GRAPHS

Main HG

```
main() {
                                                          EdgeWeighted
    int A = starting;                                        Graph
    int B = destination;                                       ↓
    int C = passing;
    DijkstraSP sp = new DijkstraSP(graph, C);
    double weight = ∞;  ←  since we don't know
    boolean pathExist;     anything about the
                               path
path C → A       Stack<Integer> pathCA;
path A → C → B   Queue<Integer> path;  ← FIFO-queue
    if (sp.hasPathTo(A)){
        pathExists = true;
        for (DirectedEdge e : sp.pathTo(A)){
            pathCA.push(e.from());             → [        C ]
            weight += e.weight();
        }
        pathCA.push(A);                        → [ A ··· C ]
        while (!pathCA.isEmpty()) {
            path.enqueue(pathCA.pop());
        }                                        back         front
    }                                          →  C  ···  A :
    else pathExists = false;
    if(sp.hasPathTo(B) && pathExists){
        for(DirectedEdge e : sp.pathTo(B)){     back              front
            path.enqueue(e.to());              → B ··· C ··· A :
            weight += e.weight();                new!
        }
    }
```

GRAPHS

DirectedEdge

```
int v;        AL
int w;        FL
double weight; 0.34
```

ex

```
AL → FL   0.34
directedEdge (AL, FL, 0.34);
```

```
int to(){
  return w;
}

int from(){
  return v;
}

double weight(){
  return weight;
}
```

creates an object with a vertex v to another vertex w with a weight of weight

... in main

path to A from C : object.from(); C... ↙ -A (not included

path to B from C : object.to(); ↖ ...B -C (not included)

GRAPHS

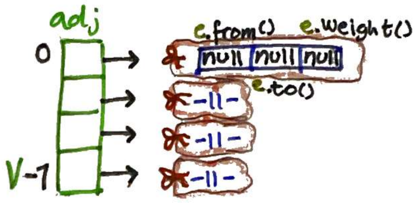
input ex.

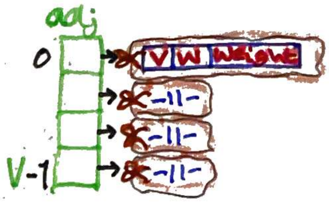
```
4 ←V
4 ←E
1  2  0.1
2  3  0.3
3  4  0.2
4  1  0.1
```

EdgeWeightedGraph

- EdgeWeightedGraph (Scanner in) {

```
    V = in.nextInt();
    adj = (Bag<DirectedEdge>[ ]) new Bag [V];
    for (int v=0; v< V; v++)
        adj [v] = new Bag<>();
    int E = in.nextInt();
    for(int i=0; i< E; i++){
        int v = in.nextInt();
        int w = in.nextInt();
        double weight = in.nextDouble();
        addEdge(new DirectedEdge( v, w, weight));
    }
```





- addEdge(DirectedEdge e) {

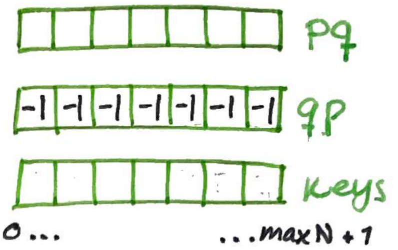
```
    int v = e.from();
    int w = e.to();
    adj[v].add(e);
    E++;
}
```

Check if w is valid, if not: do not add edge

and v ofc

adds to front in Bag

GRAPHS



IndexMinPQ...

ex. no. of vertices

- IndexMinPQ (int maxN) {

```
    this.maxN = maxN;
    n = 0;

    keys = (Key[]) new Comparable [maxN+1];
    pq   = new int [maxN+1];
    qp   = new int [maxN+1];

    for(int i=0; i<=maxN; i++)

        qp[i] = -1;
}
```

size of pq

to initilize keys[i] as non existing

- Insert (int i, Key key) {

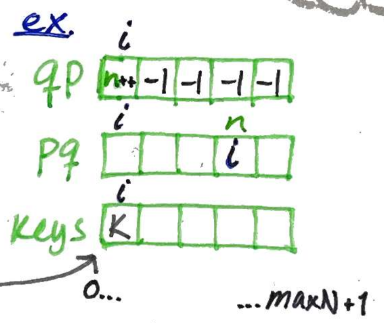
```
    if(contains(i)) trow new Exception
    n++;
    qp[i] = n;
    pq[n] = i;

    keys[i] = key;
    swim(n);
}
```

⇒ if qp[i] != -1

(else) if qp[i] is equal to -1

climb if necessary



- ChangeKey (int i, Key key) {

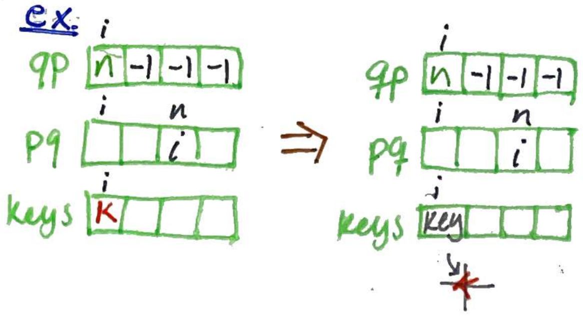
```
    if (!contains(i)) trow new Exception

    keys [i] = key;

    swim (qp[i]);

    sink (qp[i]);
}
```

climb or sink if necessary



GRAPHS

input ex

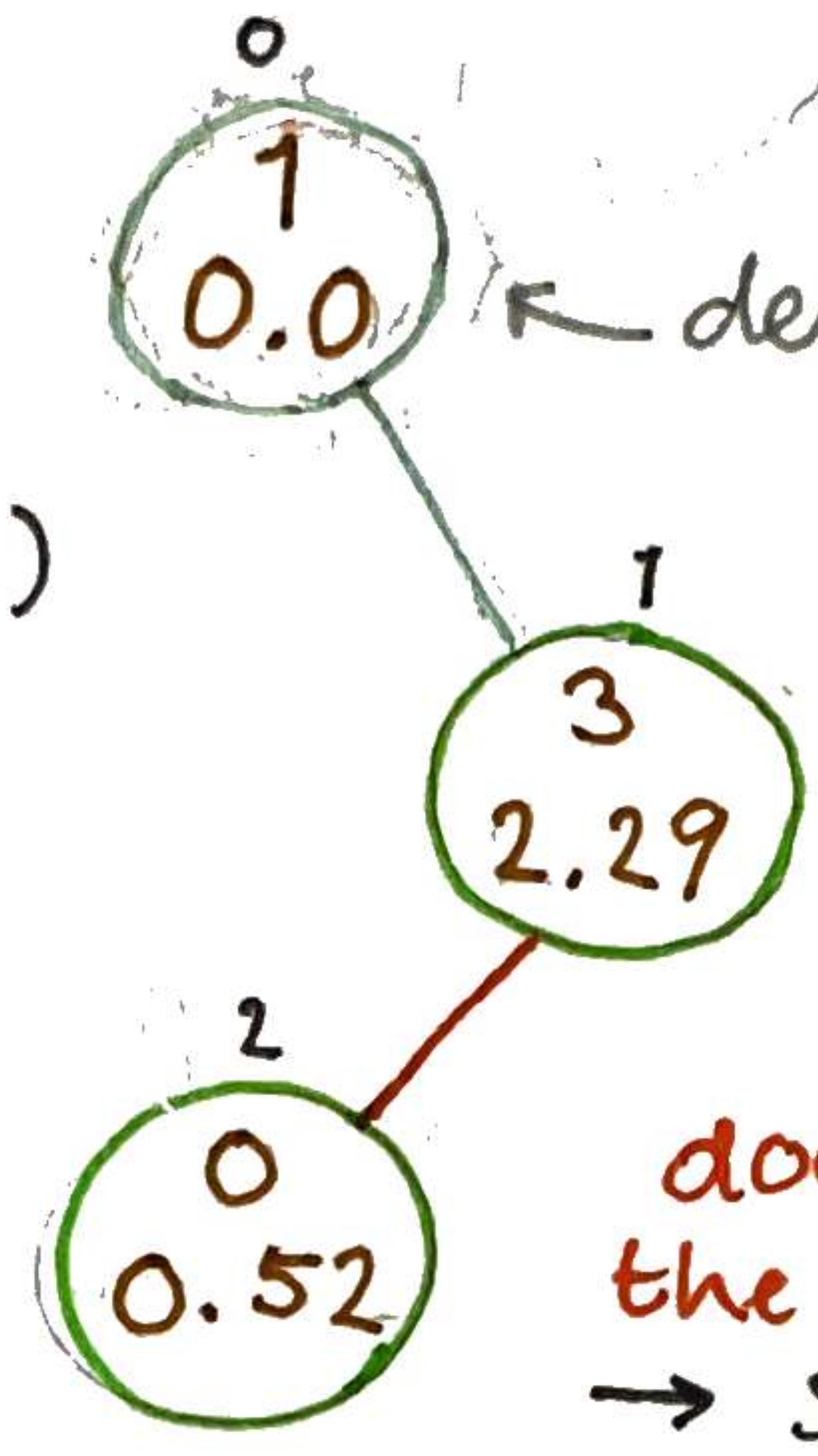
1	3	2.29
1	0	0.52
0	2	0.40
2	3	0.34

not reachable

insert() ↻

	0	1	2	3	4
pq	1	3	0		
qp	2	-1		1	

(n)



← delMin()

Keys

0	0.52
1	~~0.0~~
2	0.92
3	2.29

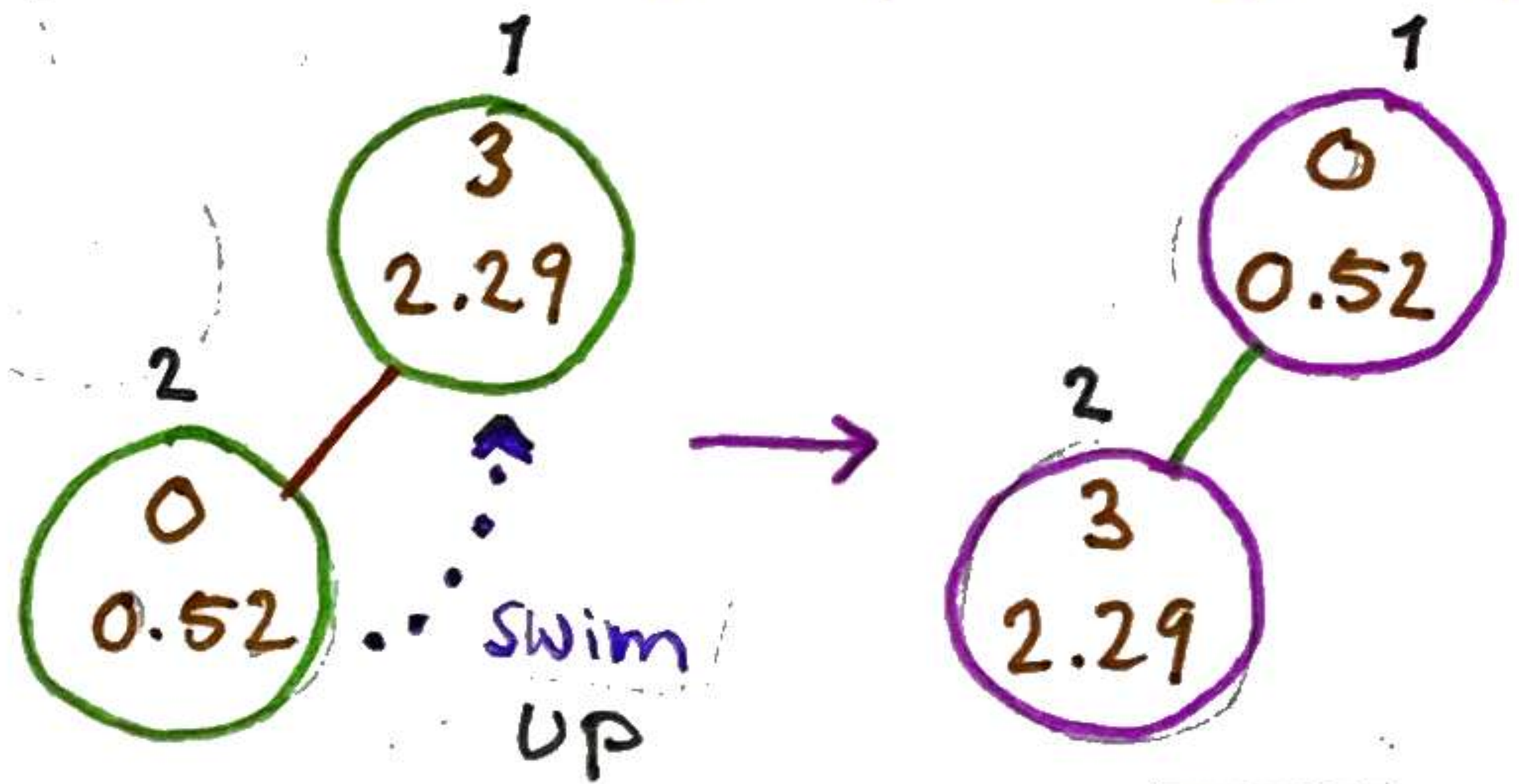
= source
delMin() ⇒ null

does not keep the heap satisfied: 2.29 > 0.52
→ swim(2);

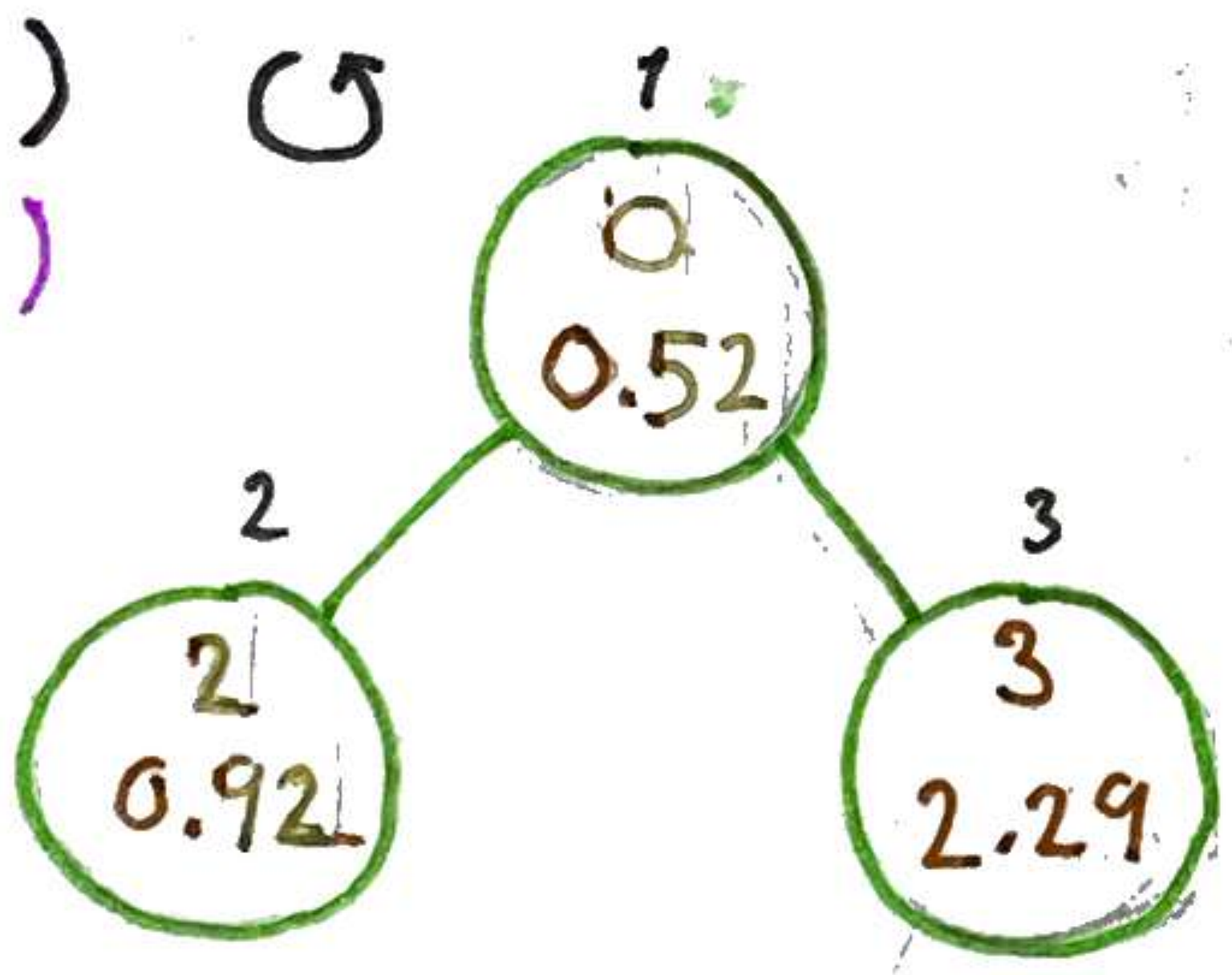
...

```
• Swim (int K) {
      While (K>1 && Keys[pq[K/2]] > Keys[pq [K]]) {
          swap(K, K/2);
          K = K/2;
      }
  }
```

→ K = 2, K/2 = 1 } pq[2] = 0, pq[1] = 3 } Keys[0] = 0.52, Keys[3] = 2.29



insert() ↻
swap()



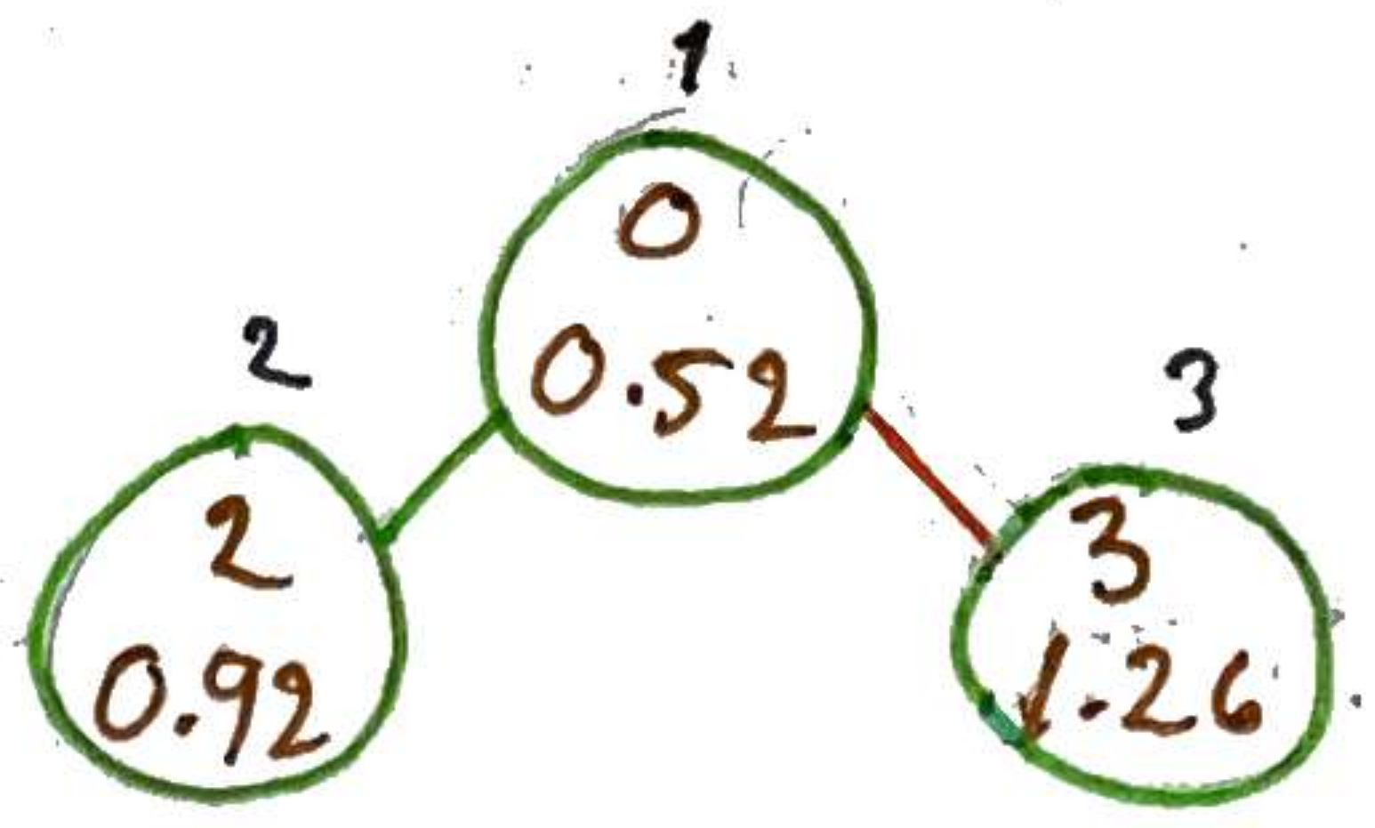
swap updates

	0	1	2	3	...
pq	1	0	2	3	
qp	1	-1	2	3	

insert(3, 1.26) → Exception ← "3" is already in queue

```
instead
→ decreaseKey(3, 1.26) {
      keys[3] = 1.26;
      swim(qp[3]);
  }
```

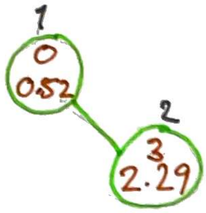
in this case not necessary



	0	1	2	3	...
Keys	0.52	/	0.92	1.26	

- swim() divides parameter by 2 to find parent node

GRAPHS

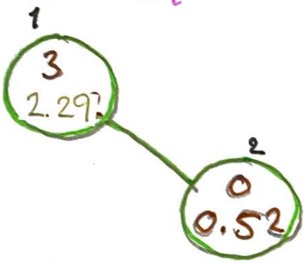


	0	1	2	3	...
pq	/	0	3		
qp	1	/		2	
keys	0.52	/		2.29	

```
delMin() {
    swap(1, n)
    n--;

    sink(1)
    qp[pq[1]] = -1;
    keys[pq[1]] = null;
    pq[n+1] = -1;
}
```

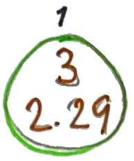
swap()



	0	1	2	3	...
pq	/	3	0		
qp	2	/		1	
keys	0.52	/		2.29	

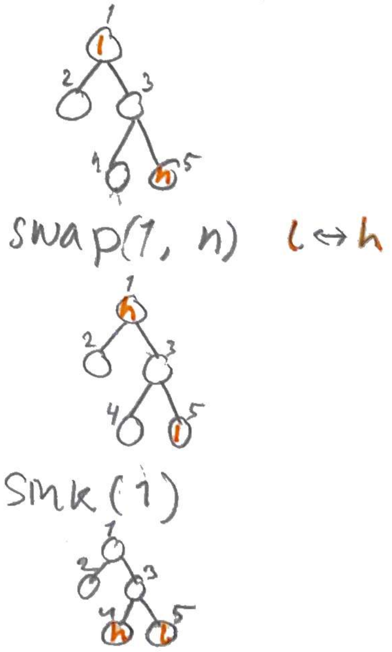
...delete values...

	0	1	2	3	...
pq	/	3	-1		
qp	-1	/		1	
keys	null	/		2.29	



```
sink(int k) {
    while (2·k <= n) {    2·1 > n
        ...
    }
}
```

ex when sink operates:



swap(1, n) l↔h

sink(1)

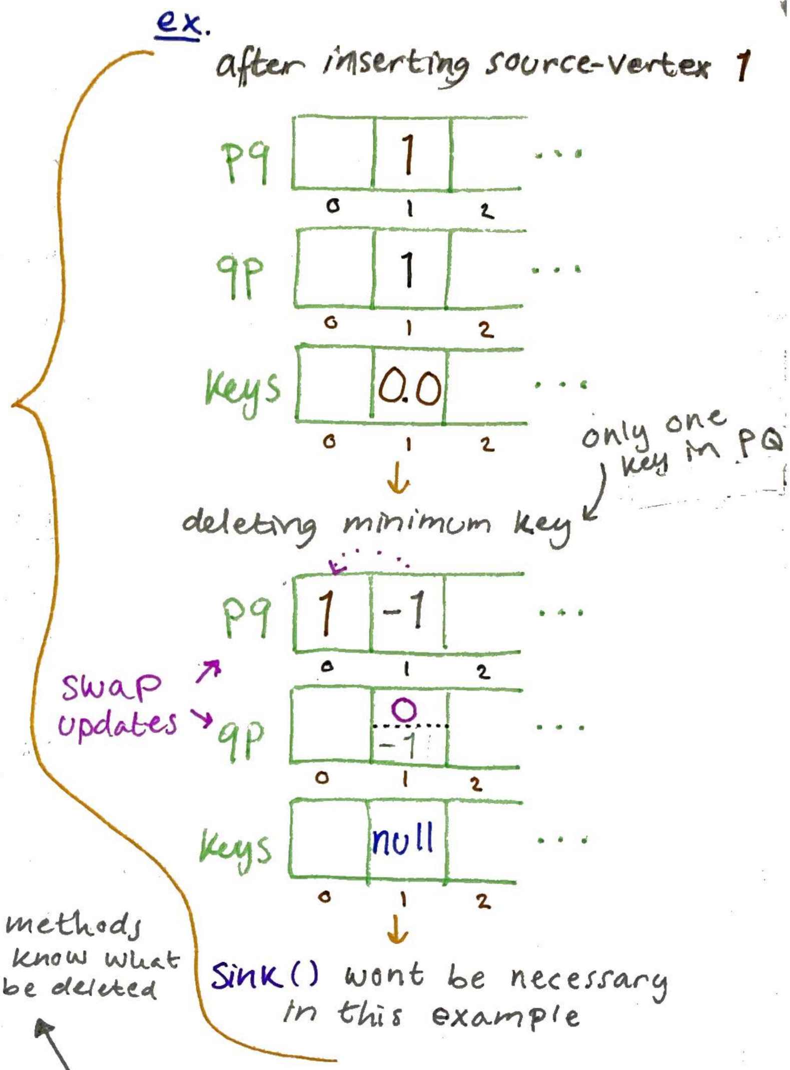
GRAPHS

...

```
• delMin() {
      int min = pq[1];
      swap(1, n--);
      sink(1);

      qp[min] = -1;
      Keys[min] = null;
      pq[n+1] = -1;

      return min;
}
```



```
• delete(int i) {
      int index = qp[i];
      swap(index, n--);

      swim(index);  }
      sink(index);  }

      Keys[i] = null;
      qp[i] = -1;

}
```

calling both methods since we don't know what vertex/key is to be deleted

```
• changeKey(int i, Key key) {
      Keys[i] = key;

      swim(qp[i]);  }
      sink(qp[i]);  }
```

calling both...

GRAPHS

Dijkstra SP

```
DijkstraSP (EdgeWeightedGraph G , int s){
   for (DirectedEdge e : G.edges())
       check so no edge is negative ↻
   distTo = new double[G.V()];
   edgeTo = new DirectedEdge[G.V()];
   for (int v=0; v < G.V(); v++)
      distTo[v]=Double.POSITIVE_INFINITY;
   distTo[s] = 0.0;

   pq = new IndexMinPQ<>(G.V());
   pq.insert(s, distTo[s]);
   while (!pq.isEmpty()) {
      int v = pq.deleteMin();
      for (DirectedEdge e : G.adj(v))
          relax(e);
   }
}
```

relax all edges in graph

	distTo
0	∞
s	0.0
⋮	⋮
V-1	∞

GRAPHS

DijkstraSP

only executes if we find a shorter path

```
relax (DirectedEdge e) {
    int v = e.from() , w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

	...	s	...	3	4	
distTo	∞	0.0	∞	∞	∞	...

→ e = $\overset{v}{s} \rightarrow \overset{w}{4}$ $\overset{\text{weight}}{0.1}$

distTo[w] > distTo[v] + e.weight()

∞ 0.0 + 0.1

	...	s	...	3	4	
distTo	∞	0.0	∞	∞	0.1	...

→ distTo[w] = distTo [v] + e.weight()

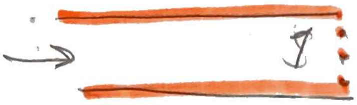
if vertex w is already in IndexMinPQ then just update the key with decreaseKey()-method.

if vertex w is not already in IndexMinPQ then insert vertex as key with insert()-method

quick example of
DijkstraSP{} + relax()

1	3	2.29
1	0	0,52
0	2	0.40
2	3	0.34

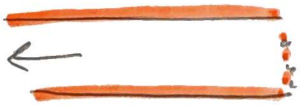
s=1



pq.insert(s) n = 1

while-loop (1) queue ej tom
delMin()

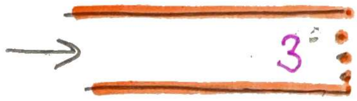
V = 1



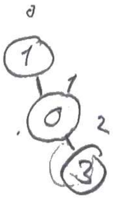
pq.delMin() n = 0

V.adj() : 3 , 0

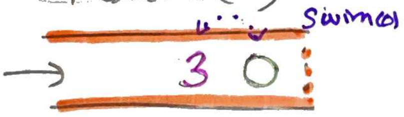
relax()



n = 1



relax()



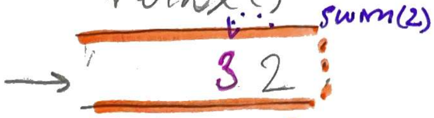
n = 2
swap(1 ; n--)
1

while (2) queue ej tom
delMin()

V = 0

n = 1

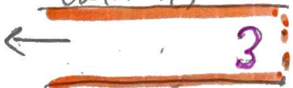
V.adj() : 2
relax()



n = 2

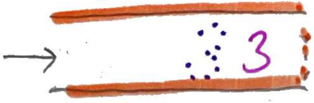
while (3) queue ej tom
delMin()

v = 2



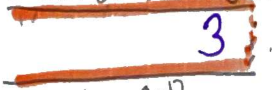
n = 1

V.adj : 3
relax



3 already in queue

decreasekey(),



n = 1

while (4) ⟶ V.adj() : — , end of while
delMin()

ex.

1 → 3	2.29
1 → 0	0.52
0 → 2	0.40
2 → 3	0.34

DijKstraSP

1. pq. insert (1 , 0.0)

	0	1	2	3
distTo		0.0		
pq		1		
qp		1		
Keys		0.0		

n = 1

pq [n = 1] = 1

qp [1] = 1 = n

Keys [1] = 0.0

2. while $\boxed{v}$ = pq. deleteMin → min = pq[1] = $\boxed{1}$

swap (1, n-- = 0)

after swap - now = 1

n = 0

sink (1)

→ if (2 <= 0) → NO

qp [1] = -1

Keys [1] = null

pq [0+1] = -1

	0	1	2	3
distTo		0.0		
pq		-1		
qp		-1		
Keys		null		

3. for adj to v → relax

1) 3 → v = from = 1
w = to = 3

if (∞ > 0.0 + 2.29) → YES

dist [3] = 0.0 + 2.29

edge [3] = (1 3 2.29)

if pq contains(3) → NO

→ pq insert (3 , 2.29)

n ++ = 0+1 = 1

pq [1] = 3

qp [3] = 1

Keys [3] = 2.29

	0	1	2	3
distTo	0.52	0.0		2.29
pq		3	0	
qp	2	-1		1
Keys	0.52	null		2.29

2) 0 → v = from = 1
w = to = 0

if (∞ > 0.0 + 0.52) → YES → dist [0] = 0.0 + 0.52

edge [0] = (1 0 0.52)

if pq.contains (0) → NO → pq insert (0 , 0.52) n ++ = 2

pq [2] = 0

qp [0] = 2

Keys[0] = 0.52

↓

swim(n = 2) → if (2 > 1 && Keys[3] > Keys[0]) → YES
                          2.29   0.52

swap(2, 1)

K = 2/2 → if (1 > 1) → NO (break)

	0	1	2	3
distTo	0.52	0.0		2.29
pq		0	3	
qp	1	-1		2
Keys	0.52	null		2.29

pq ← priority queue - input (vertices)

qp index of pq - contains - to change priority

Keys ← input

4. while (v)= pq deleteMin → min = pq[1] = (0)

	0	1	2	3
distTo	0.52	0.0		2.29
pq		3	~~0~~ -1	
qp	~~2~~ -1			1
Keys	~~0.52~~ null	null		2.29

swap(1, n = 2)

n-- = 1

sink(1)
→ if (2 <= 1) → NO

qp [0] = -1
Keys [0] = null
pq [1 + 1] = -1

5. for adj to v → relax

1) 2 → v = from = 0
          w = to = 2

	0	1	2	3
distTo	0.52	0.0	0.92	2.29
pq		3	2	
qp	-1		2	1
Keys	null	null	0.92	2.29

swap

	0	1	2	3
pq		2	3	
qp			1	2

if (∞ > 0.52 + 0.40) → YES

dist [2] = 0.52 + 0.40 = 0.92
edge [2] = (0 2 0.40)

if pq contains(2) → NO
→ pq insert (2, 0.92)
n ++ = 1 + 1 = 2
pq [2] = 2
qp [2] = 2
Keys [2] = 0.92

swim(2) → if(2 > 1 &&
-Keys[3] > Keys[2]) → YES
   2.29   0.92
swap(2, 1) : K = 1 → break

6. while v = pq.deleteMin → min = pq[1] = 2

	0	1	2	3
distTo	0.52	0.0	0.92	2.29
pq		3	~~2~~-1	
qp			~~2~~-1	1
Keys			~~0.92~~ null	2.29

swap (1, n = 2)

n-- = 1

sink (1)
→ if (2 <= 1) → NO

qp [2] = -1
Keys [2] = null
pq [1+1] = -1

7. for adj to v → relax

1) 3 → v = from = 2
w = to = 3

if (2.29 > 0.92 + 0.34)
1.26

dist [3] = 1.26
edge [3] = (2 3 0.34)

if pq contains (3) → YES
→ pq decreaseKey(3, 1.26)

Keys [3] = 1.26

swim (qp[3] = 1)
→ if (1 > 1) → NO

	0	1	2	3
distTo	0.52	0.0	0.92	1.26
pq		3		
qp	-1	-1	-1	1
Keys	null	null	null	1.26

8. while v = pq.deleteMin → min pq[1] = 3

	0	1	2	3
distTo	0.52	0.0	0.92	1.26
pq		~~3~~-1		
qp				~~1~~-1
Keys				~~1.26~~ null

swap (1, n = 1)

n-- = 0

sink (1)
→ if (2 <= 0) → NO

qp [3] = -1
Keys [3] = null
pq [0 + 1] = -1

9. EXIT loop - no more elements in IndexMinPQ

0	1	2	3
0.52	0.0	0.92	1.26

← vertices

→ distTo - all vertices from source 1

↑ source