhile

Procedure Build-Heap (A, n)

// Array A of size n. (assuming A has valid structure)

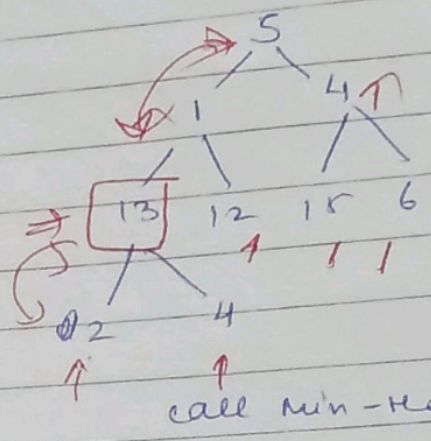
maybe a global variable { heap-size ← n

for int i = n to 1

Minheapify(A, n)

endfor

| 5 | 1 | 4 | 13 | 2 | 15 | 6 | 2 | 4 |



T.C.

Q: ~~How ma~~ ht of a heap with n items?

Let $ht = l \Rightarrow l-1$ levels are full.

$$1 + 2 + 2^2 + \cdots + 2^{l-1} + [1, 2^l]$$

$$= 2^l - 1 + [1, 2^l]$$

$$= [2^l, 2^{l+1} - 1]$$

$$\Rightarrow \quad l \le \log_2 n$$

$$\log_2(n+1) - 1 \le l$$

$$l = \Theta(\log_2 n)$$

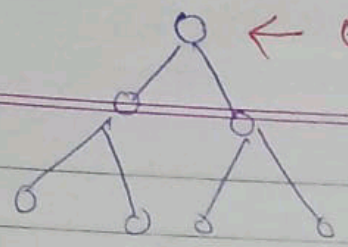
$\therefore$ Min Heapify $\quad T(l) = T(l-1) + \Theta(1)$

$$\Rightarrow T(l) = \Theta(l)$$

$$= \Theta(\log_2 n)$$

T.C. of Build Heap $\Rightarrow$ crude estimate ~~$\Theta(n$~~ $O(n \cdot \log_2 n)$

Let's do calcⁿ more carefully:- $S - 2S = 1 + \sum_{j=2}^{l} 2^{j-1} - 2^l \cdot l - 1$



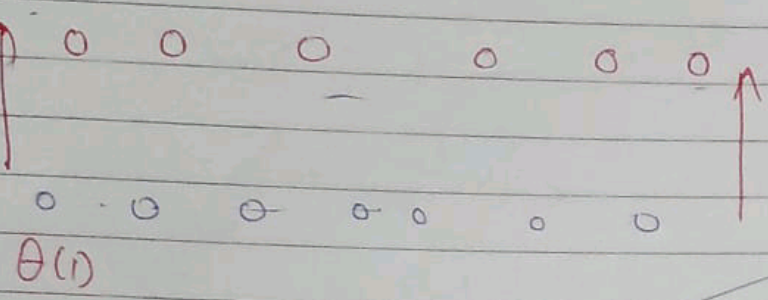
$\leftarrow \Theta(\log_2 n)$

$\sum_{j=1}^{l} 2^{j-1}(l+1) - \sum_{j=1}^{\log n} 2^{j-1} \cdot j$

$(l+1) \cdot \frac{(2^l - 1)}{1} - (2^l - 1 - 2^l \cdot l)$

$= (l+1)(2^l - 1) - (2^l(l-2)+1)$

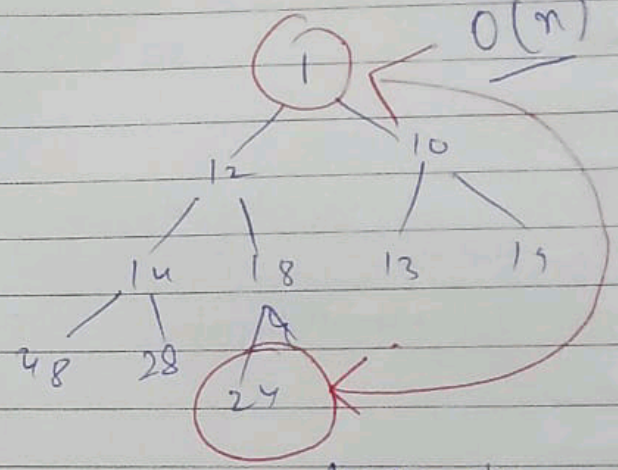
$(l+1)(2^l - 1) - 1$




Ex.

$$C = \sum_{j=1}^{\log n} 2^{j-1}(l-j+1) = \Theta(n)$$

By Product : Heapsort




$O(n)$

heap-size --
call min heapify

Procedure ExtractMin(A)

// Heap A, return min

tmp ← A[1]

A[1] ← A[heap-size]

A[heap-size] ← tmp

heap-size --

Min Heapify(A, 1)

return A[heap-size +1]

$\leftarrow O(\log n)$

~~if (heap-size = 1)~~

~~heap-size --~~

~~return~~

if (heap-size == 0) return error endif

# Heap Sort

Procedure HeapSort (A, n)

B ← BuildHeap (A, n) // O(n)  correctness is obvious

~~heap size ← n~~

for j = 1 to n

print (ExtractMin (B)) // O (n log n)  reason why not O(n)?

- stability?
- space?
- etc.

log n + log n-1 + log (n-2) ... log(1)

= log (n!) ~~≤ log (n^n) = n log n~~

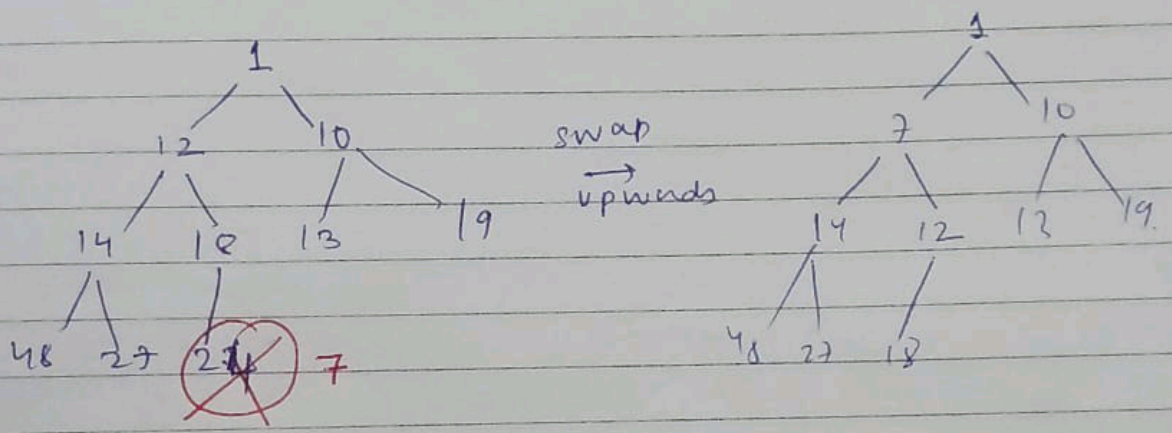
// Stirling's approx

n log(n) − n + O(log n) ⇒ log (n!) = O (n log n)

~~Min log(n!) / n→∞ n log(n^n) =~~

(15)

Decrease-key operation.




Procedure Decrease-Key (A, x, newKey)

~~A[x] ← newkey~~

if x > heap-size return error endif.

// check if A[x].key is already less equal a.

A[x].key ← newkey

while (1)

if (parent(x) = 0) break endif

~~tmp ← A[x]~~

if (A[parent(x)].key < A[x].key) break endif

tmp ← A[x]

A[x] ← A[parent(x)]

A[parent(x)] ← tmp.

x ← parent(x)

endwhile

T.C ← O(log (n))

as ~~goes above each~~

at each iteration it goes up one level.

ity Queue — heap
— mapping: vertex → heap index

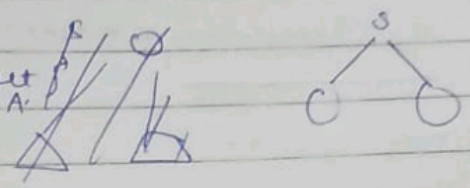
$\infty \equiv \sum w_{ij} + 1$ (in shortest path algo)

d:

| | | | | |
|---|---|---|---|---|
| ∞ | ∞ | | 0 | ∞ |
| 1 | 2 | | s | n |

(source node)

we'll call build heap on d. ⇒ to get heap A.
(ofc. s will be at top)
∵ d[s] = 0



heap is array of structs: {key, vertex #}

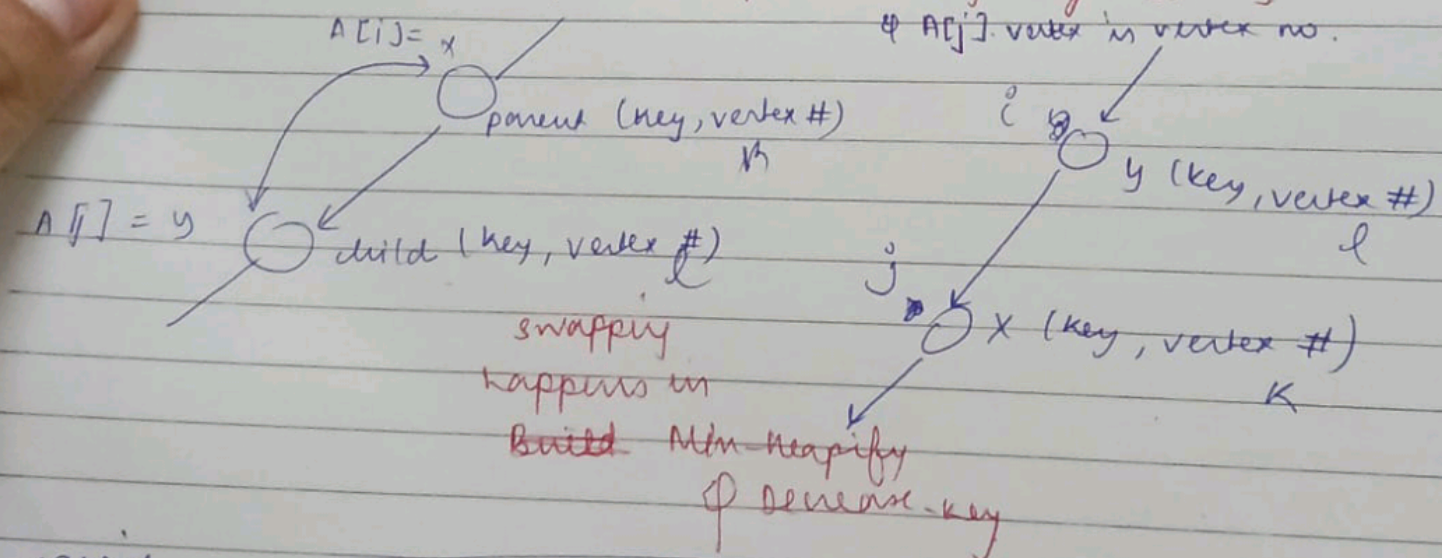
mapping:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | j | 5 | 4 | | 1 | | |
| 1 | 2 | 3 | 4 | . . | 8 | . . . | n |

vertex #

store index in heap

gives the pos° in heap say j then A[j].key is key & A[j].vertex is vertex no.




swapping happens in ~~Build~~ Min-Heapify & Decrease-key

mapping

| | |
|---|---|
| i | j |
| k | l |

mapping

| | |
|---|---|
| j | i |
| k | l |

mapping stores current position of vertex j within heap A.

Ask sir + write d implementation

A[j].key

.vertex# → ~~mapping[A[j].vertex#] ← i~~

extract-min : call extract min of underlying heap.

(after implementing ~~this~~ updating ~~the~~ of mapping after ~~swapping~~ swaps)

decrease-key : call decrease-key of heap.

Ask

~~Procedure~~

PQ → heap A , mapping array

n = |V| , A ← ~~struct~~ array of structs ~~made~~ = {v, d(v)}

PQ.heap ← Buildheap (A, n)

~~mapping array~~ = PQ.mapping ← empty array of size n.

for (i = 1 to n)

PQ.mapping[A[i].vertex number].idx ← i

~~endfor~~

PQ.mapping[A[i].vertex number].key ← A[i].key

endif

Procedure Extract-min ()

if (heap-size = 0) return error endif

x ← A[1].~~key~~ vertex#

key ← A[1] // A[1] ← A[heap-size] // A[heap-size] ← key

PQ.mapping[A[1].vertex no.].idx = 1 // PQ.mapping[A[heap-size].vertex no] = —

~~key = A[1].vertex no.key~~

heap-size --

Min-heapify (A, 1)

return x

In min-heapify & Bubble whenever swap remember to update in PQ.mapping

↳ PQ.mapping[A[x].vertex no].idx = x

PQ.mapping[A[y].vertex no].idx = y

Procedure Decrease-key (i, a) at vertex i

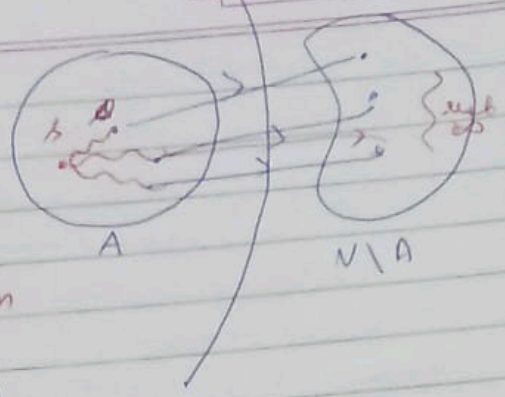
//check i in bounds & val greater than a.

PQ.mapping[i].key = a // A[PQ.mapping[i].idx].key = a

Bubble up (A, PQ.mapping[i].idx)

T.C. of Dijkstra

Procedure SSSP ( . . . . . )




while i=1 to |V|

$\Theta(\log n$ times) → u ← extract-min(A) } n log n

for each (u,w) ∈ E

If (d(w) > d(u) + wt(u,w))

(m time only as u only visited once)

π(w) ← u

~~d(w) ←~~

$\Theta(\log n)$ → decrease_key(A, —, —)

A × V\A

left endpt  right endpt

Time Complexity

$O((m+n)\log n)$  $O(m+n^2)$

$m \le \left(\frac{n}{2}\right) = \Theta(n^2)$

$O((m+n)\log n) \ll O(n^2)$

unless. $m = \omega(n^2/\log n)$

$O(m + n\log n) \ll O(n^2)$

↓

Fibonacci Heap

can also use π array here

Procedure SSSP (G, s)

d ← empty array of size |V| initialised with ∞.

A ← array of ~~structs~~ pairs made of (idx, d[idx])

PQ ← heap ← BuildHeap(A, n)

PQ.mapping ← empty array of size |V|

for i = 1 to n

PQ.mapping[PQ.heap[i].vertex] ← i  inverse

Decrease-key(PQ, s, 0)

for i = 1 to n

u ← Extract-min(PQ)

for each (u,w) ∈ E do

if (wt(u,w))

{ PQ.mapping[u].key

PQ.mapping[w].key

Decrease-key(PQ, u, s)