SHORTEST PATH

SHORTEST PATH

- Shortest path problem: finding a path between two vertices in a graph such that the sum of the weights of its edges is minimized
- Dijkstra algorithm
- Bellman-Ford algorithm
- A* search
- Floyd-Warshall algorithm

Dijkstra algorithm

- It was constructed by computer scientist Edsger Dijkstra in 1956
- Dijkstra can handle positive edge weights !!! // Bellman-Ford algorithm can have negative weights as well
- Several variants: it can find the shortest path from A to B, but it is able to construct a shortest path tree as well → defines the shortest paths from a source to all the other nodes
- This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights

Dijkstra algorithm

- Dijkstra's algorithm time complexity: **O(V*logV + E)**
- Dijkstra's algorithm is a greedy one: it tries to find the global optimum with the help of local minimum → it turns out to be good !!!
- It is greedy → on every iteration we want to find the minimum distance to the next vertex possible → appropriate data structures: heaps (binary or Fibonacci) or in general a priority queue

Dijkstra algorithm: pseudocode

```
class Node

    name
    min_distance
    Node predecessor
```

Dijkstra algorithm: pseudocode

```
function DijkstraAlgorithm(Graph, source)

    distance[source] = 0
    create vertex queue Q

    for v in Graph
        distance[v] = inf
        predecessor[v] = undefined  // previous node in the shortest path
        add v to Q

    while Q not empty
        u = vertex in Q with min distance // this is why to use heaps !!!
        remove v from Q

        for each neighbor v of u
            tempDist = distance[u] + distBetween(u,v)
            if tempDist < distance[v]
                distance[v] = tempDist
                predecessor[v] = u

    return distance[]  // contains the shortest distances from source to other nodes
```

Dijkstra algorithm: pseudocode

```
function DijkstraAlgorithm(Graph, source)

    distance[source] = 0
    create vertex queue Q

    for v in Graph
        distance[v] = inf
        predecessor[v] = undefined  // previous node in the shortest path
        add v to Q

    while Q not empty
        u = vertex in Q with min distance // this is why to use heaps !!!
        remove v from Q

        for each neighbor v of u
            tempDist = distance[u] + distBetween(u,v)
            if tempDist < distance[v]
                distance[v] = tempDist
                predecessor[v] = u

    return distance[]  // contains the shortest distances from source to other nodes
```

Initialization phase: distance from source is 0, because that is the starting point. All the other nodes distances are infinity because we do not know the distances in advance

Dijkstra algorithm: pseudocode

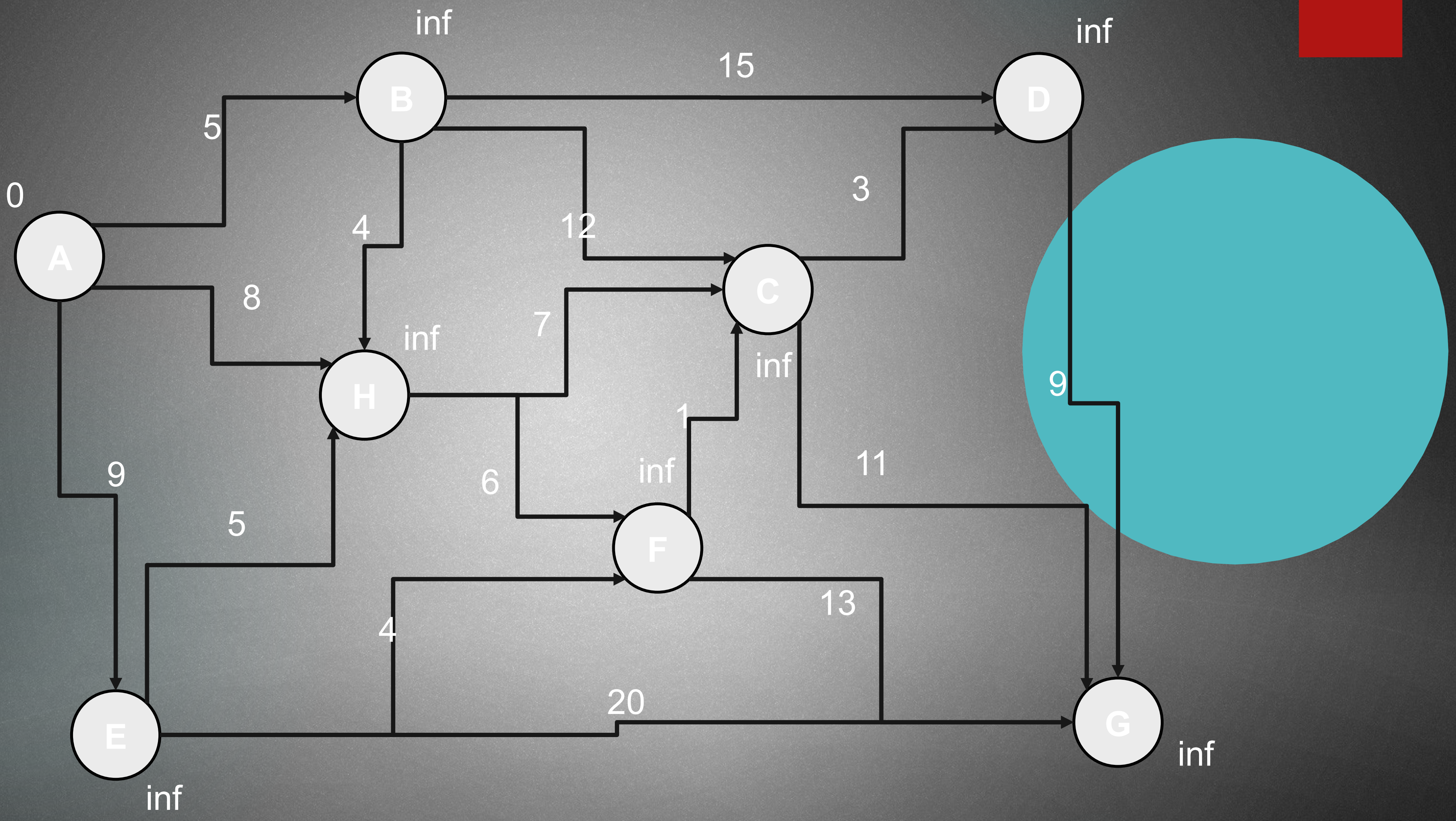
```
function DijkstraAlgorithm(Graph, source)

    distance[source] = 0
    create vertex queue Q

    for v in Graph
        distance[v] = inf
        predecessor[v] = undefined  // previous node in the shortest path
        add v to Q

    while Q not empty
        u = vertex in Q with min distance // this is why to use heaps !!!
        remove u from Q

        for each neighbor v of u
            tempDist = distance[u] + distBetween(u,v)
            if tempDist < distance[v]
                distance[v] = tempDist
                predecessor[v] = u

    return distance[]  // contains the shortest distances from source to other nodes
```

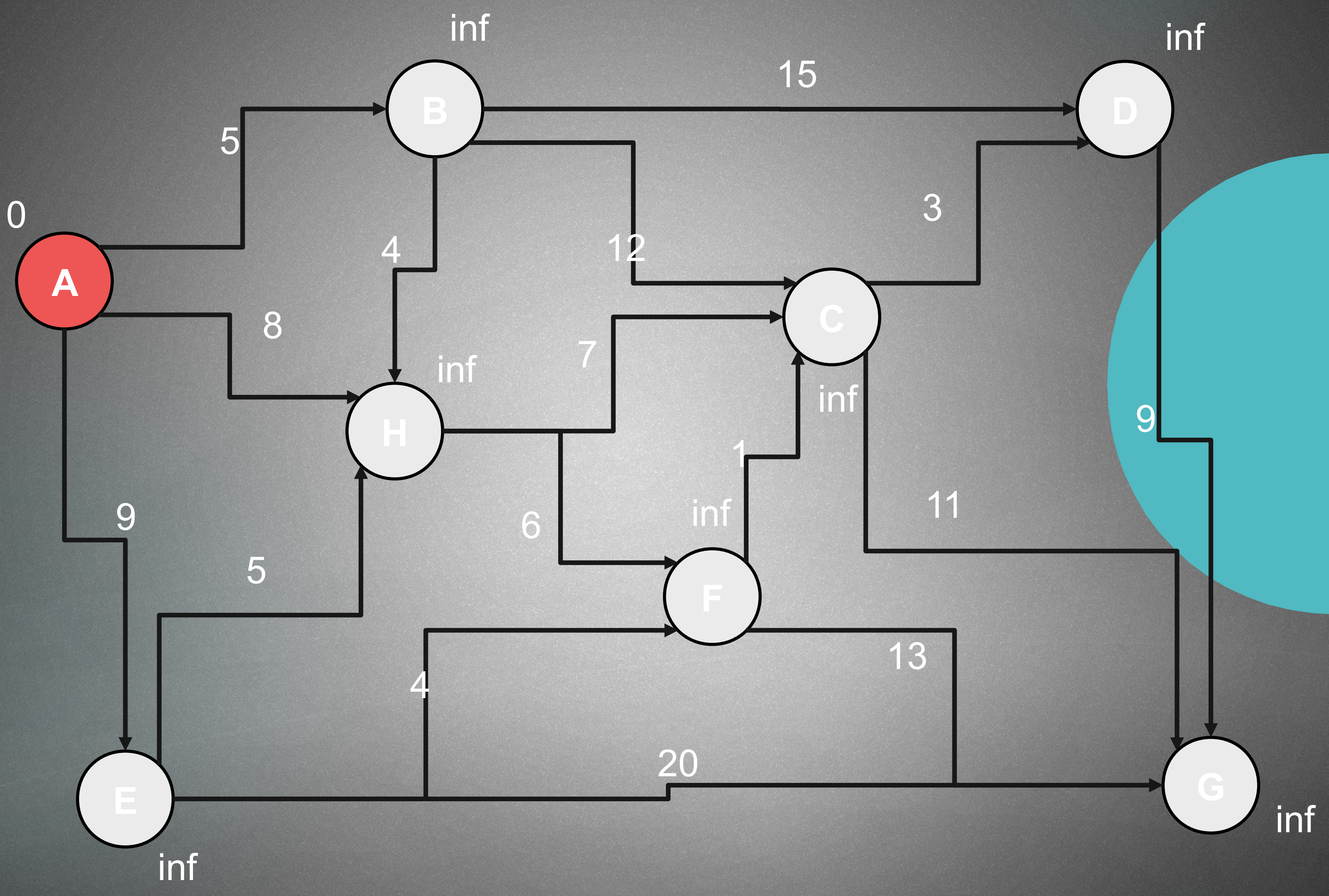
Dijkstra algorithm: pseudocode

```
function DijkstraAlgorithm(Graph, source)

    distance[source] = 0
    create vertex queue Q

    for v in Graph
        distance[v] = inf
        predecessor[v] = undefined  // previous node in the shortest path
        add v to Q

    while Q not empty
        u = vertex in Q with min distance // this is why to use heaps !!!
        remove u from Q

        for each neighbor v of u
            tempDist = distance[u] + distBetween(u,v)
            if tempDist < distance[v]
                distance[v] = tempDist
                predecessor[v] = u

    return distance[]  // contains the shortest distances from source to other nodes
```

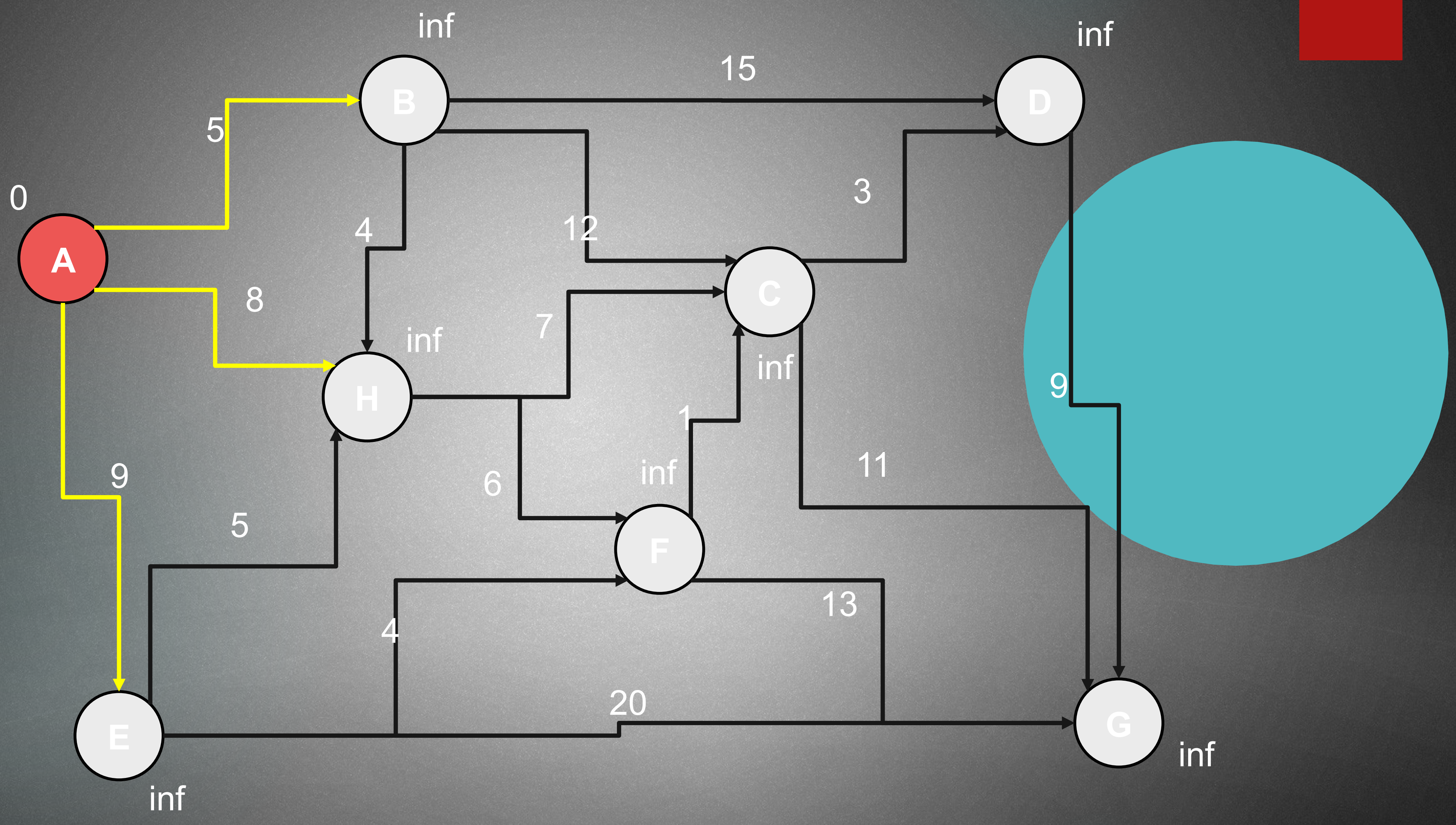
```
function DijkstraAlgorithm(Graph, source)

    distance[source] = 0
    create vertex queue Q

    for v in Graph
        distance[v] = inf
        predecessor[v] = undefined  // previous node in the shortest path
        add v to Q

    while Q not empty
        u = vertex in Q with min distance // this is why to use heaps !!!
        remove u from Q

        for each neighbor v of u
            tempDist = distance[u] + distBetween(u,v)
            if tempDist < distance[v]
                distance[v] = tempDist
                predecessor[v] = u

    return distance[]  // contains the shortest distances from source to other nodes
```

Initialize → source vertex distance is 0, all the other vertex have infinity distance from the source





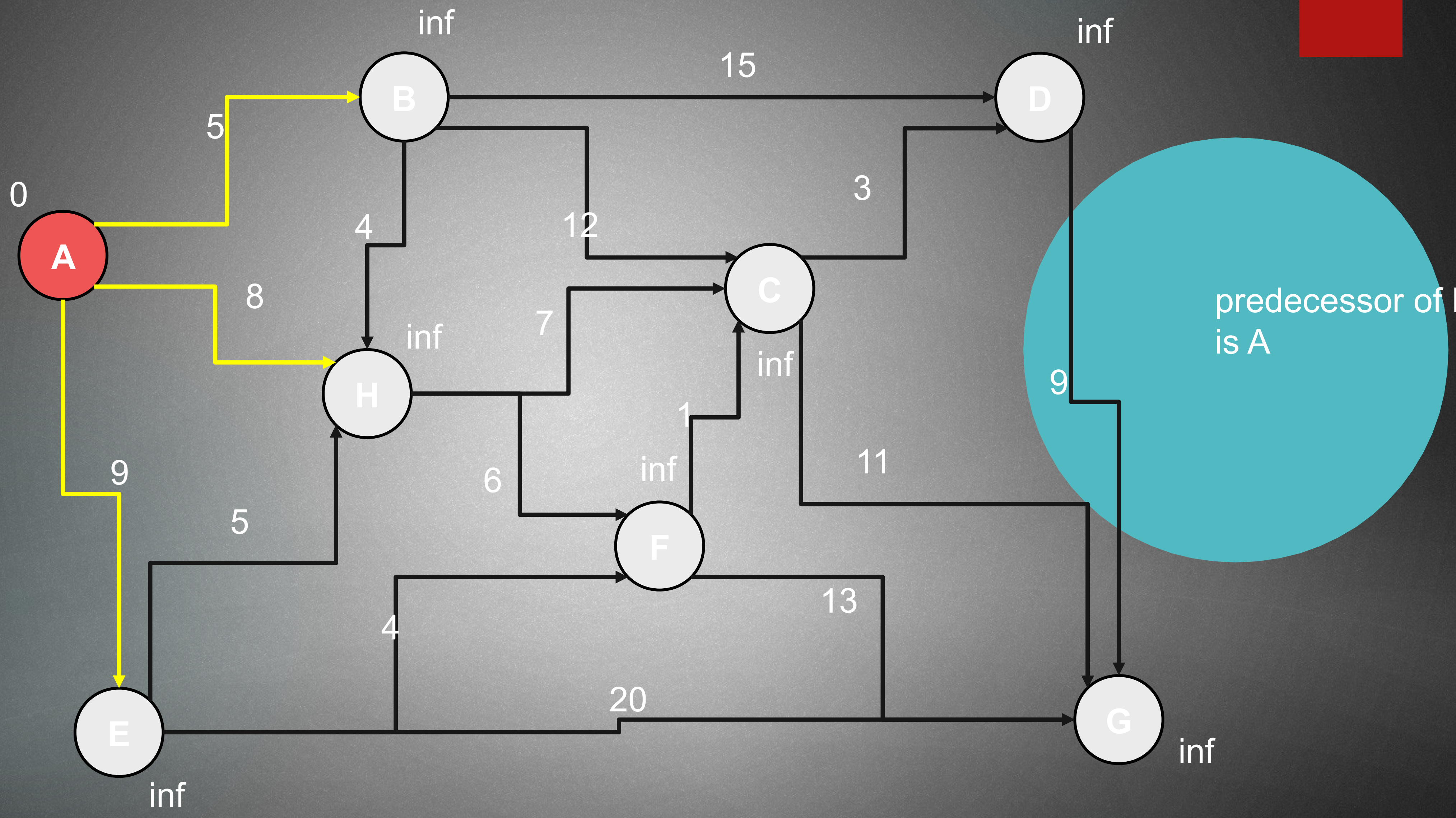
0
A
5
inf
B
15
inf
D
4
12
3
8
C
7
inf
H
inf
9
1
11
9
inf
6
5
F
13
4
20
E
G
inf
inf

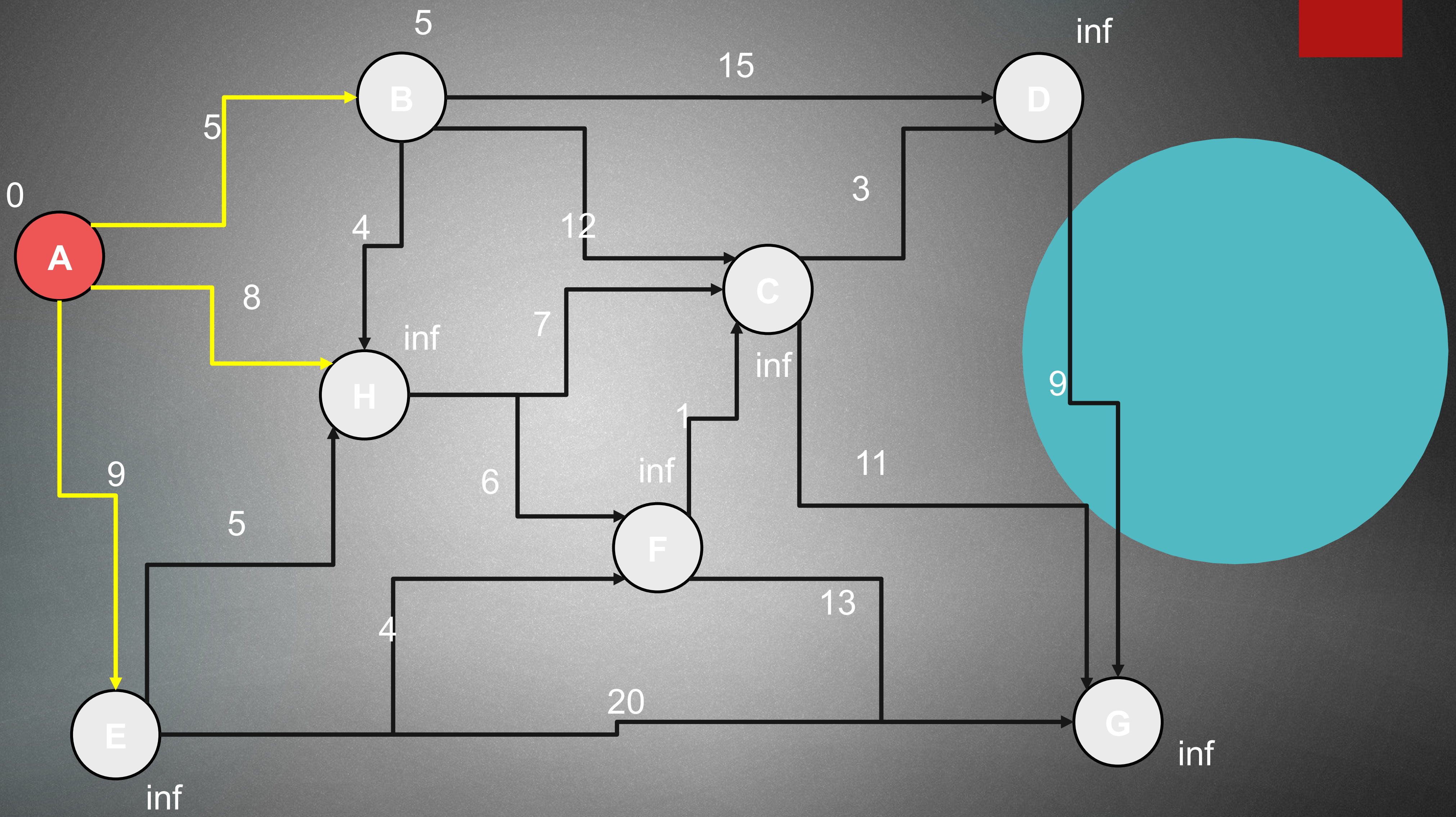


inf
inf
15
B
D
5
3
0
4
12
A
C
8
7
inf
inf
H
9
1
9
inf
11
6
5
F
13
4
20
E
G
inf
inf

Node B: decide what is smaller 0+5 or inf ... 5 is smaller so UPDATE

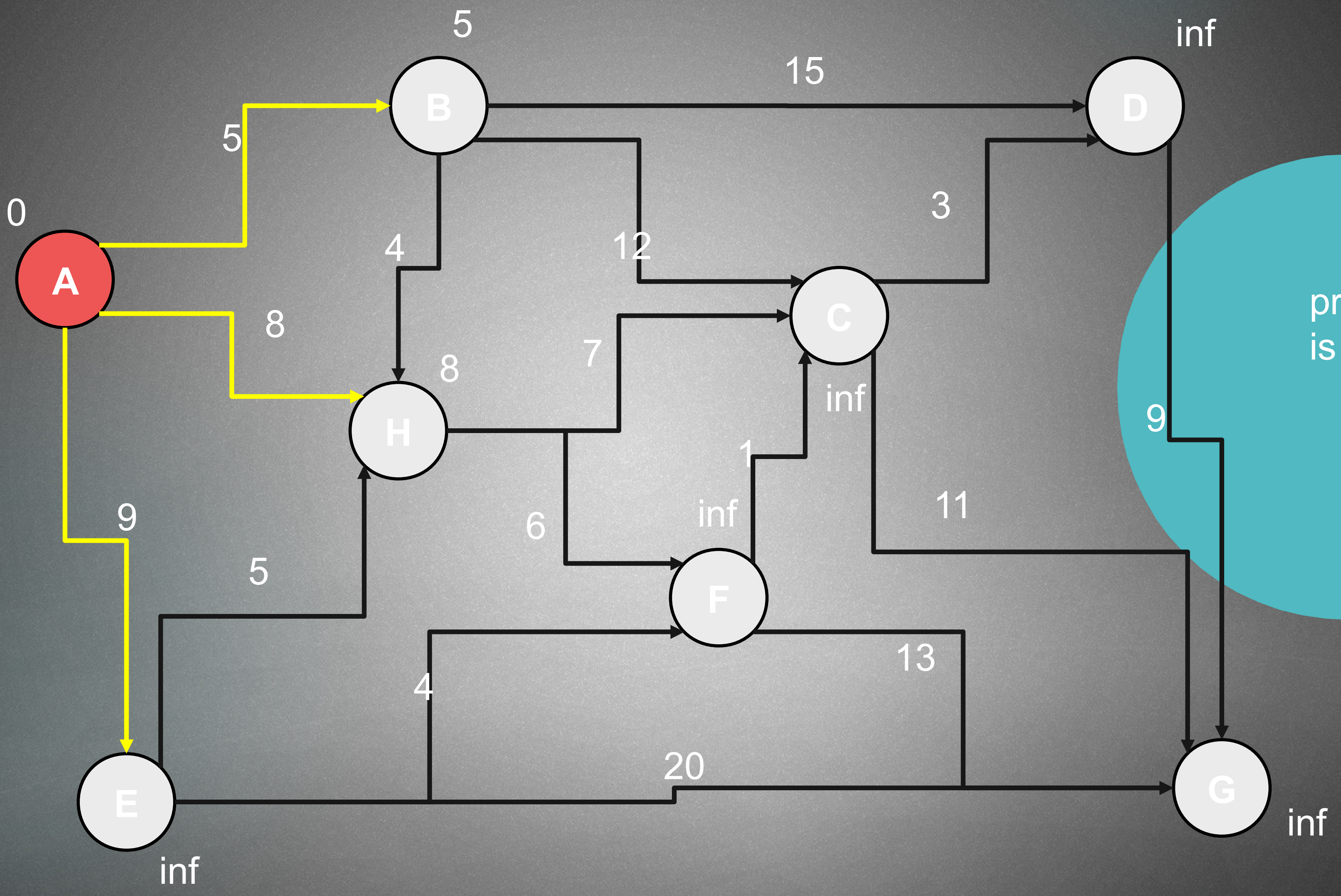
+ we have to track predecessor when we update (if we do not update, we don't)





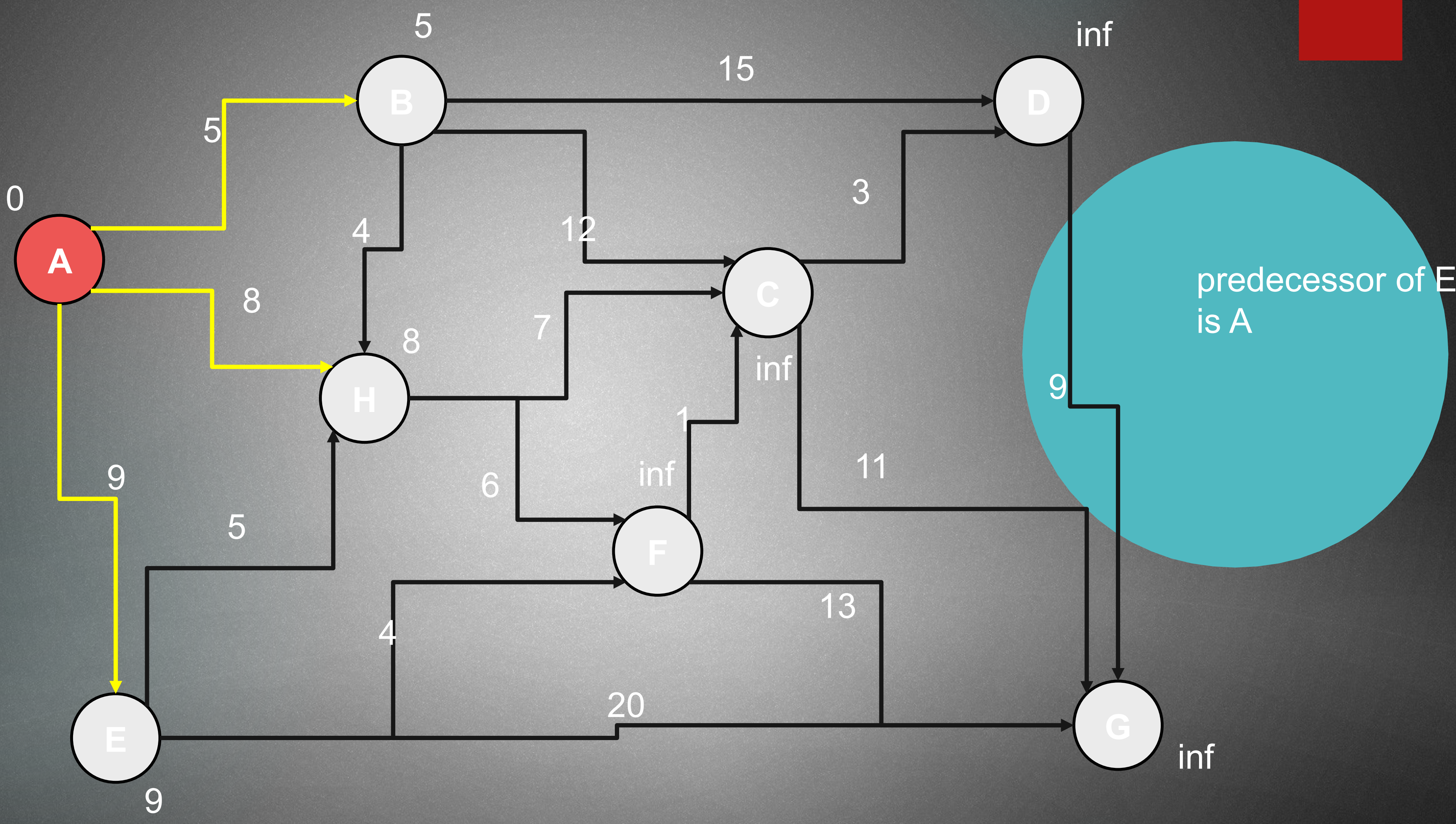
5
inf
15
B
D
5
0
4
12
3
A
C
8
7
inf
inf
H
9
1
9
inf
11
6
5
F
13
4
20
E
G
inf
inf

Node H: decide what is smaller 0+8 or inf ... 8 is smaller so UPDATE

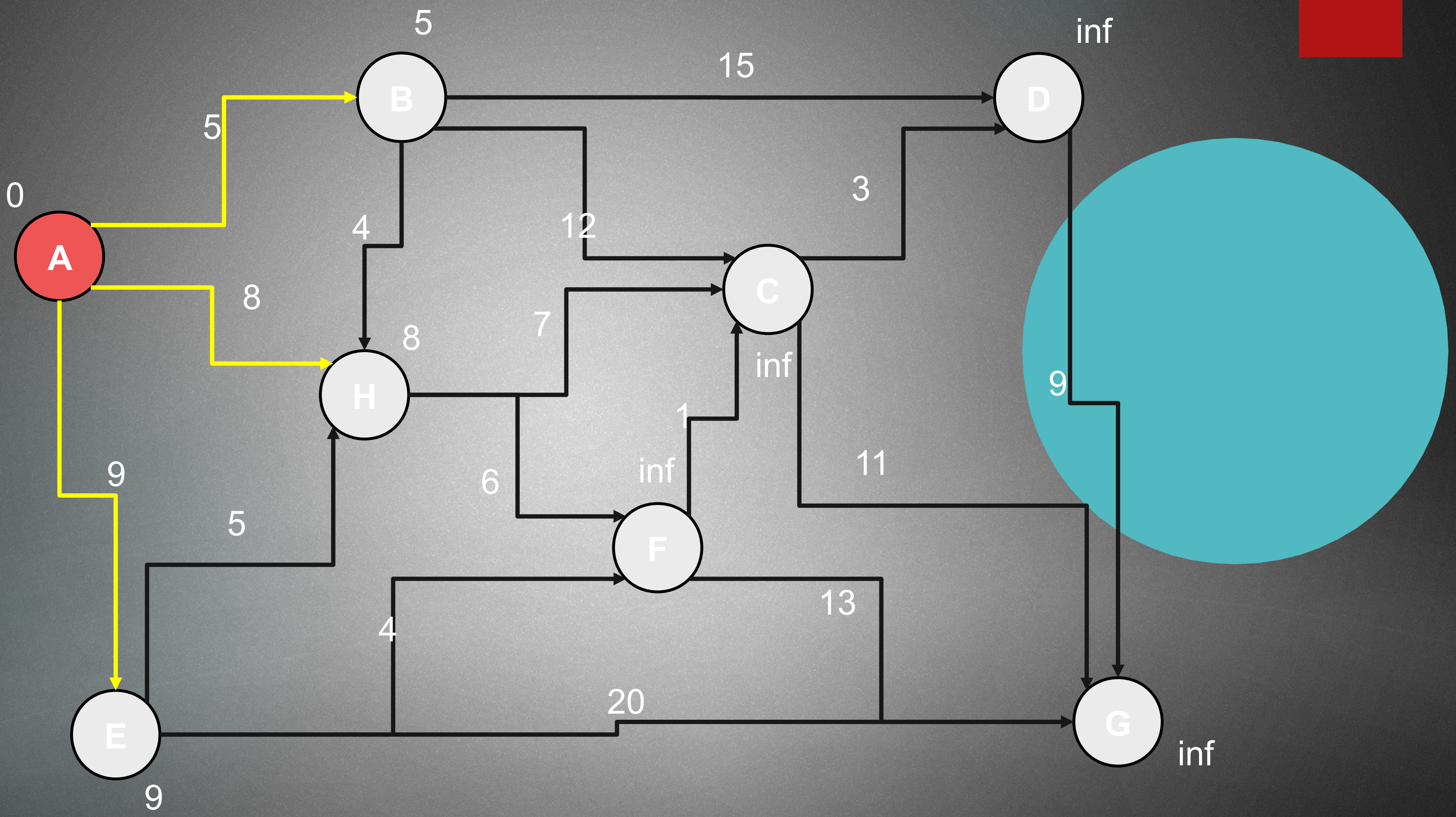


predecessor of H is A

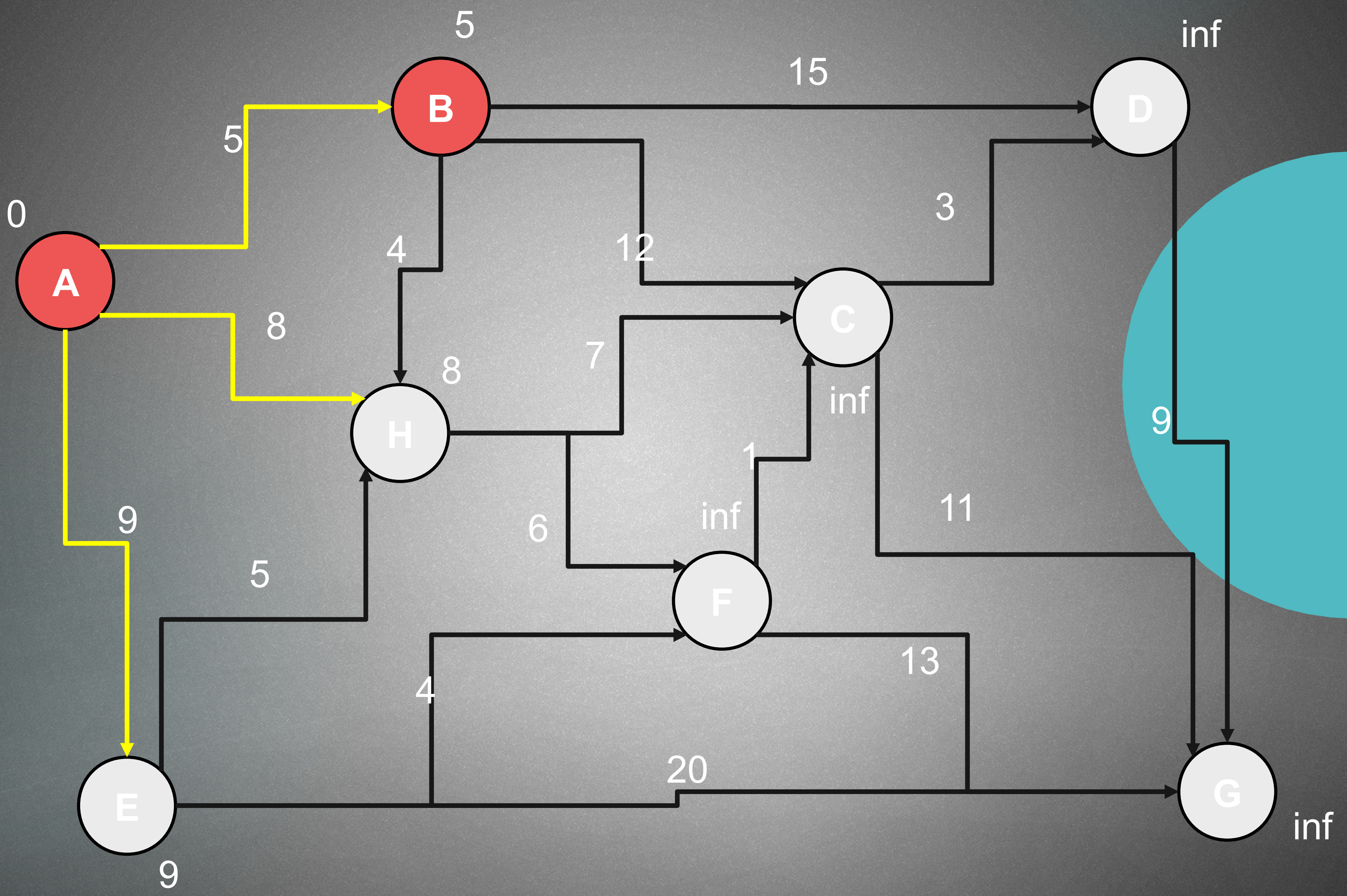
Node E: decide what is smaller 0+9 or inf ... 9 is smaller so UPDATE



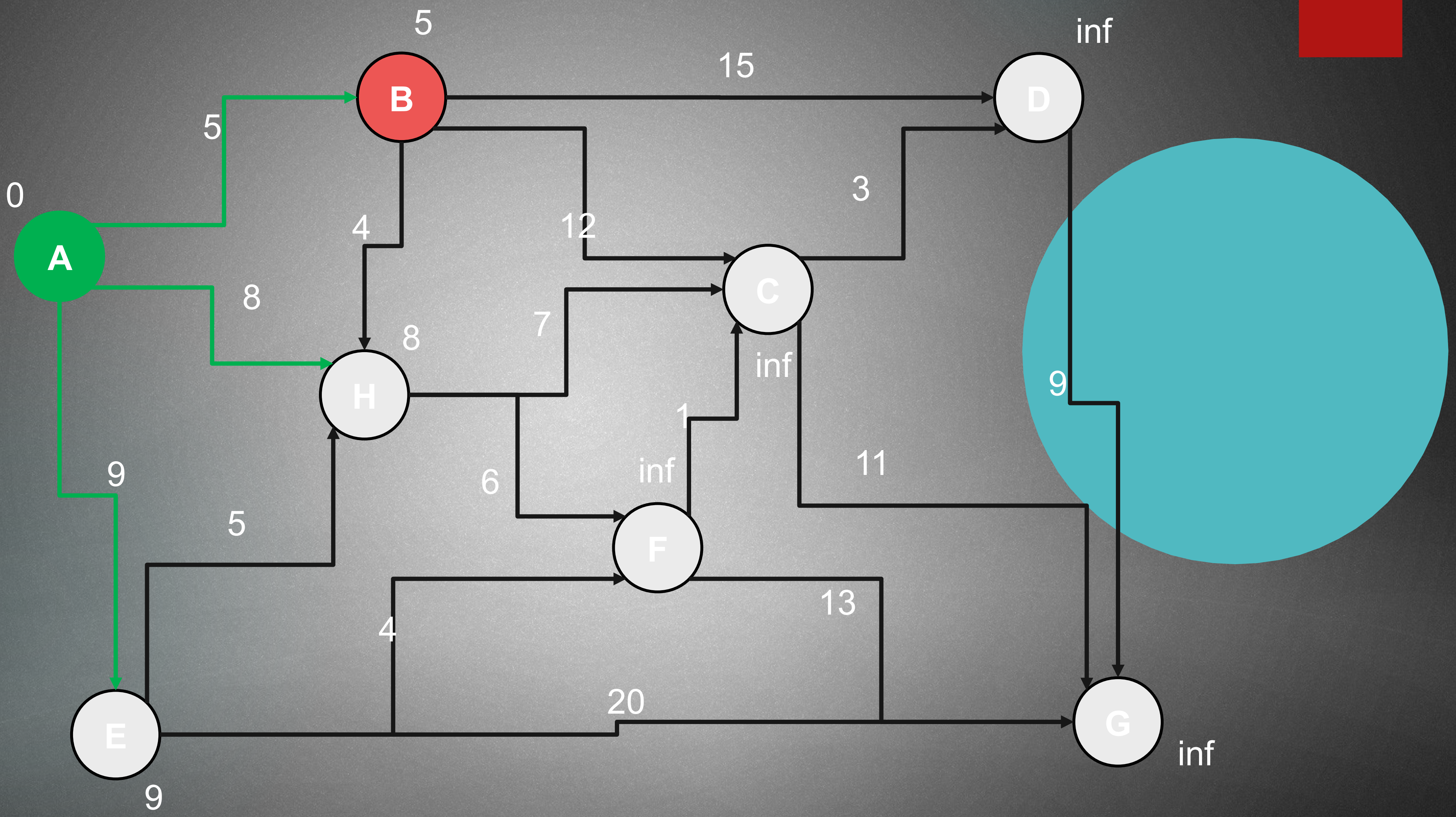
Heap content: B – 5 ; H – 8 ; E - 9



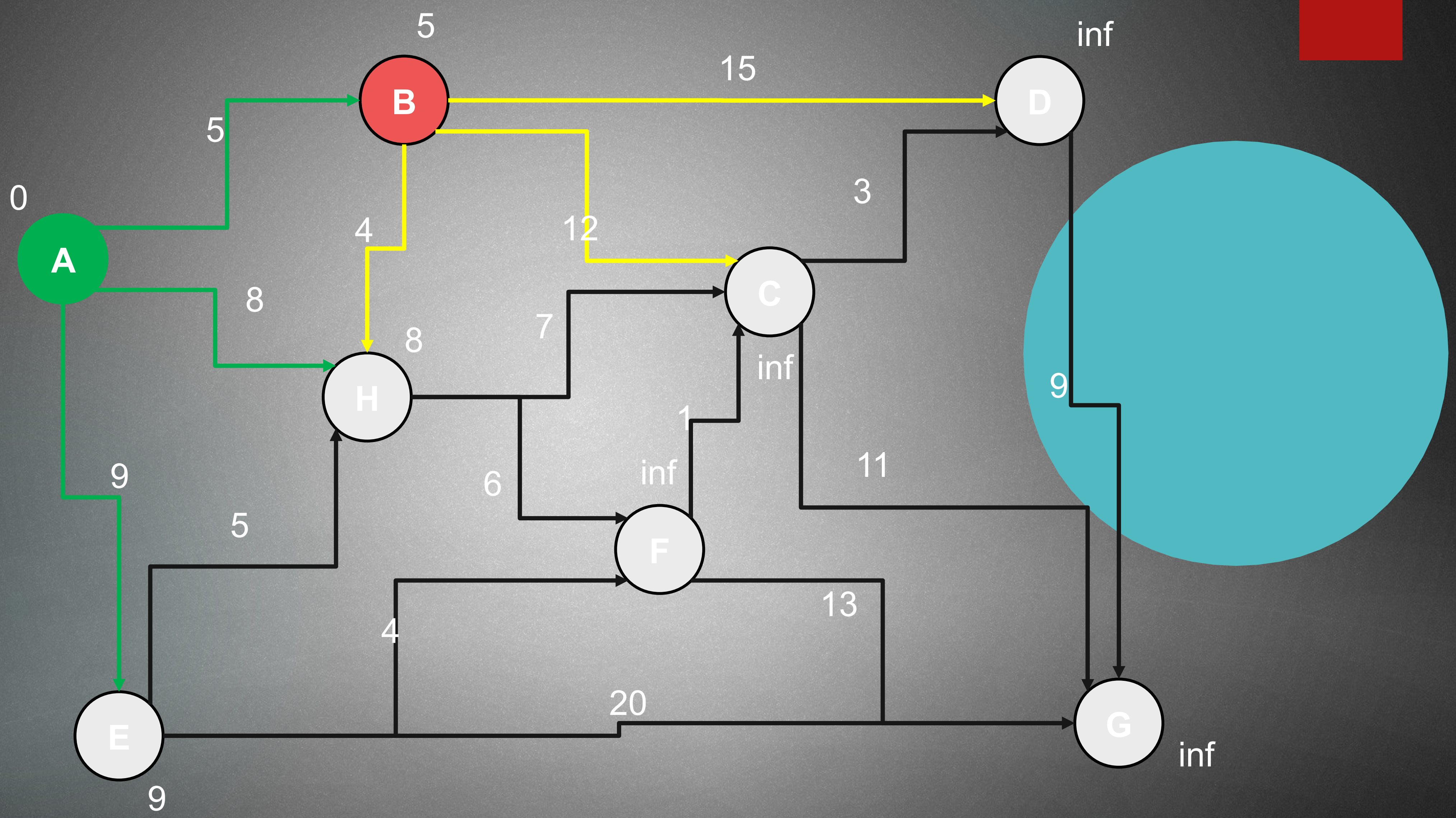
Heap content: **B – 5** ; H – 8 ; E - 9



Heap content: H – 8 ; E - 9

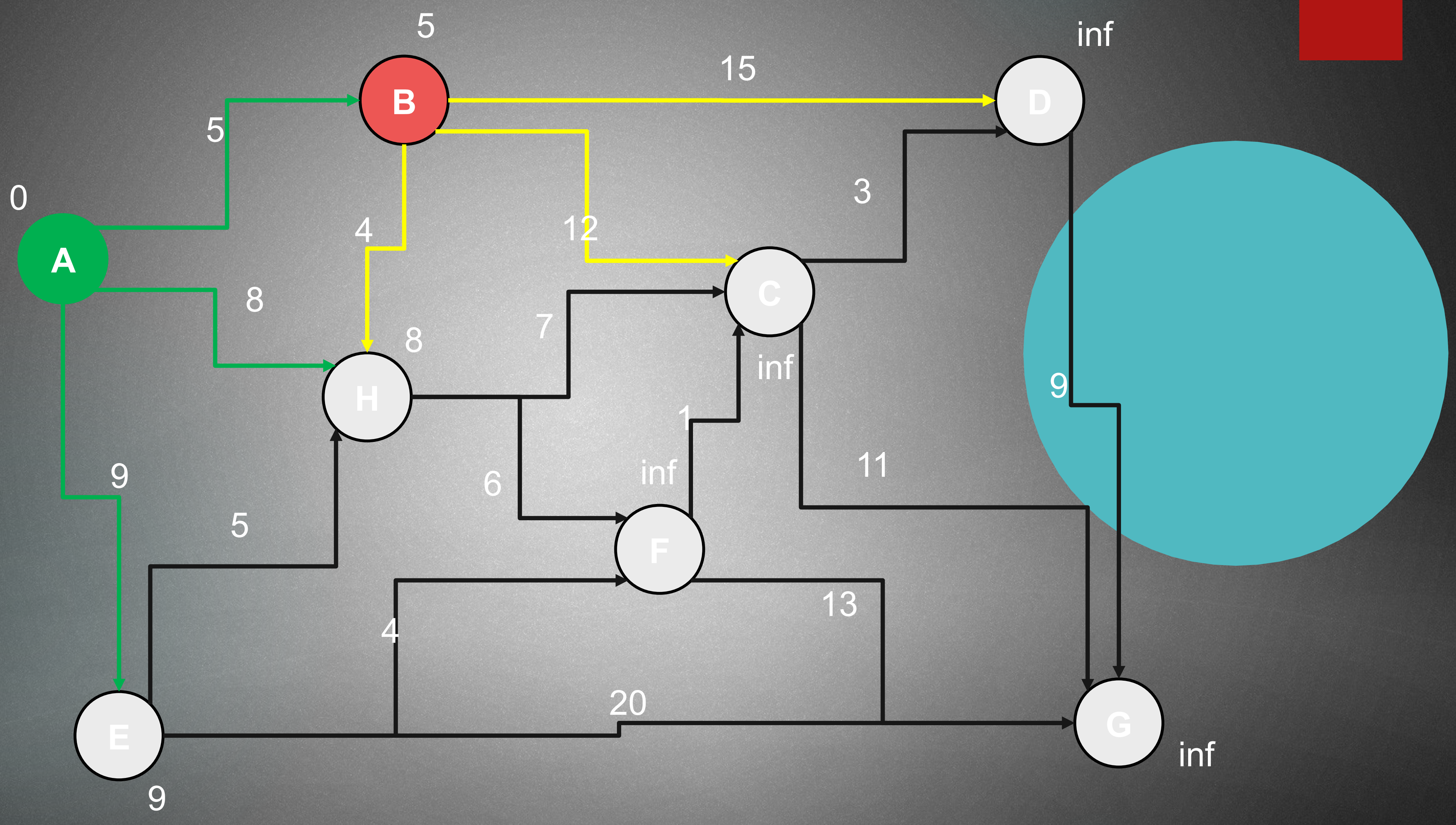


Heap content: H – 8 ; E - 9



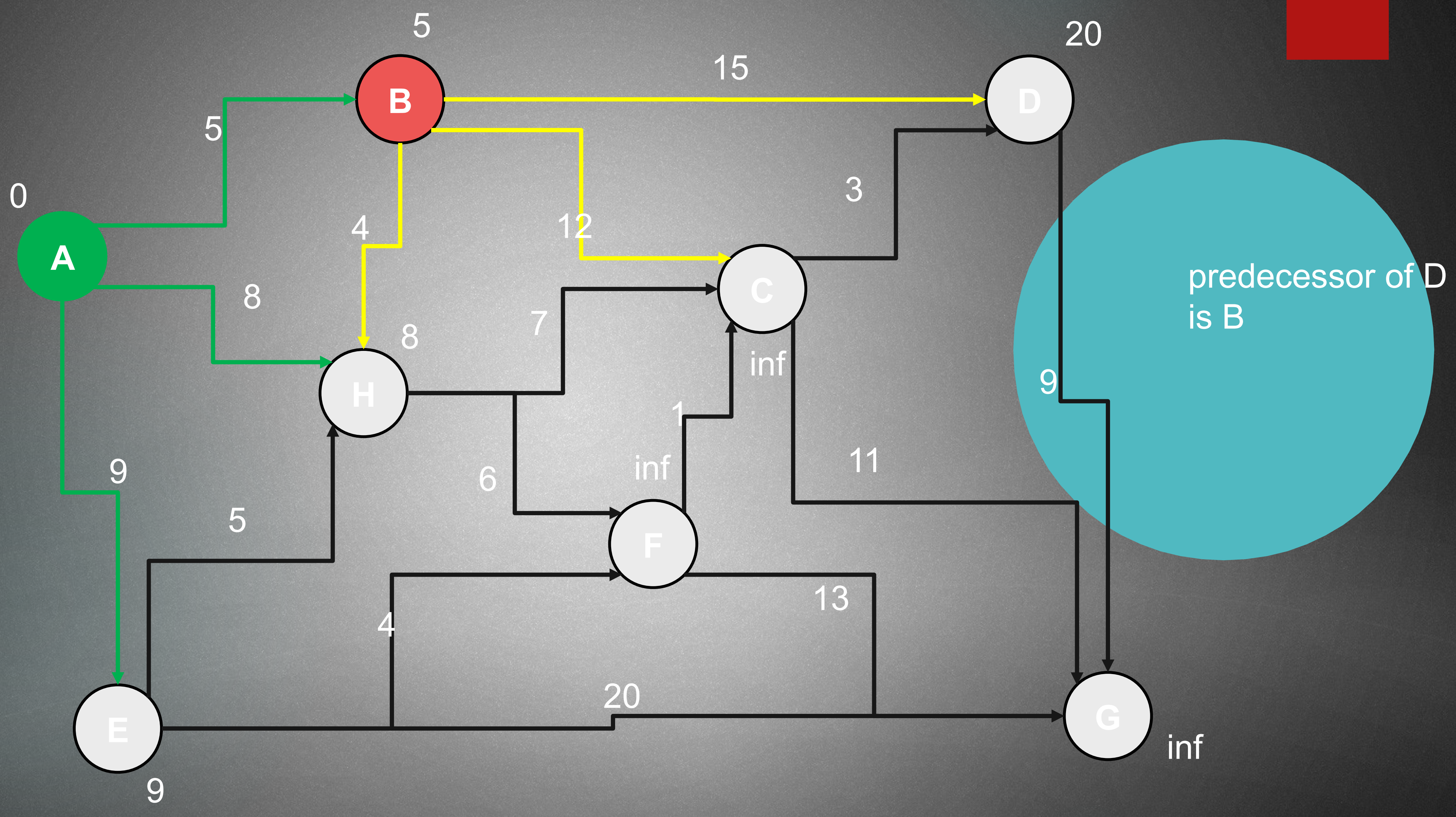
Node D: decide what is smaller 5+15 or inf ... 20 is smaller so UPDATE

Heap content: H – 8 ; E - 9



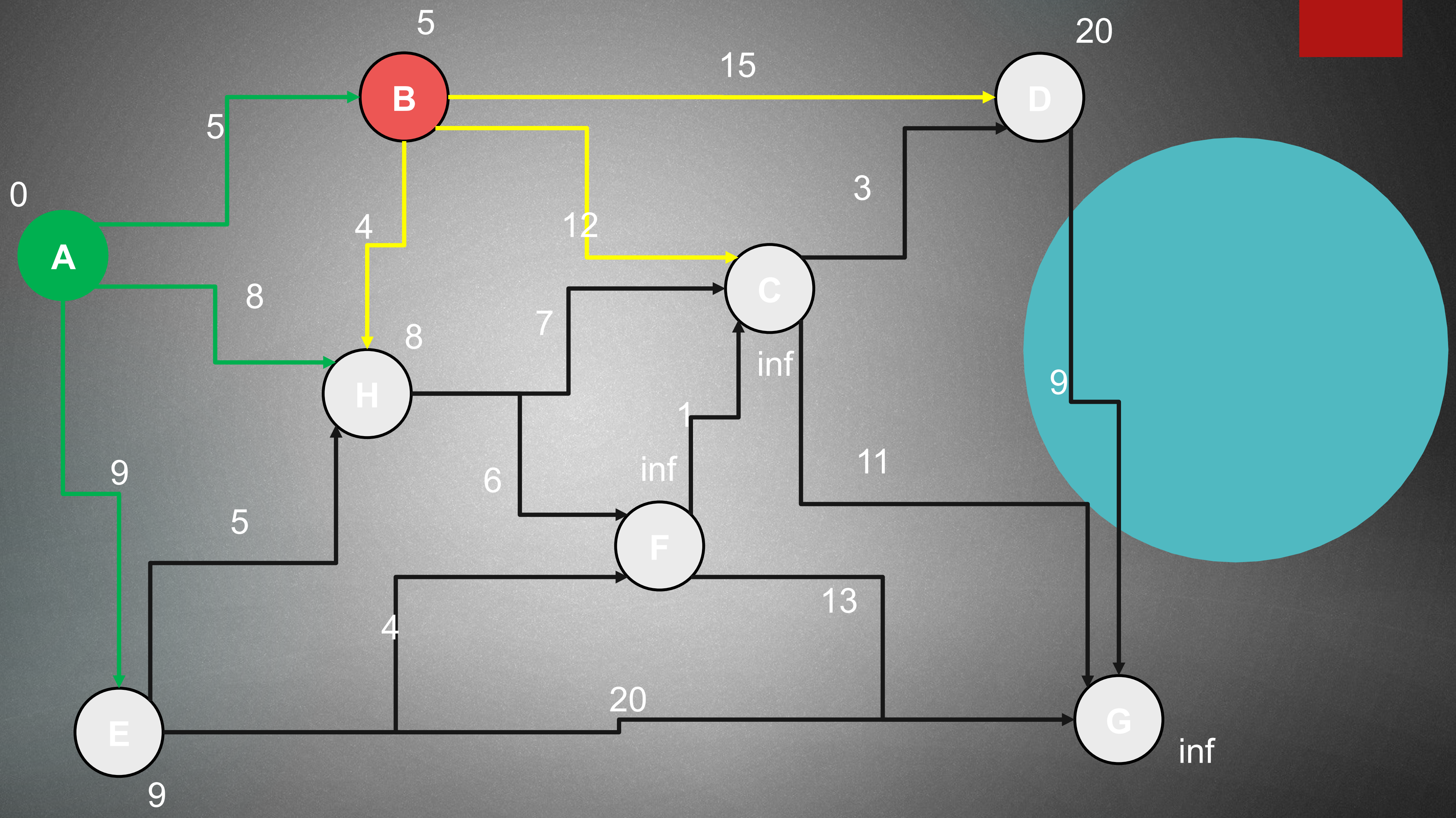
Node D: decide what is smaller 5+15 or inf ... 20 is smaller so UPDATE

Heap content: H – 8 ; E - 9



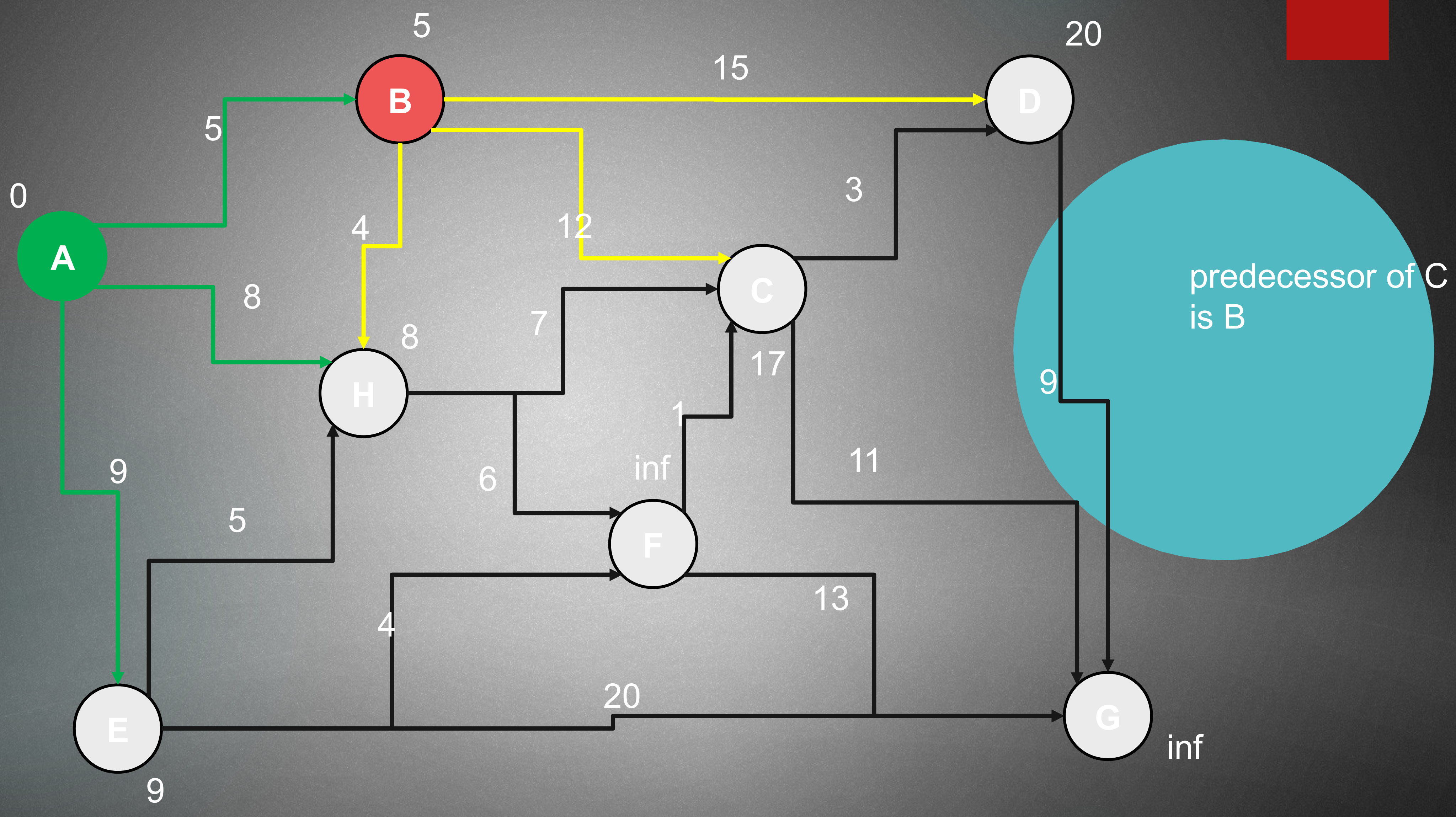
Node C: decide what is smaller 5+12 or inf ... 17 is smaller so UPDATE

Heap content: H – 8 ; E - 9



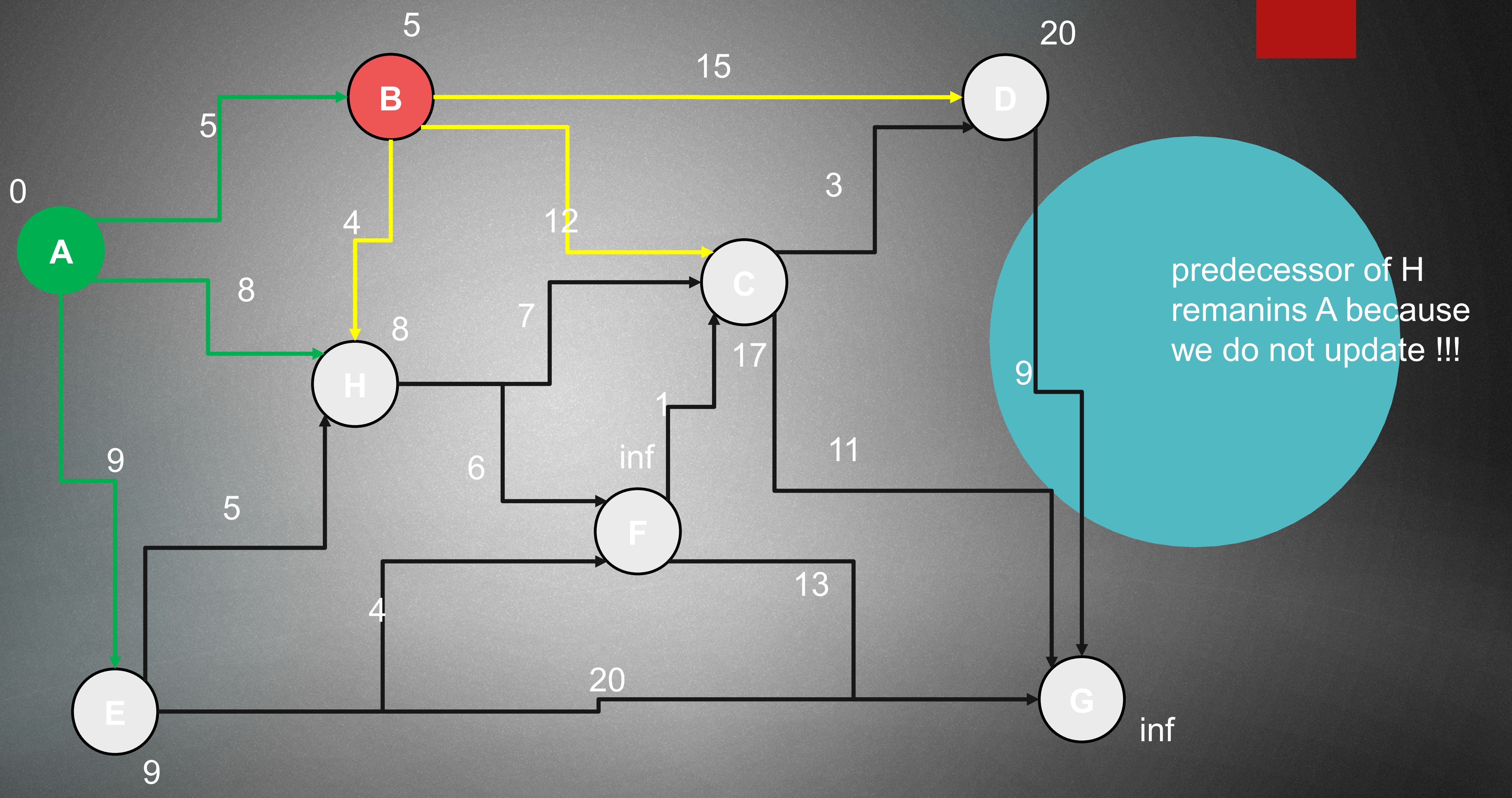
Node C: decide what is smaller 5+12 or inf ... 17 is smaller so UPDATE

Heap content: H – 8 ; E - 9



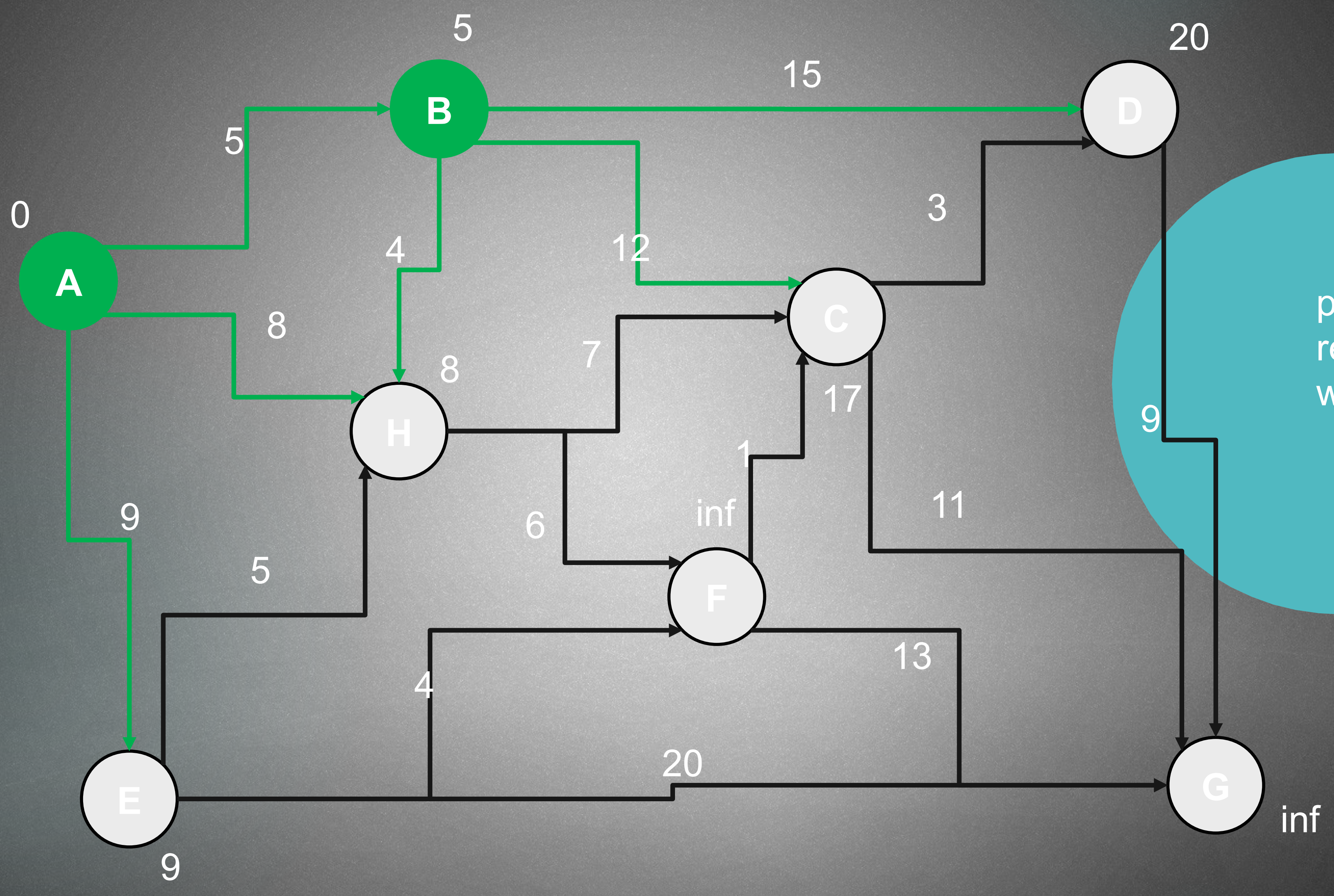
Node H: decide what is smaller 5+4 or 8 ... 8 is smaller so DO NOT UPDATE

Heap content: H – 8 ; E - 9



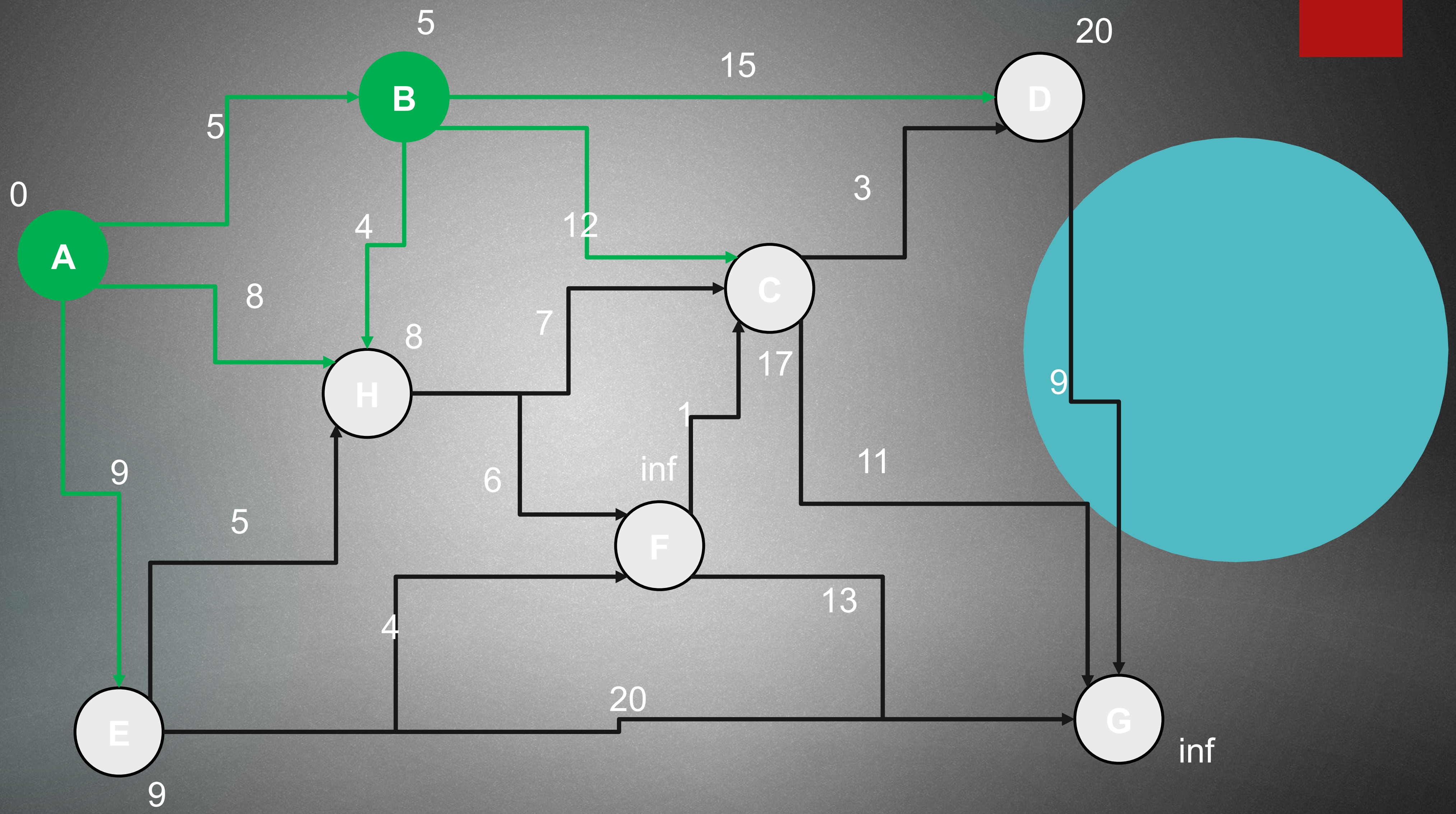
Node H: decide what is smaller 5+4 or 8 ... 8 is smaller so DO NOT UPDATE

Heap content: H – 8 ; E - 9

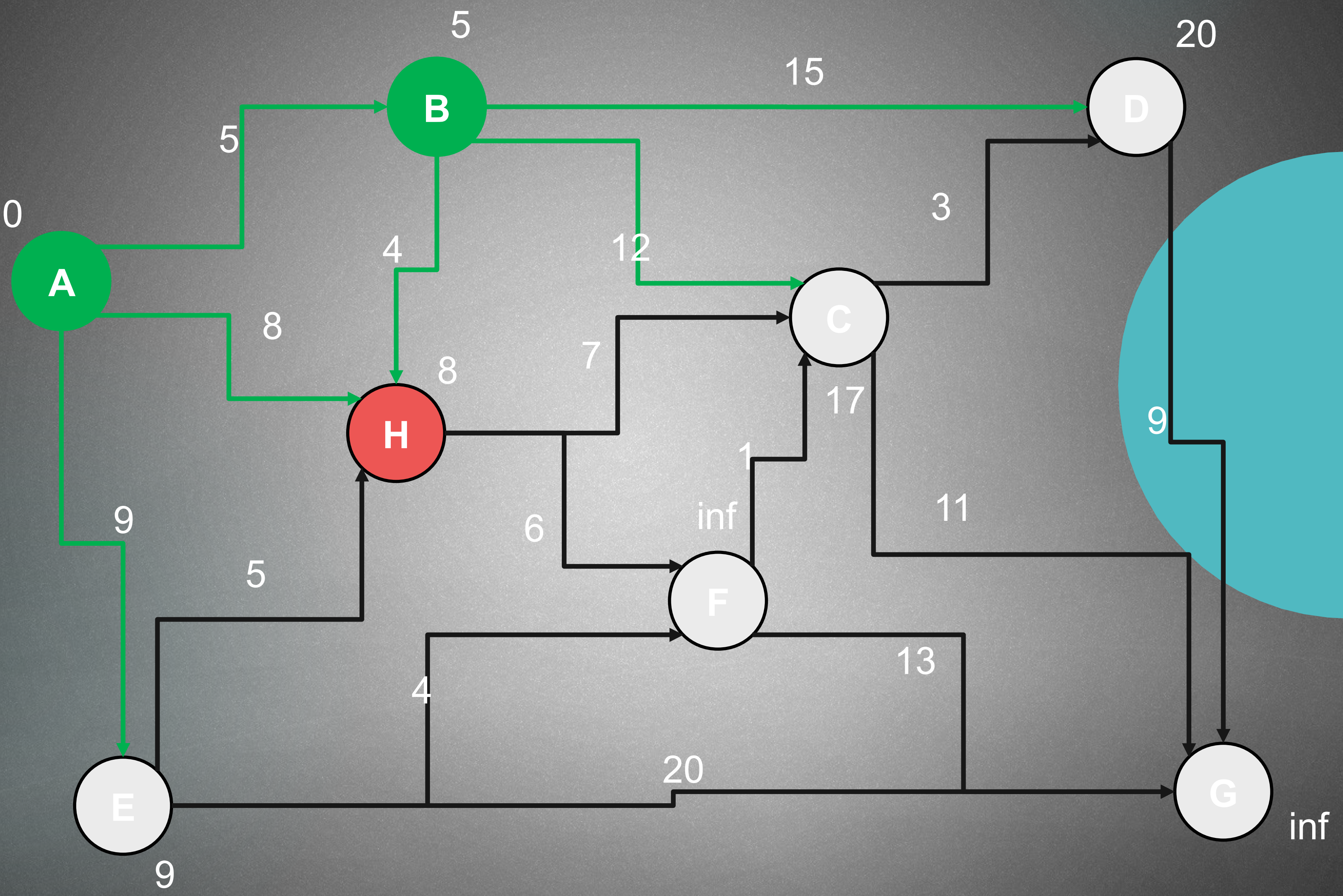


predecessor of H remanins A because we do not update !!!

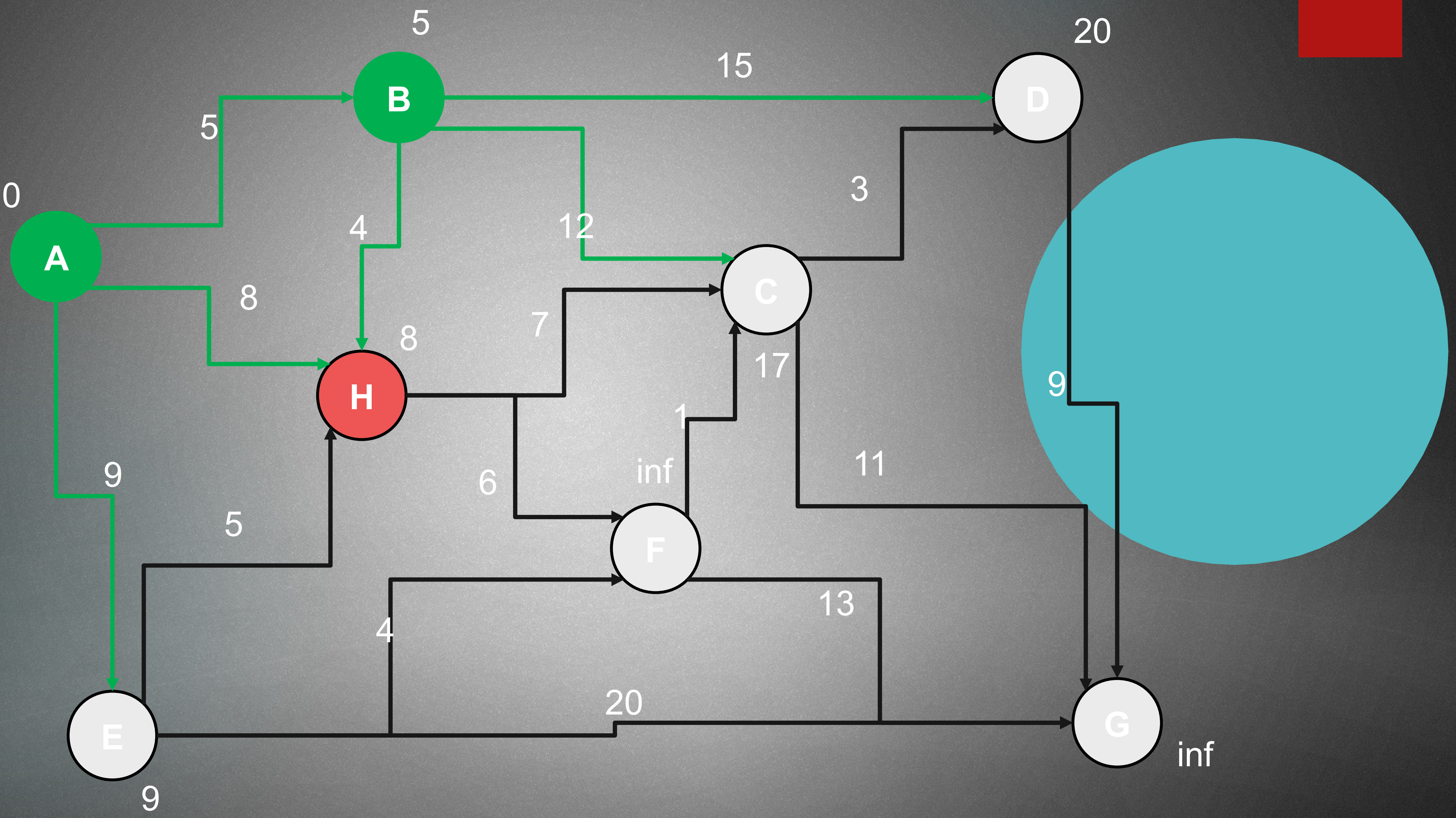
Heap content: H – 8 ; E – 9 ; C – 17 ; D – 20



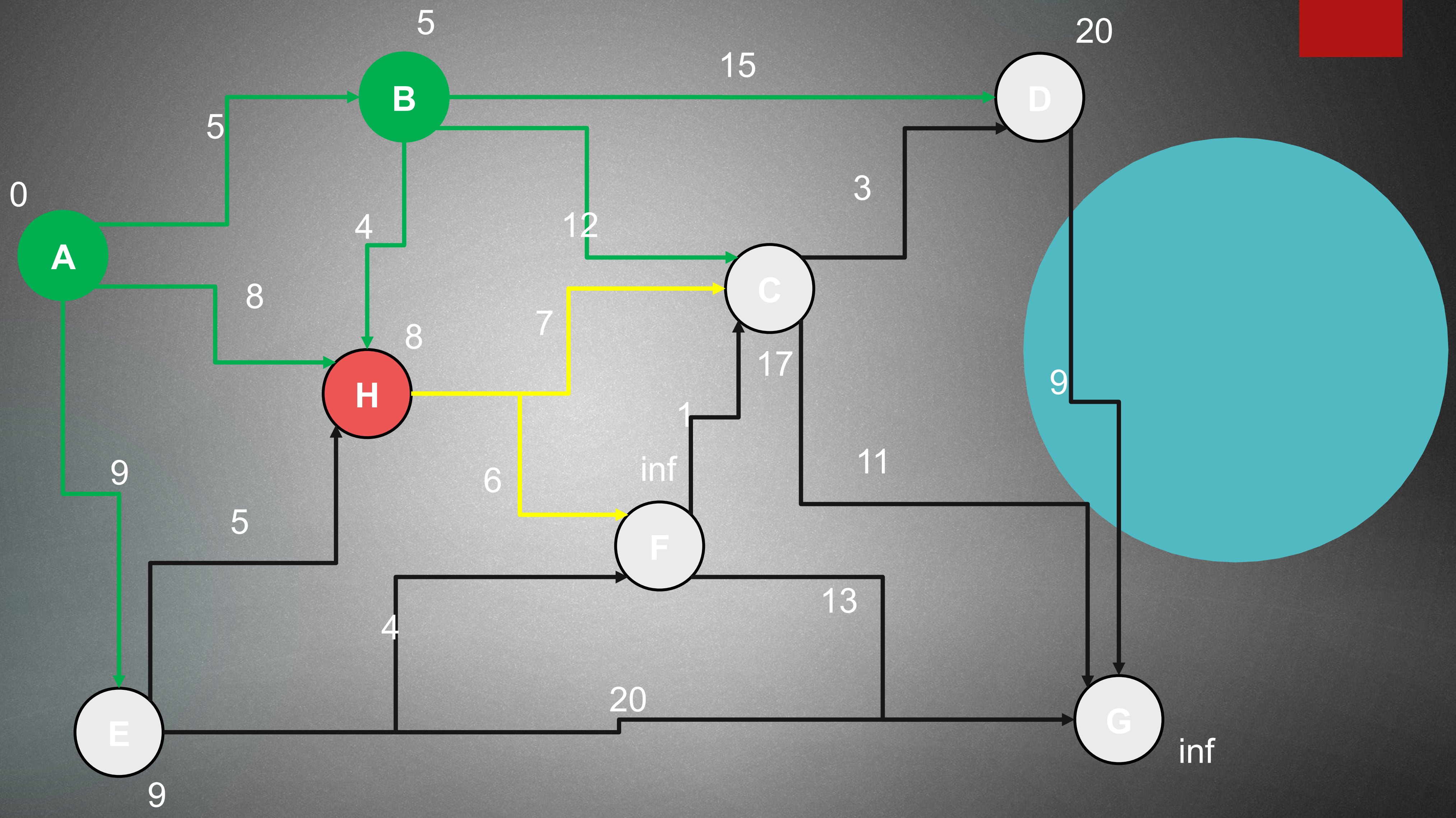
Heap content: **H – 8** ; E – 9 ; C – 17 ; D – 20



Heap content: E – 9 ; C – 17 ; D – 20

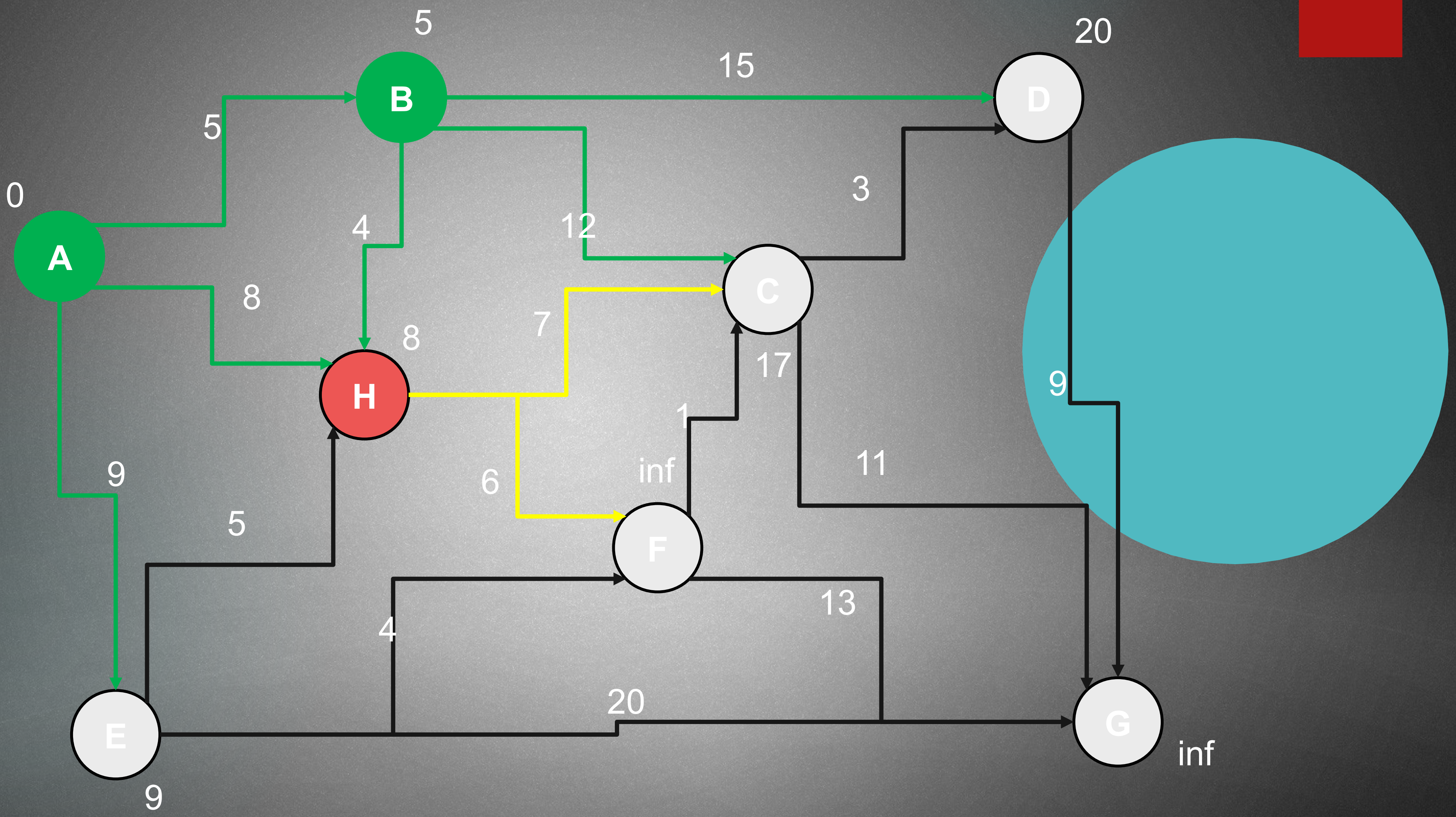


Heap content: E – 9 ; C – 17 ; D – 20



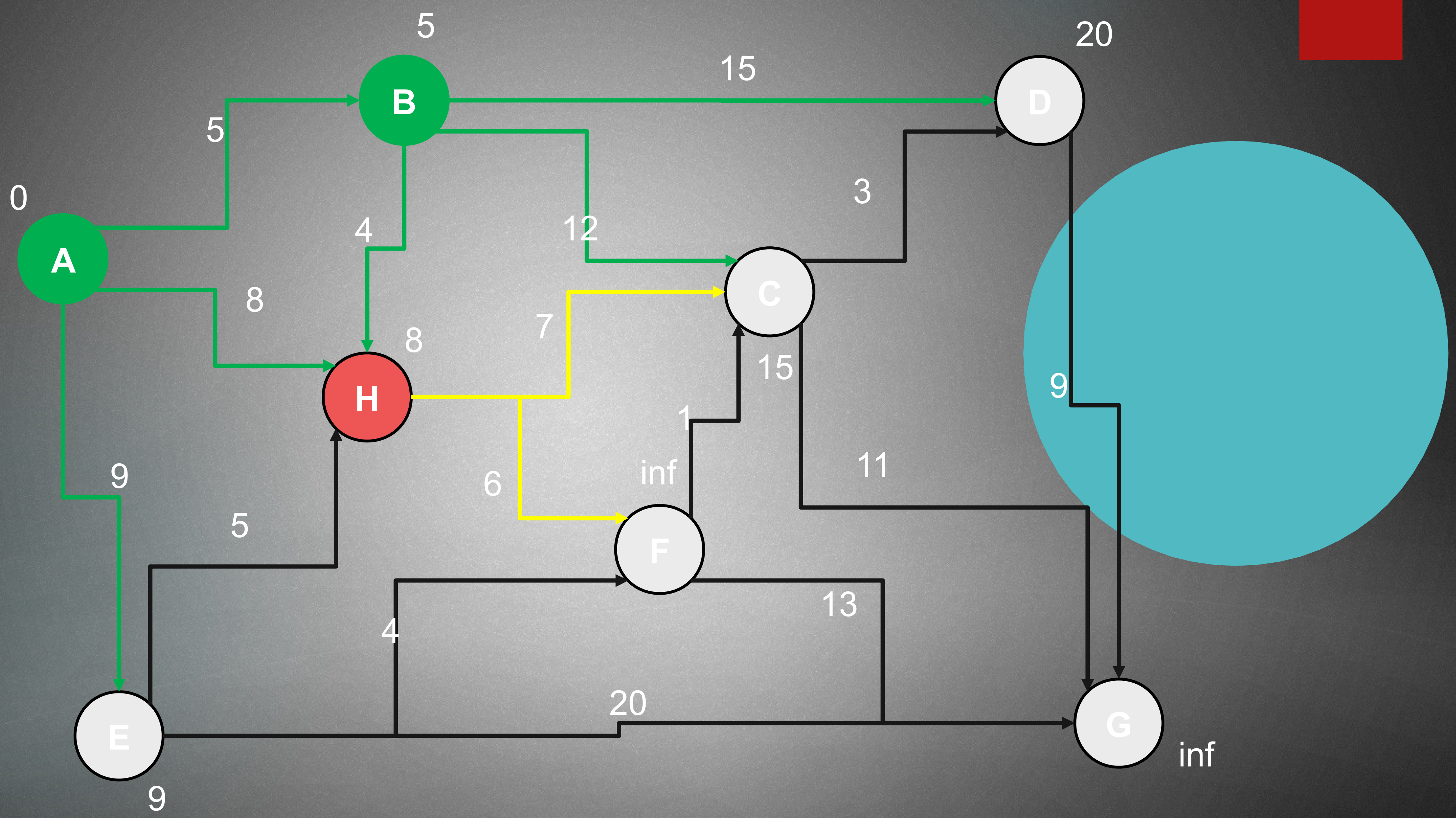
Node C: decide what is smaller 8+7 or 17 ... 15 is smaller so UPDATE

Heap content: E – 9 ; C – 17 ; D – 20



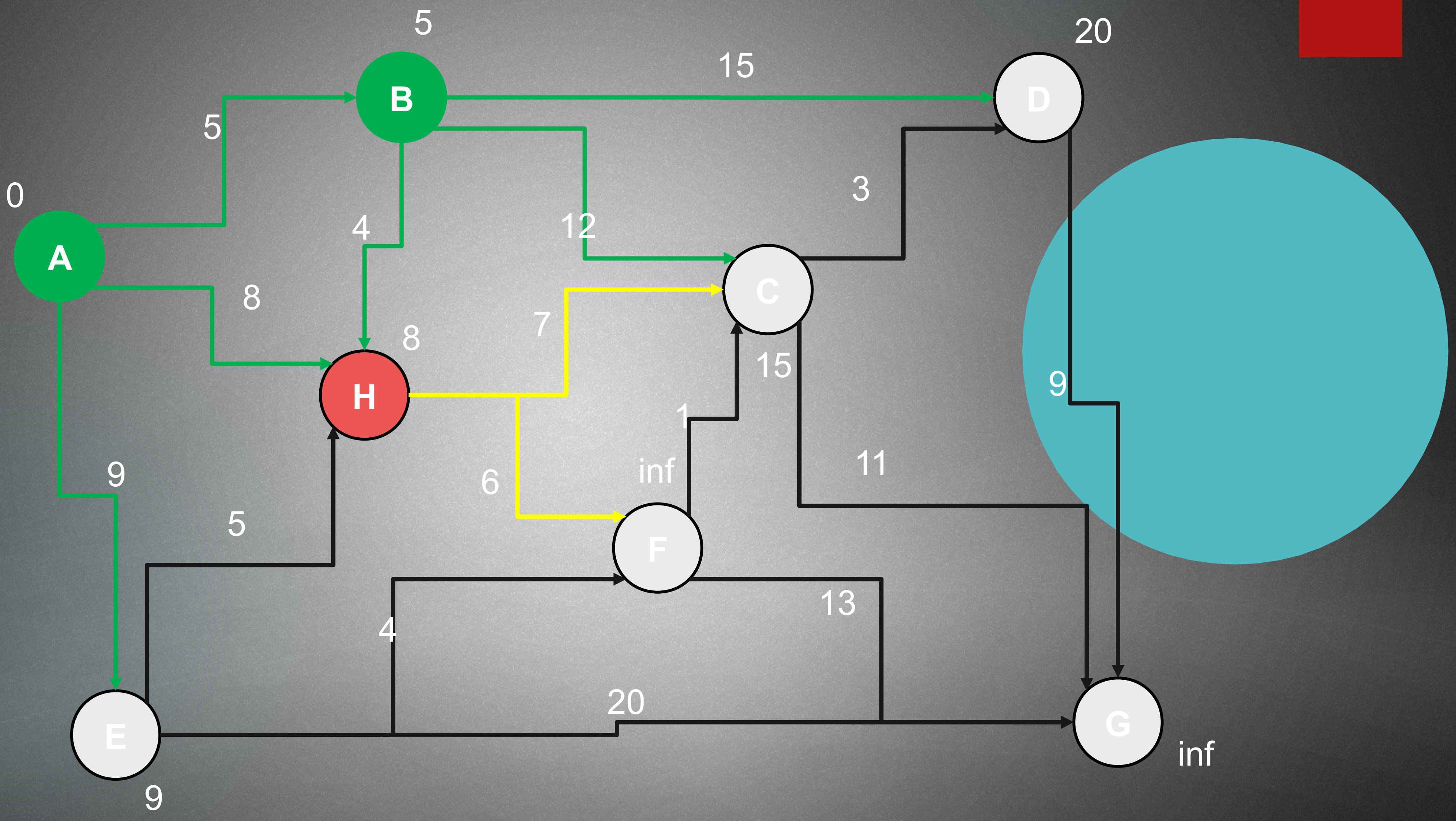
Node C: decide what is smaller 8+7 or 17 ... 15 is smaller so UPDATE // we have to update the heap

Heap content: E – 9 ; C – 15 ; D – 20



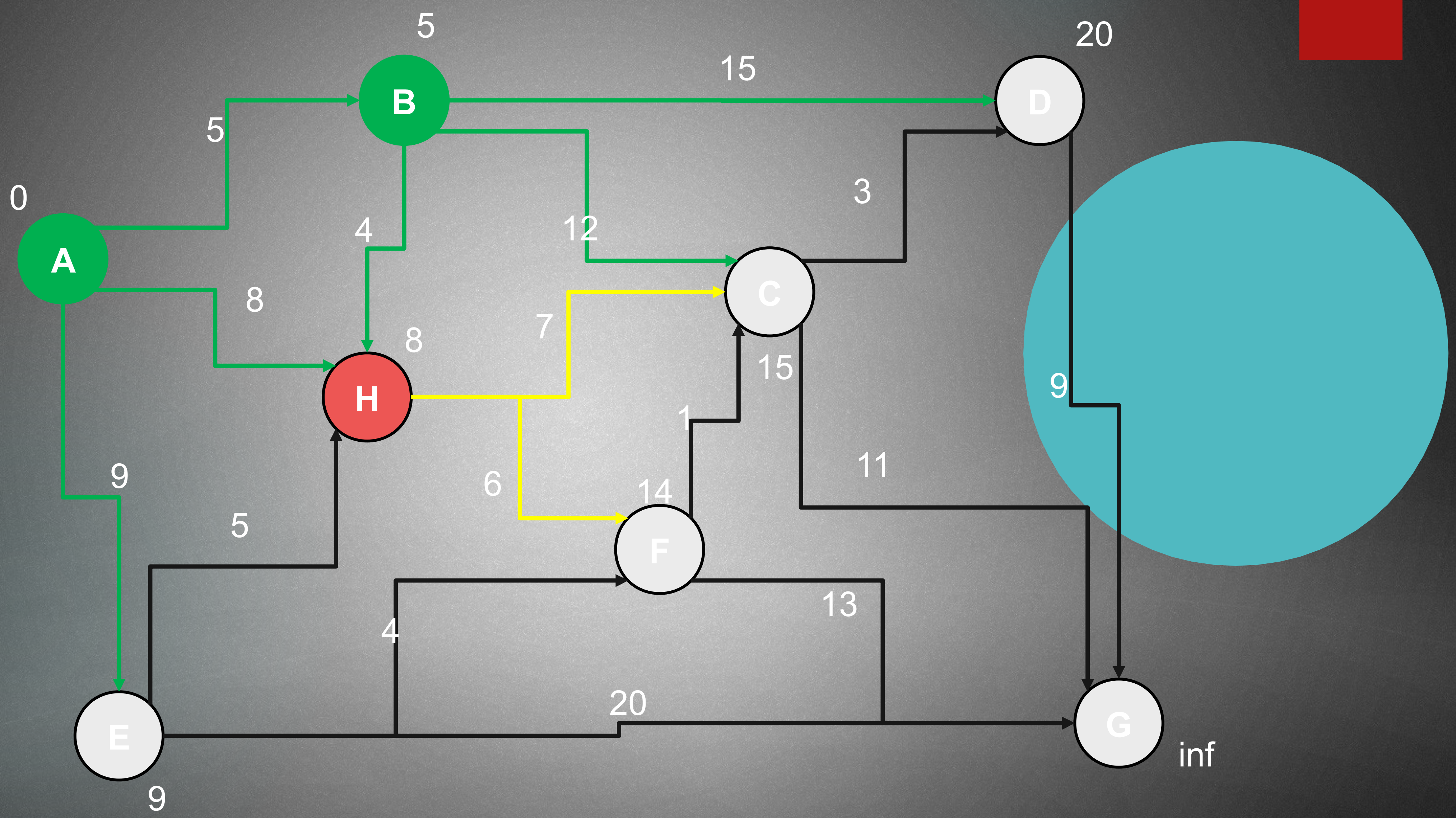
Node F: decide what is smaller 8+6 or inf ... 14 is smaller so UPDATE

Heap content: E – 9 ; C – 15 ; D – 20

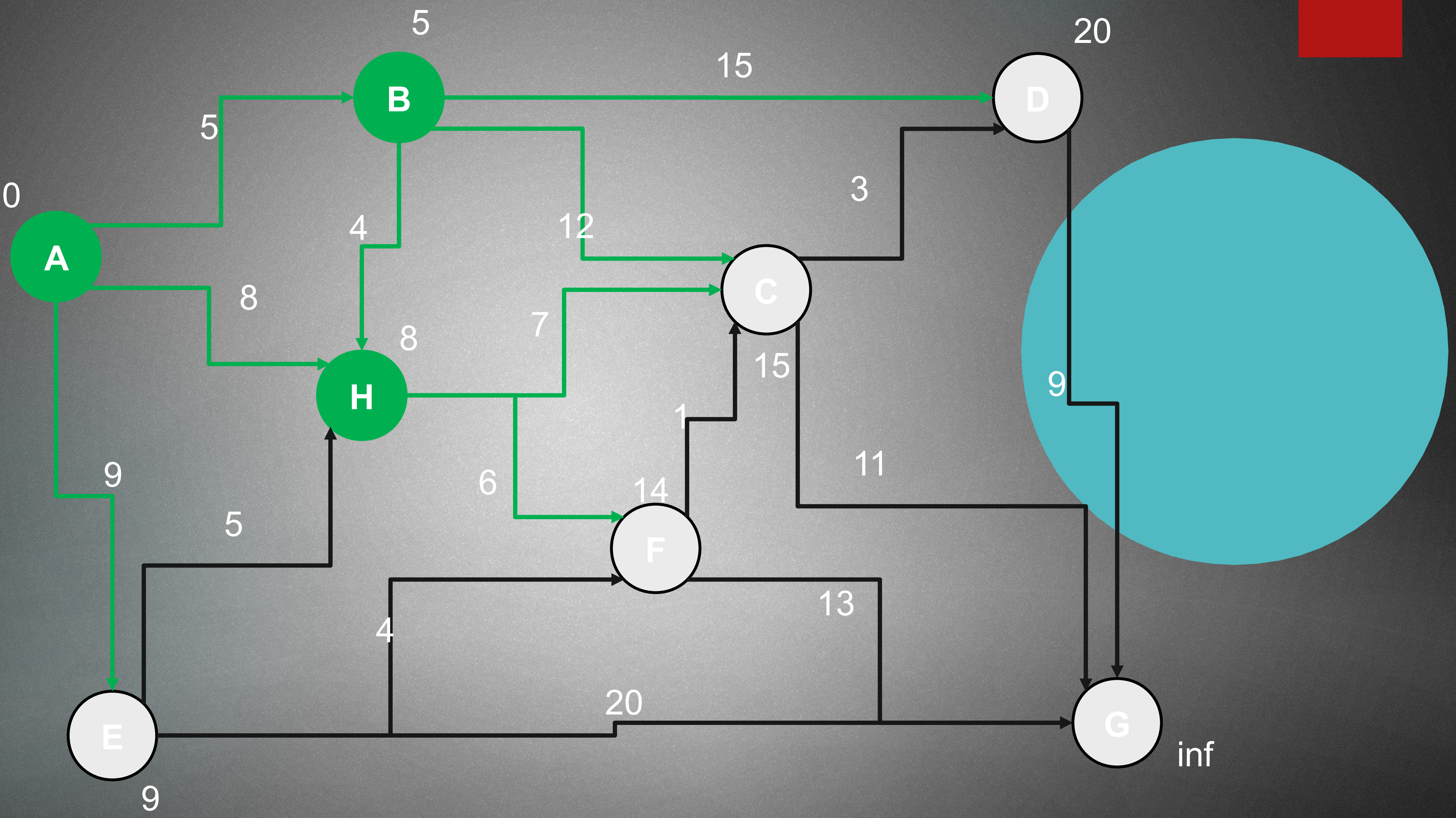


Node F: decide what is smaller 8+6 or inf ... 14 is smaller so UPDATE

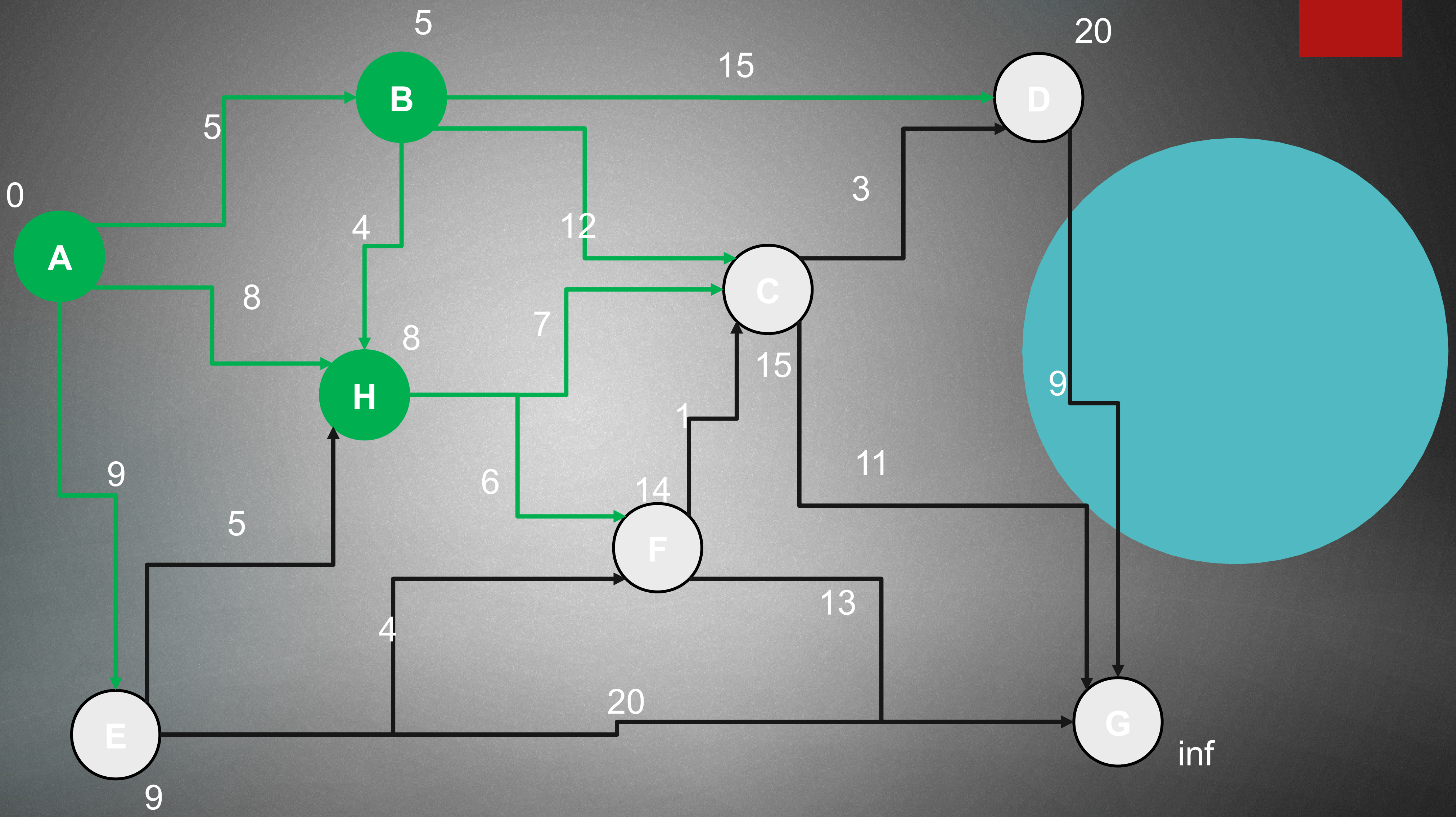
Heap content: E – 9 ; C – 15 ; D – 20



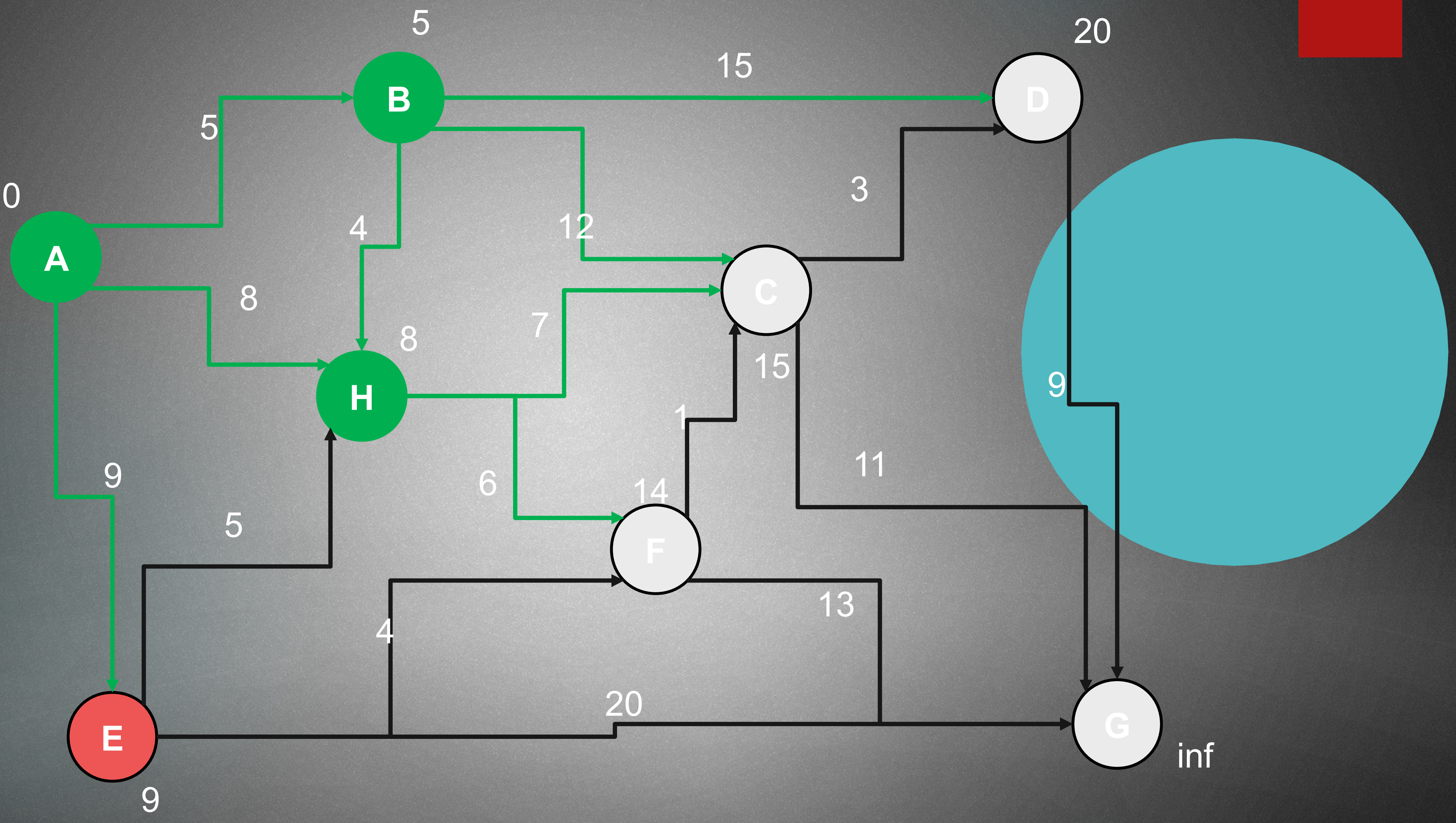
Heap content: E – 9 ; C – 15 ; D – 20 ; F – 14



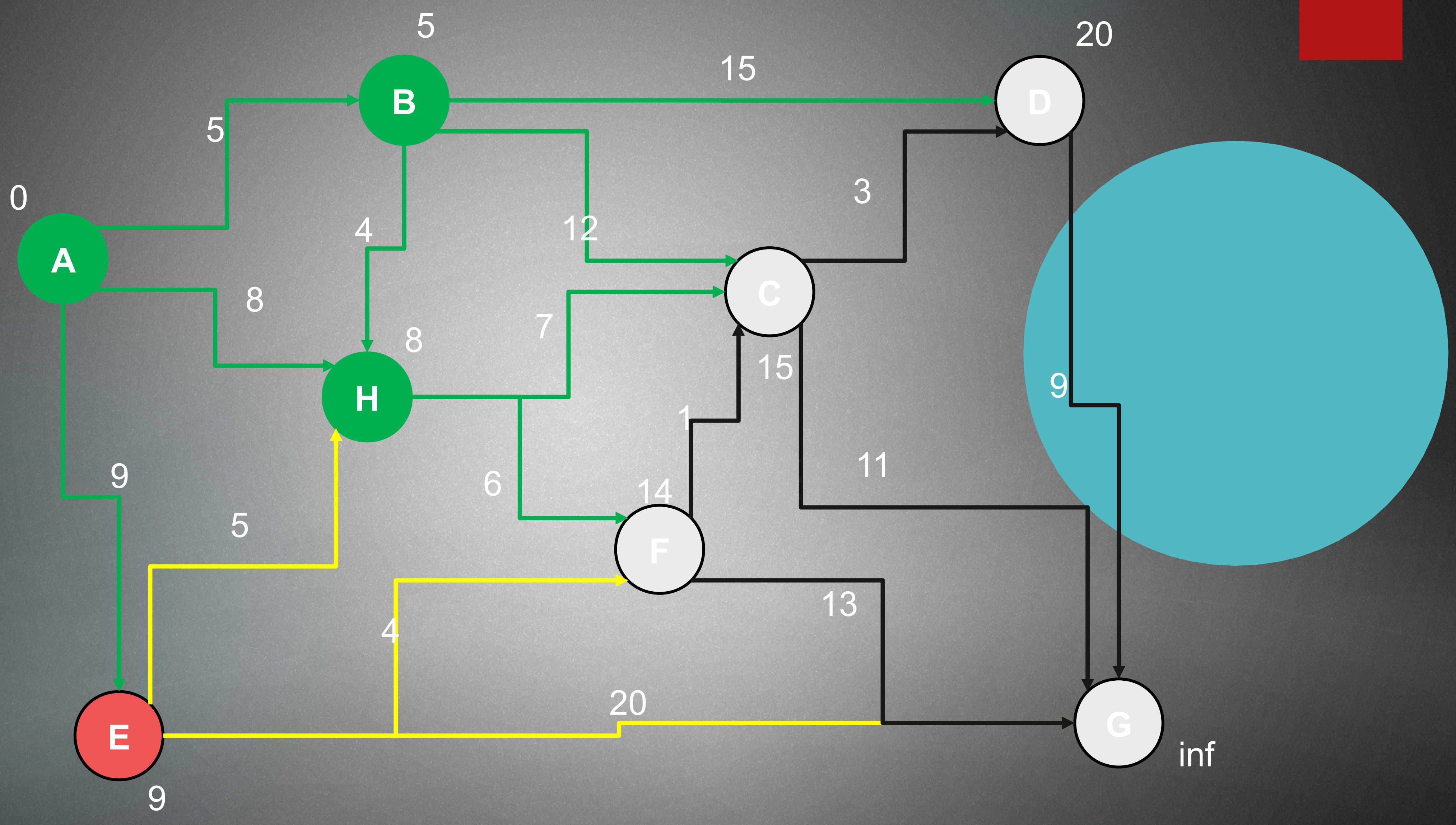
Heap content: **E – 9** ; C – 15 ; D – 20 ; F – 14



Heap content: C – 15 ; D – 20 ; F – 14

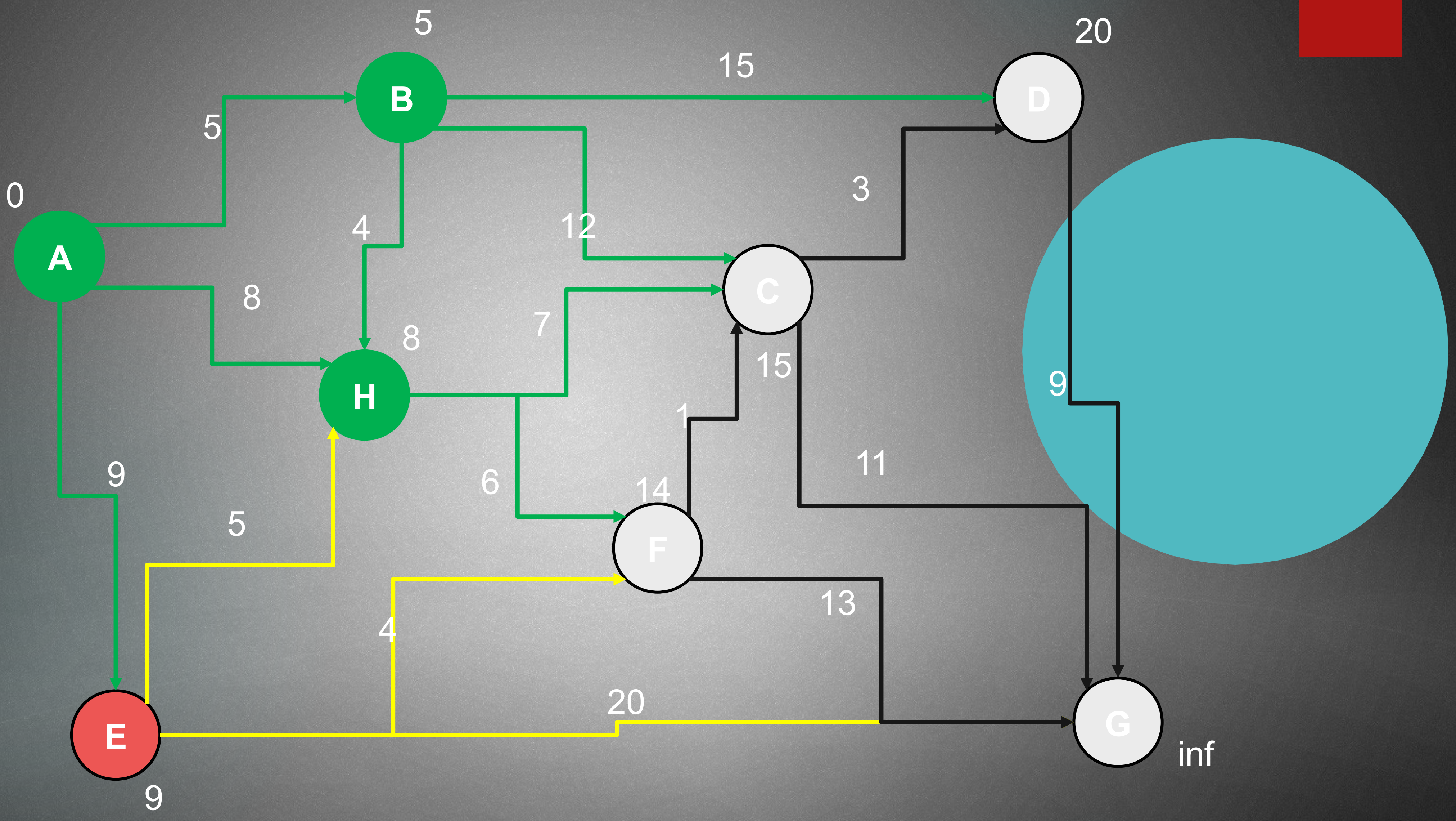


Heap content: C – 15 ; D – 20 ; F – 14



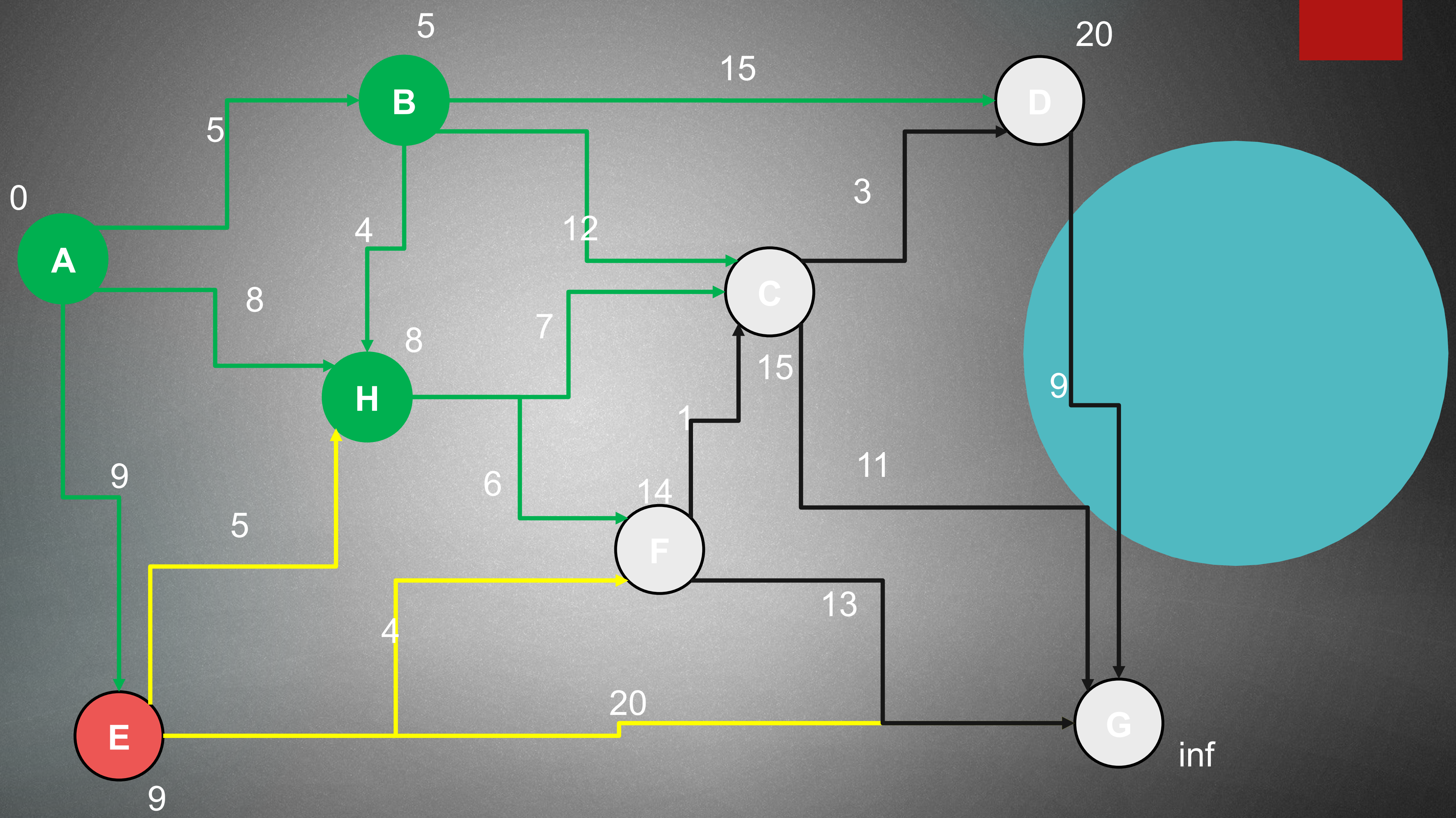
Node H: decide what is smaller 9+5 or 8 ... 8 is smaller so DO NOT UPDATE

Heap content: C – 15 ; D – 20 ; F – 14



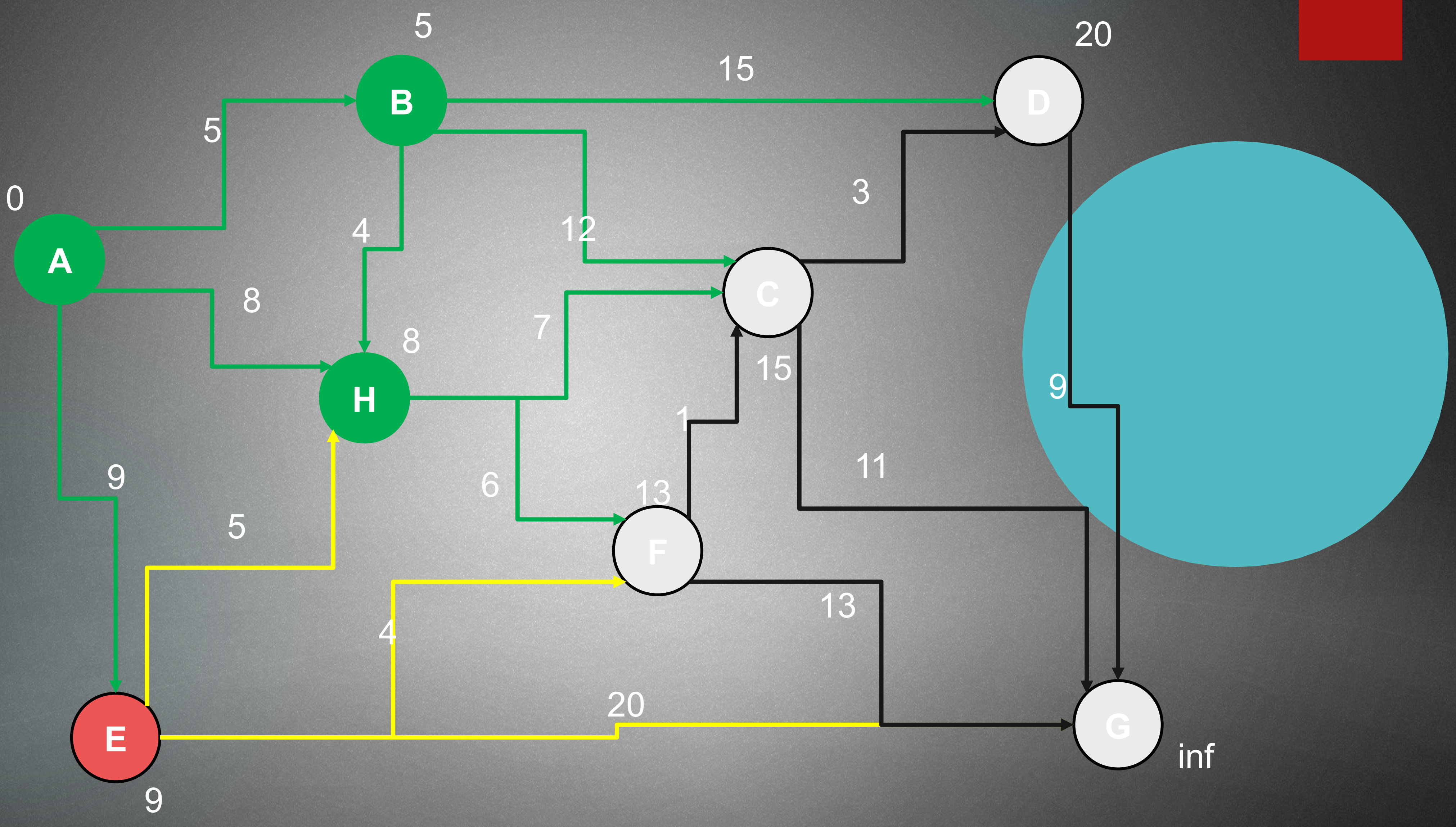
Node F: decide what is smaller 9+4 or 14 ... 13 is smaller so UPDATE

Heap content: C – 15 ; D – 20 ; F – 14



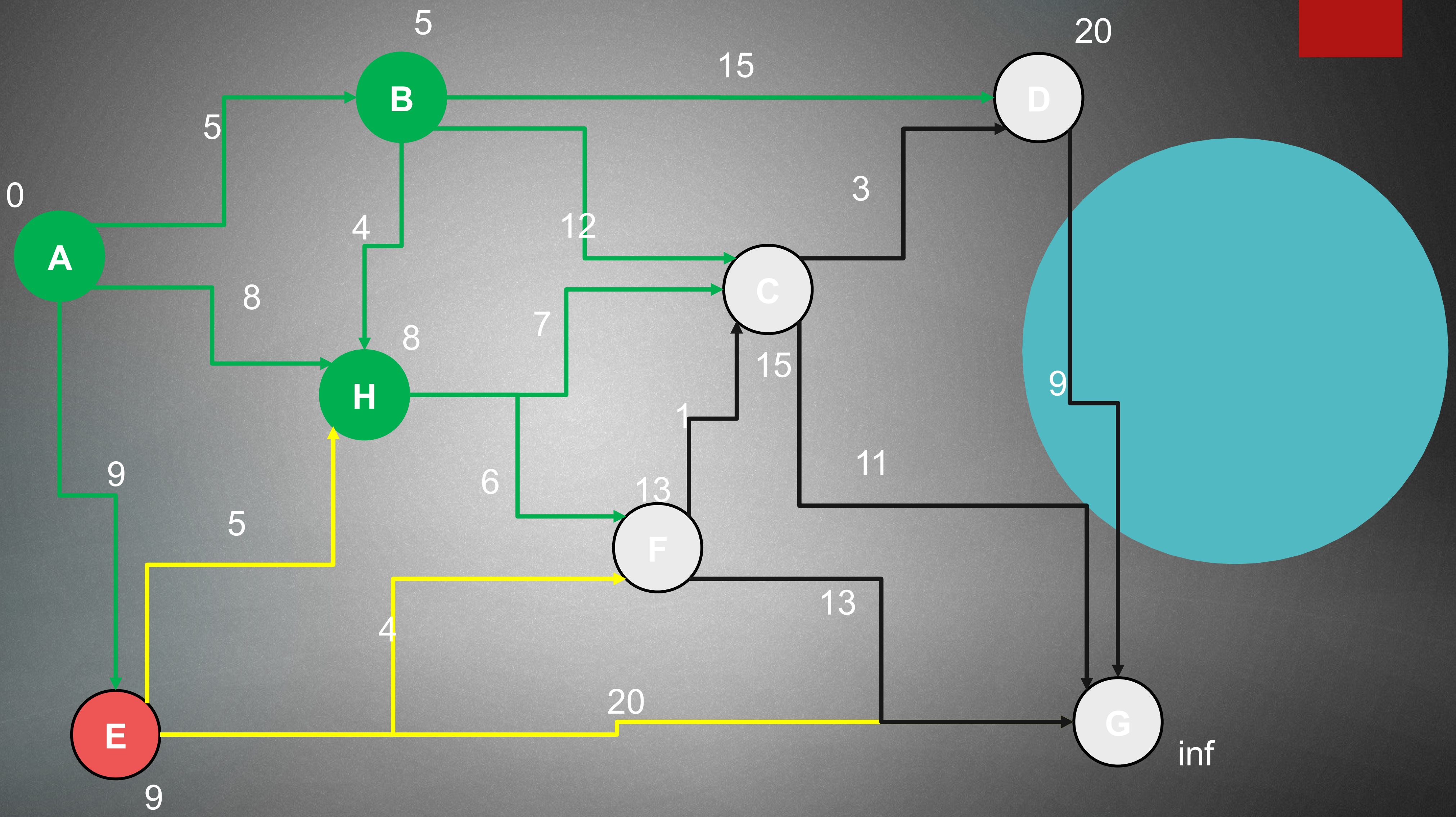
Node F: decide what is smaller 9+4 or 14 ... 13 is smaller so UPDATE // update heap

Heap content: C – 15 ; D – 20 ; F – 13



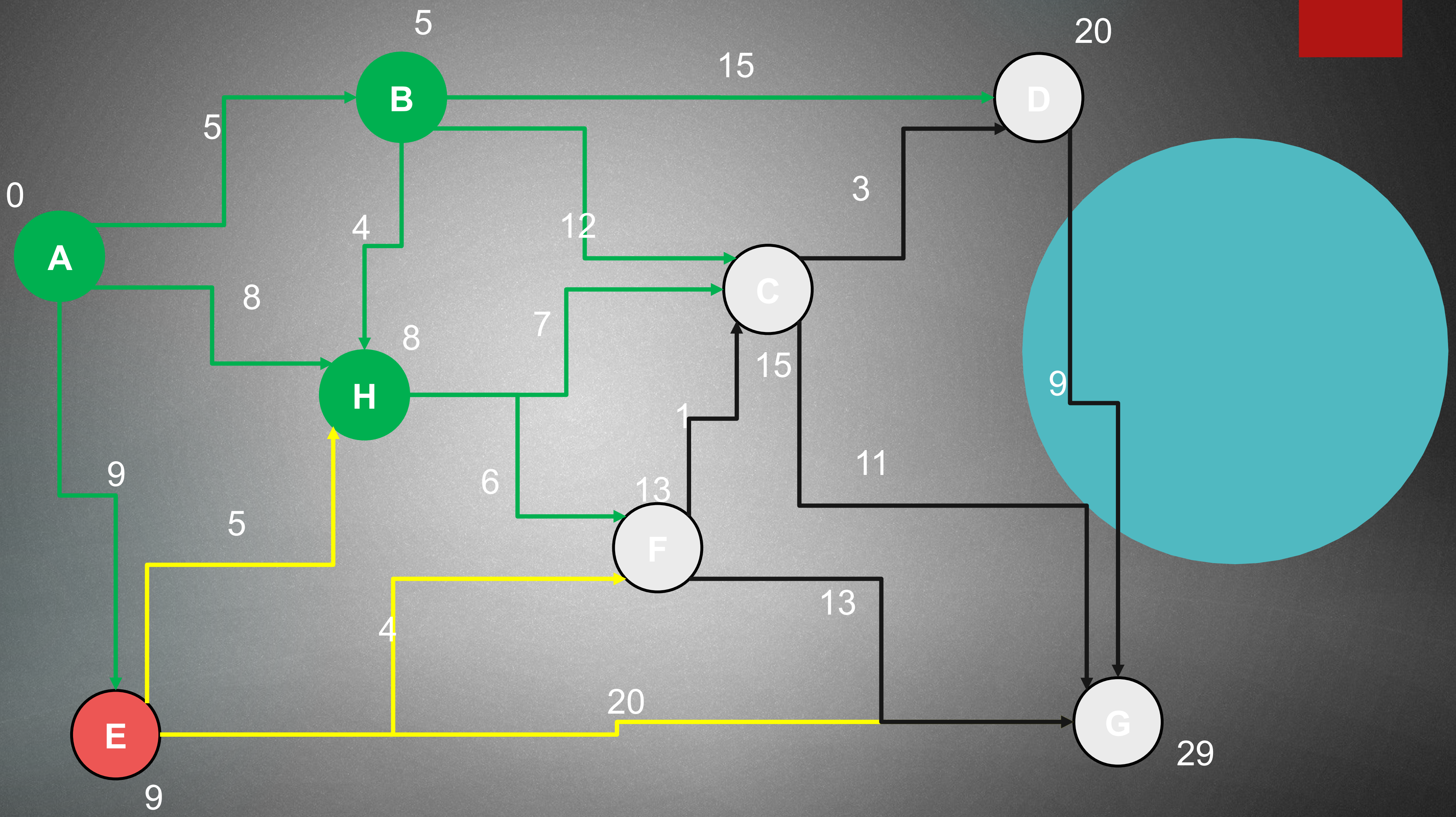
Node G: decide what is smaller 9+20 or inf ... 29 is smaller so UPDATE

Heap content: C – 15 ; D – 20 ; F – 13

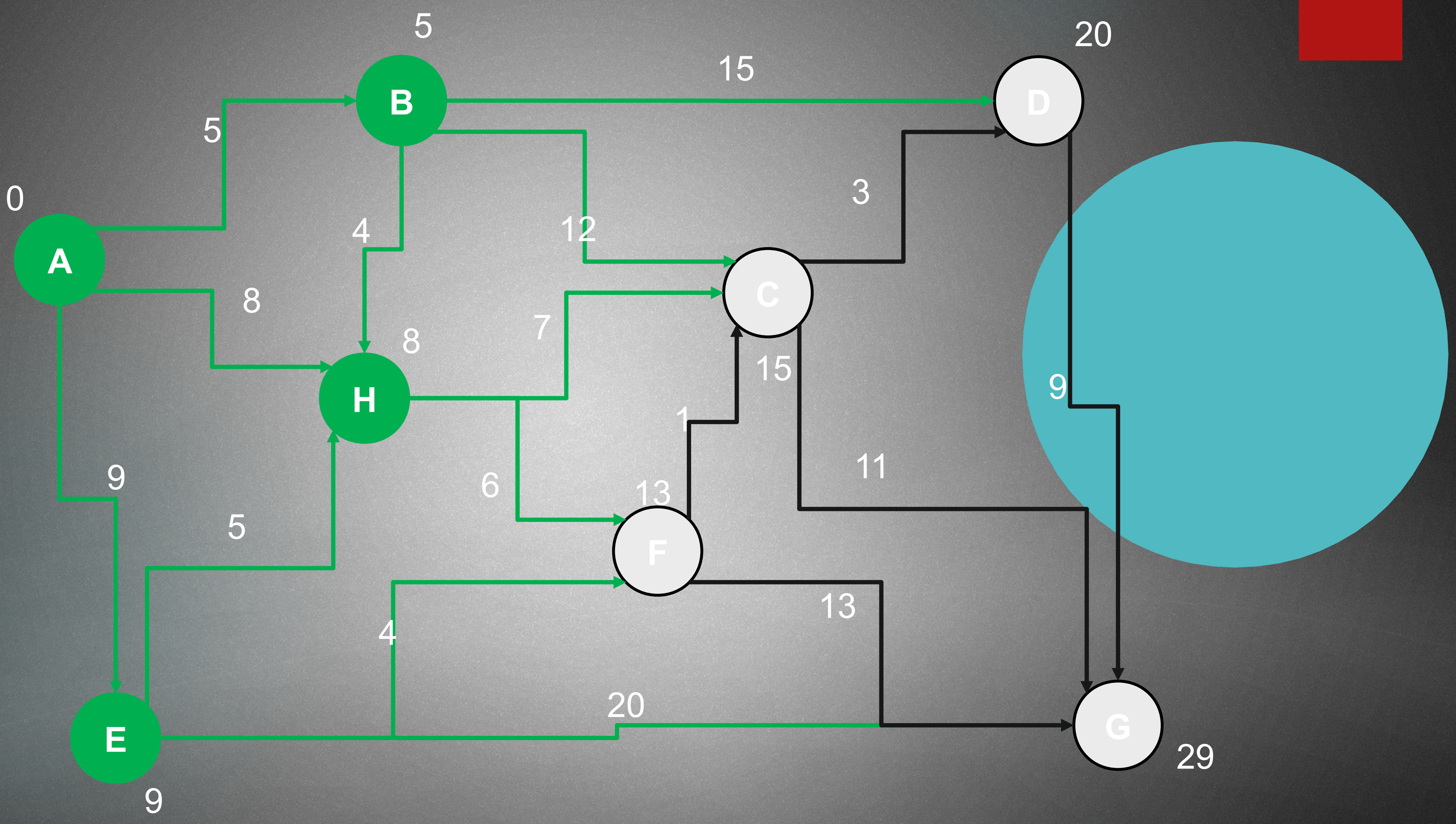


Node G: decide what is smaller 9+20 or inf ... 29 is smaller so UPDATE

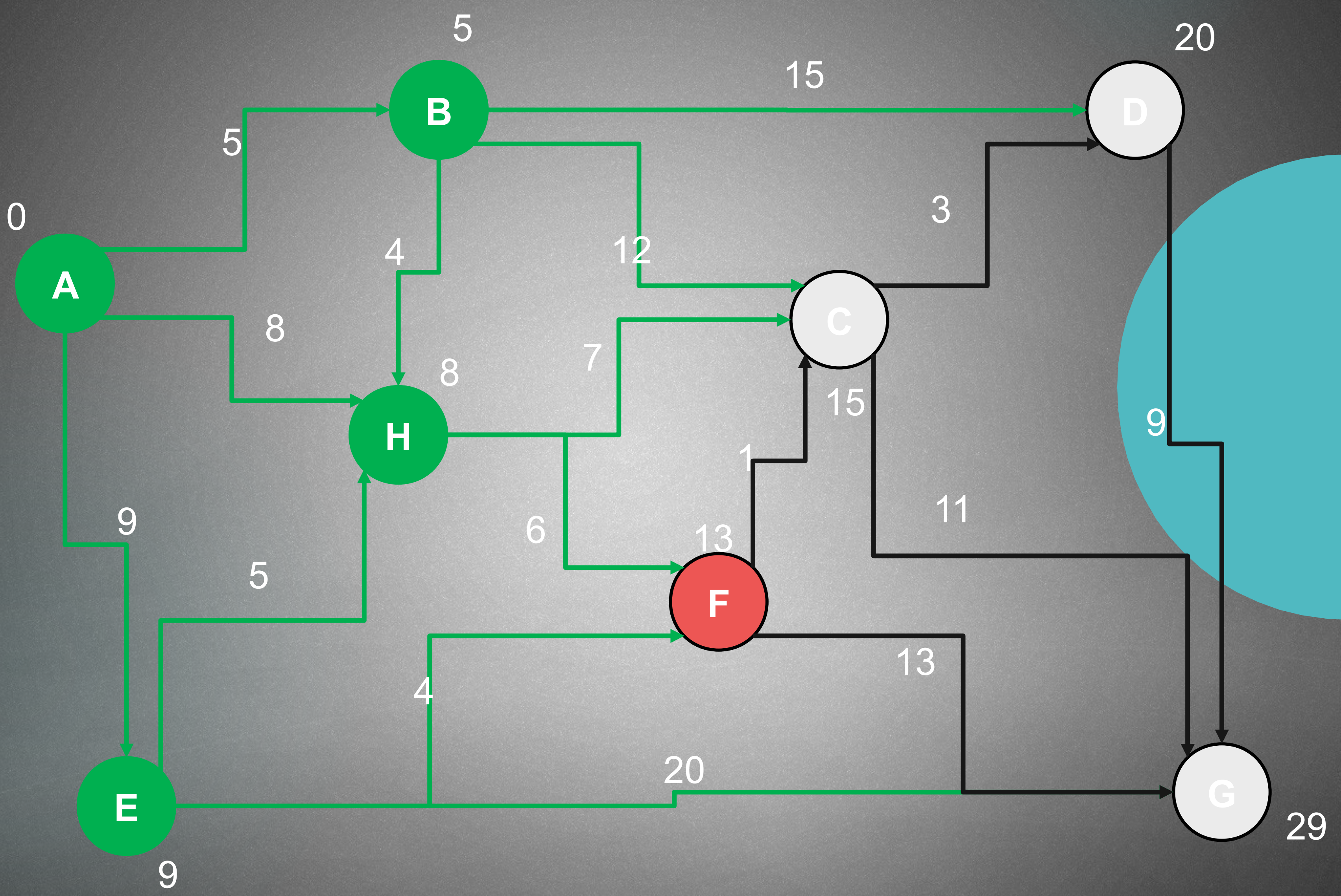
Heap content: C – 15 ; D – 20 ; F – 13 ; G – 29



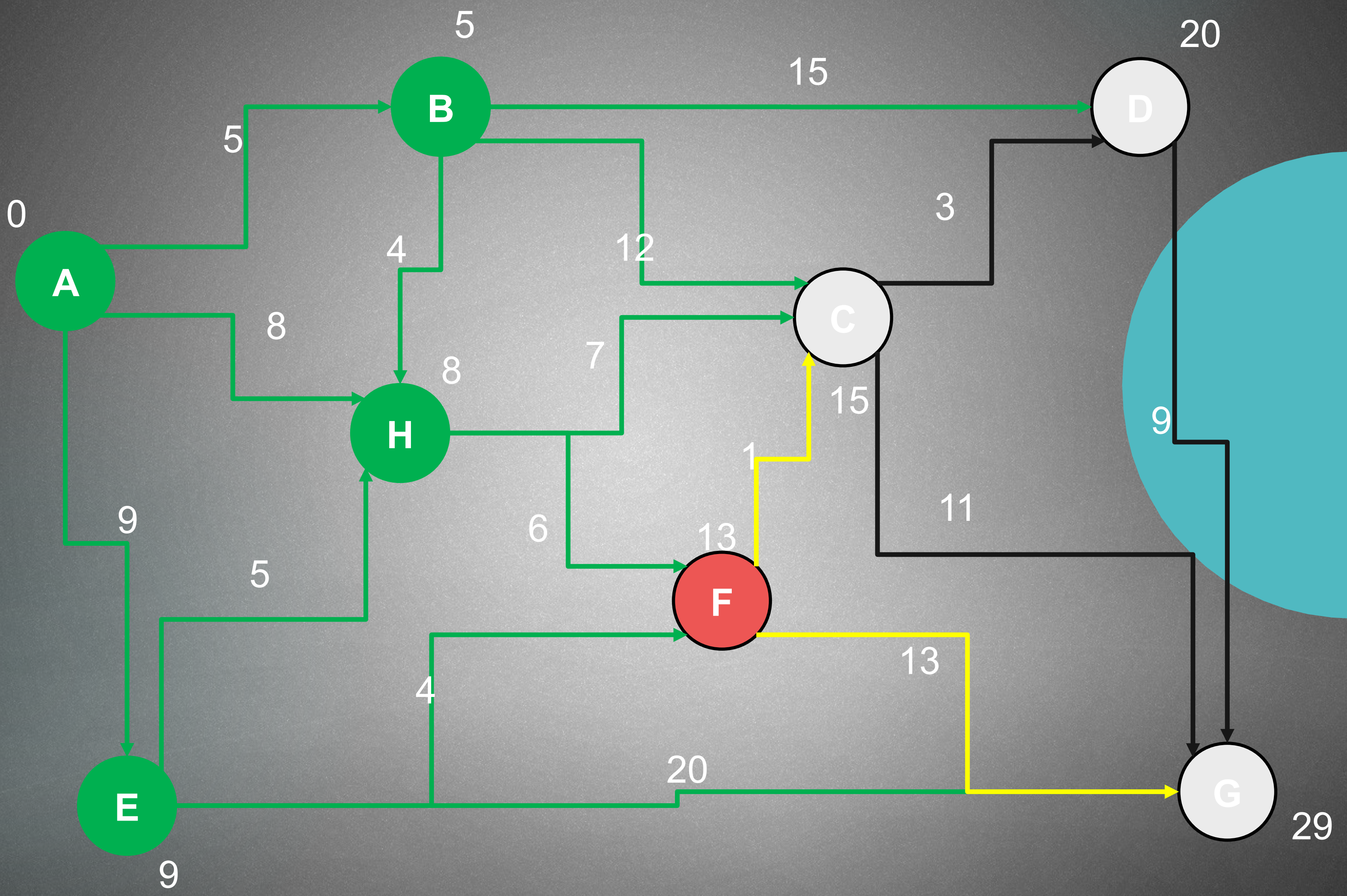
Heap content: C – 15 ; D – 20 ; F – 13 ; G – 29



Heap content: C – 15 ; D – 20 ; **F – 13** ; G – 29

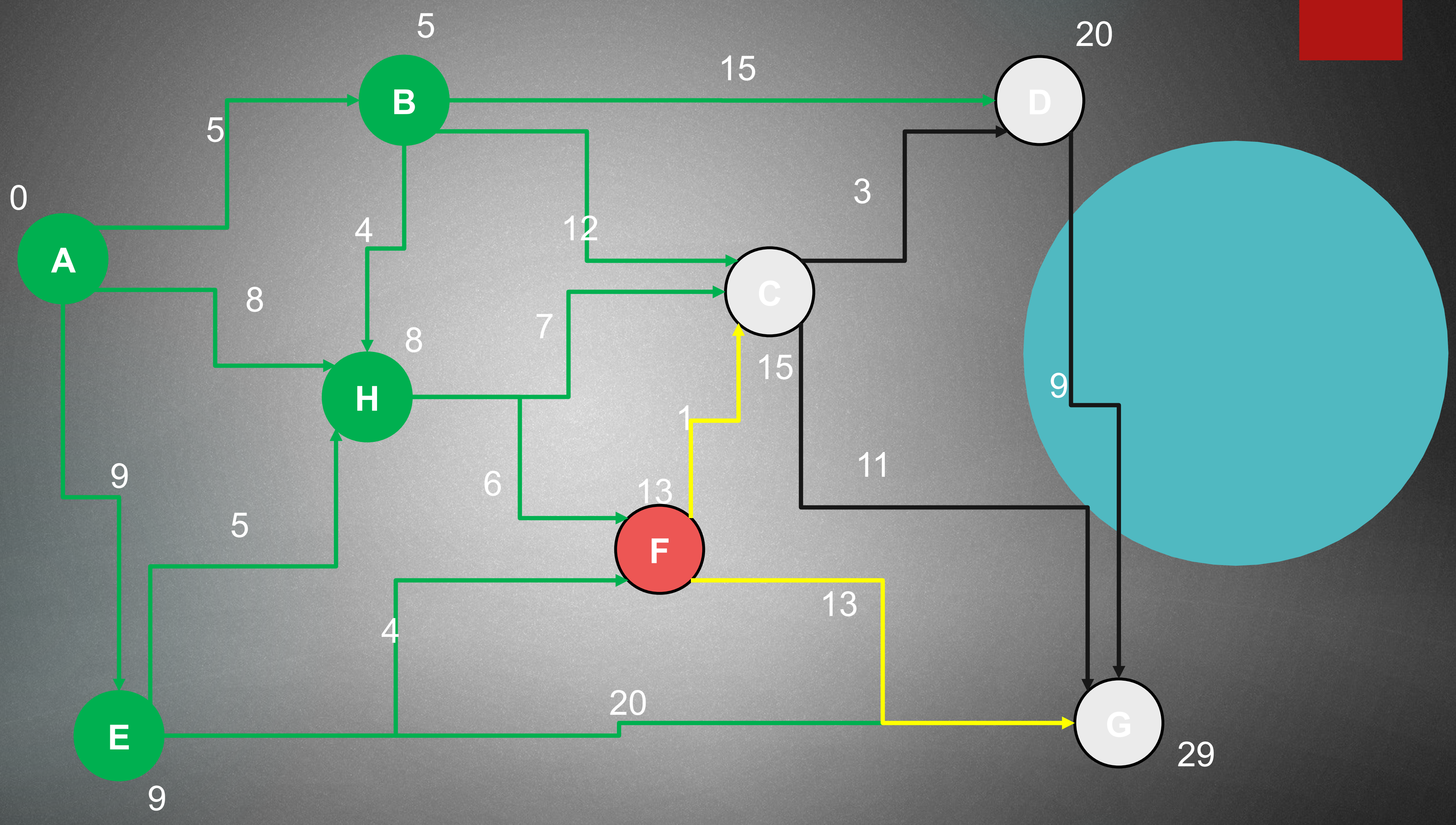


Heap content: C – 15 ; D – 20 ; G – 29



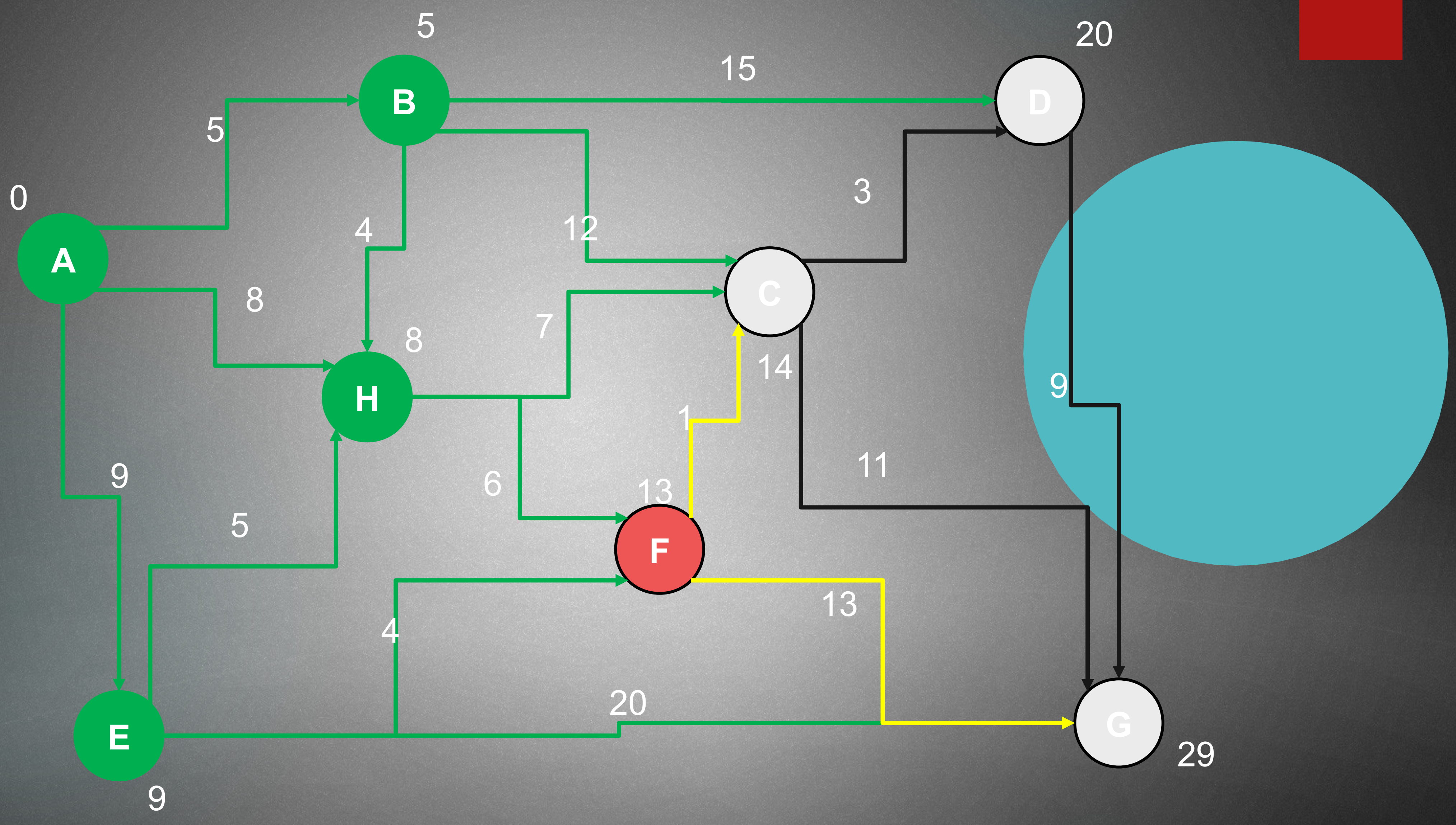
Node C: decide what is smaller 13+1 or 15 ... 14 is smaller so UPDATE

Heap content: C – 15 ; D – 20 ; G – 29



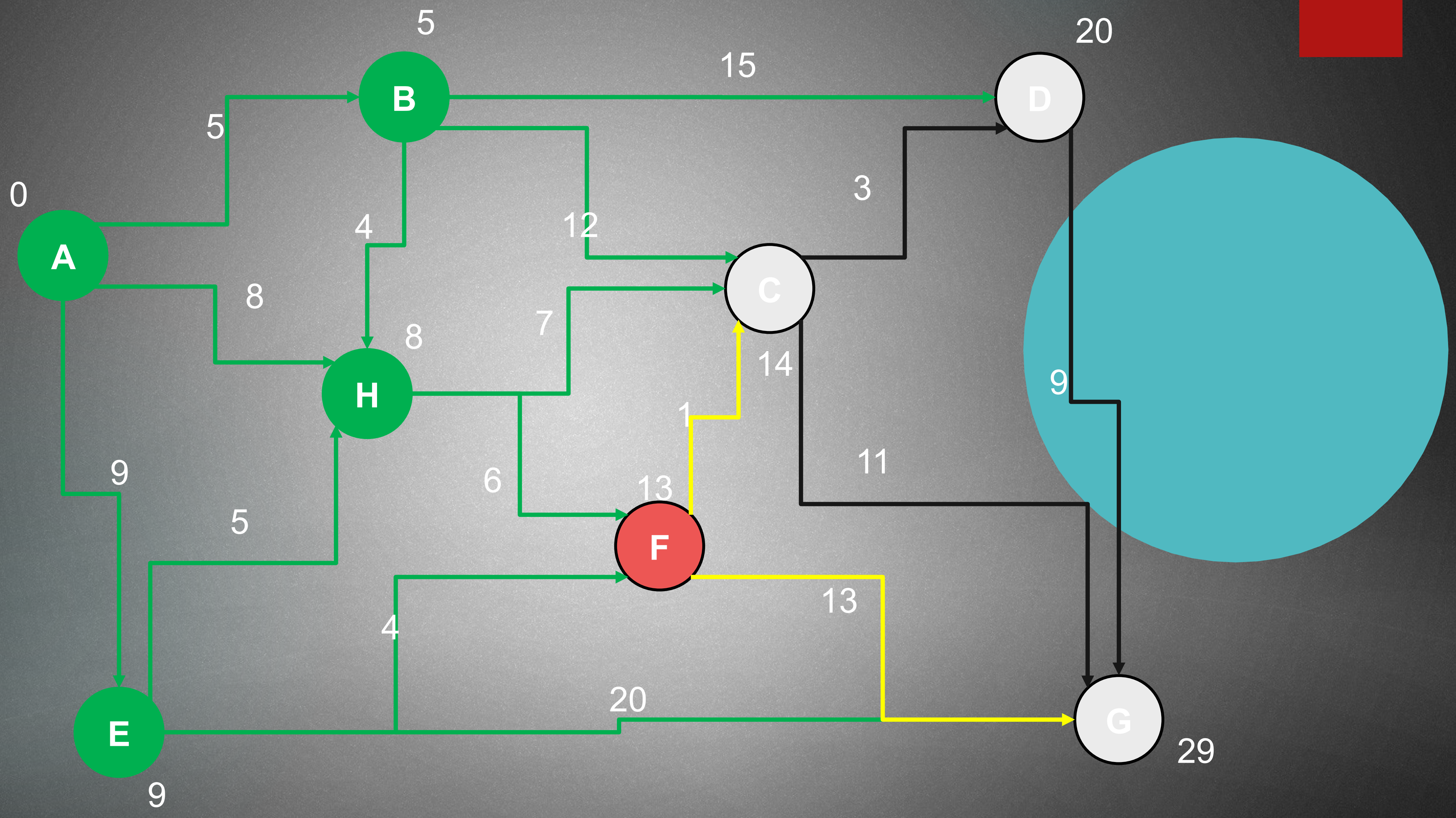
Node C: decide what is smaller 13+1 or 15 ... 14 is smaller so UPDATE

Heap content: C – 14 ; D – 20 ; G – 29



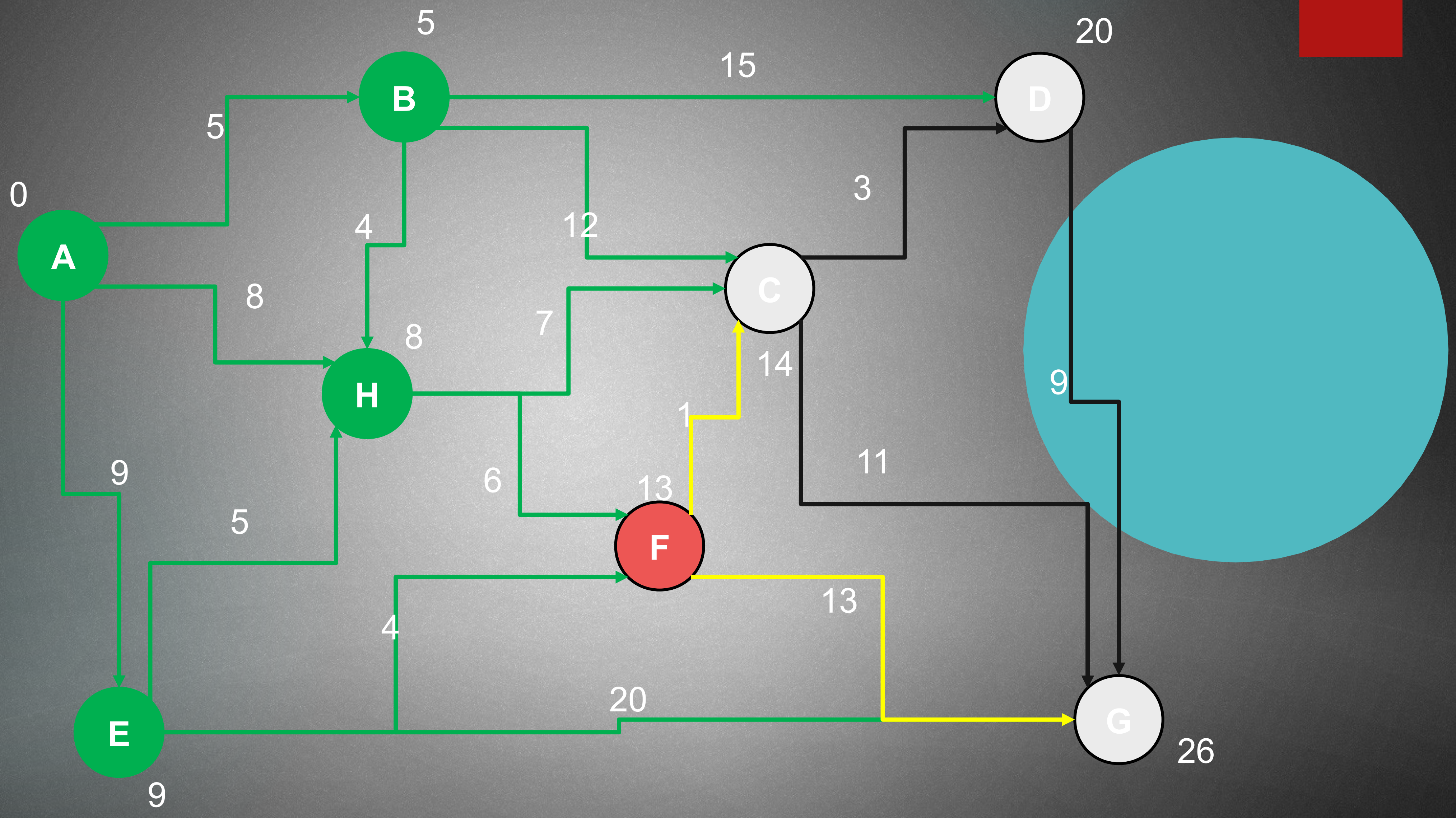
Node G: decide what is smaller 13+13 or 29 ... 26 is smaller so UPDATE

Heap content: C – 14 ; D – 20 ; G – 29

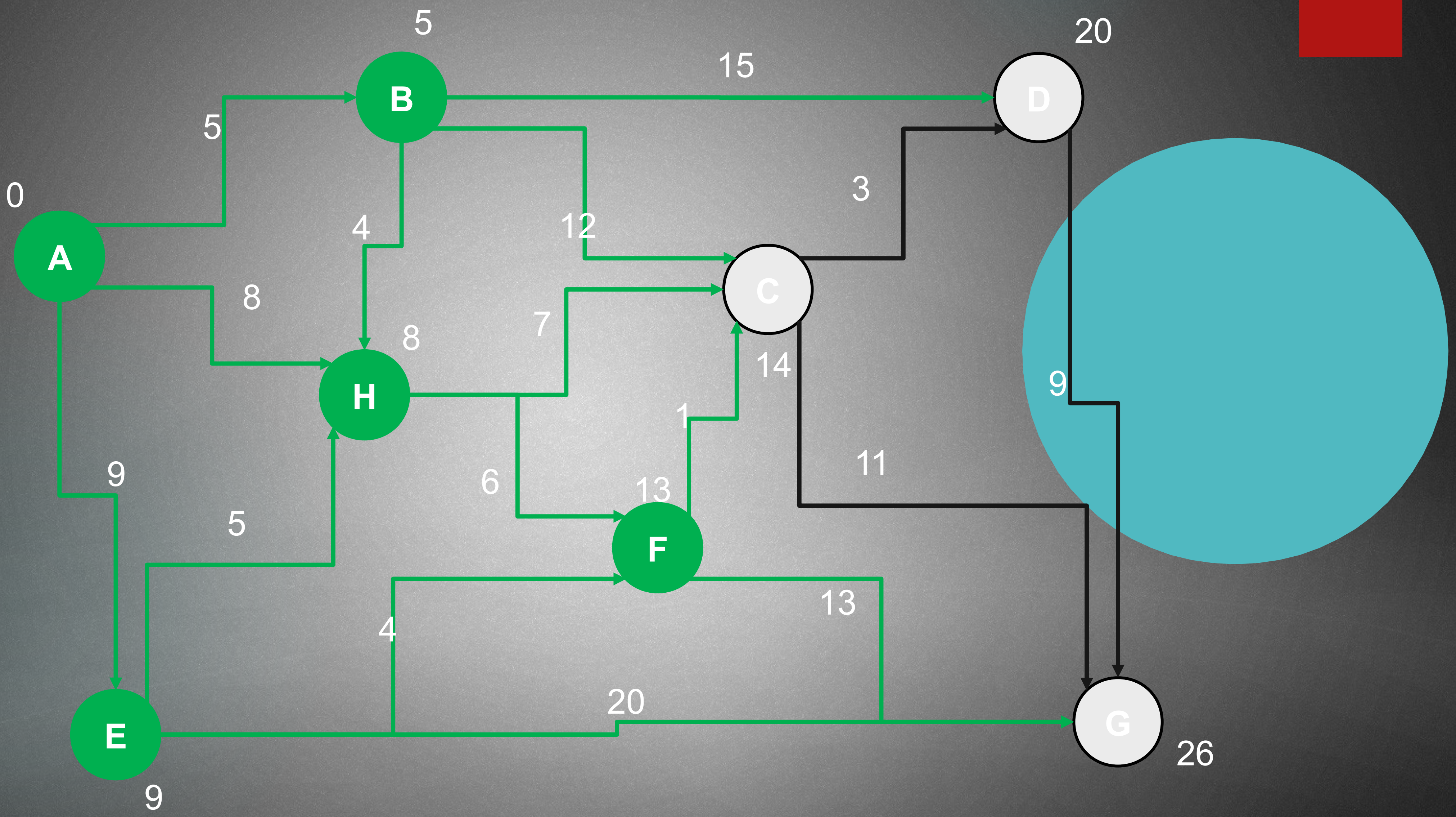


Node G: decide what is smaller 13+13 or 29 ... 26 is smaller so UPDATE

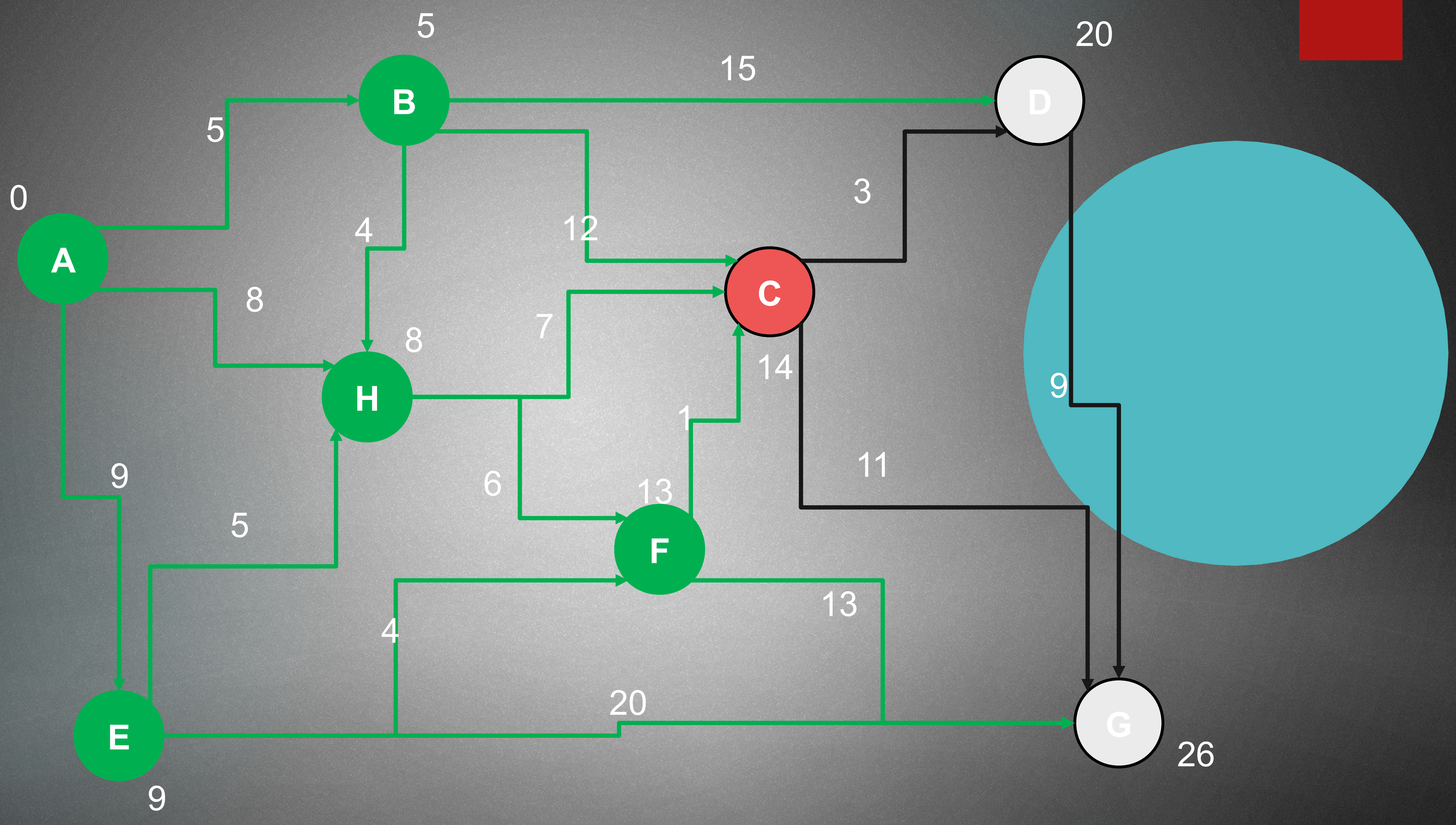
Heap content: C – 14 ; D – 20 ; G – 26



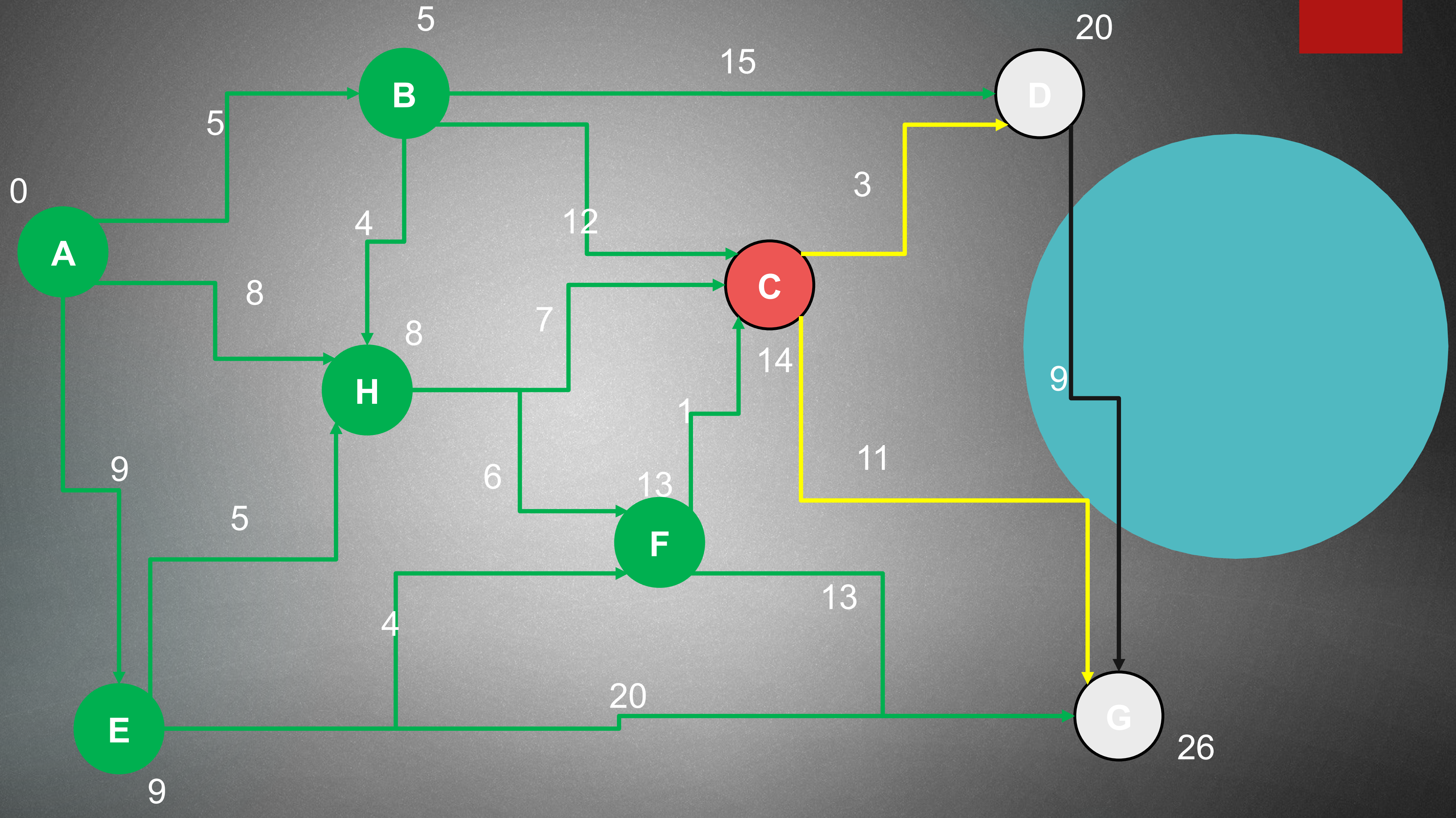
Heap content: C – 14 ; D – 20 ; G – 26



Heap content: **C – 14** ; D – 20 ; G – 26

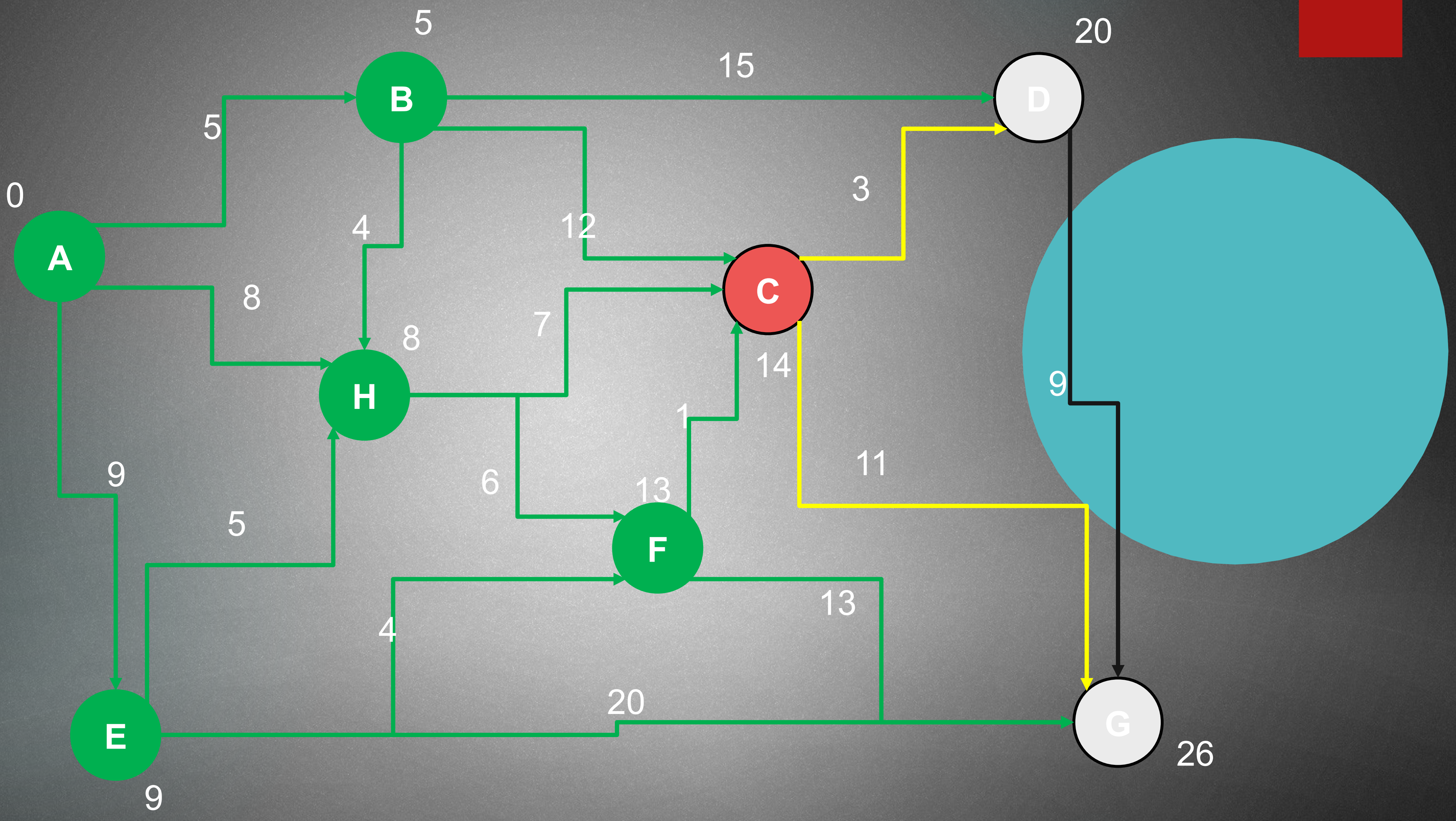


Heap content: D – 20 ; G – 26



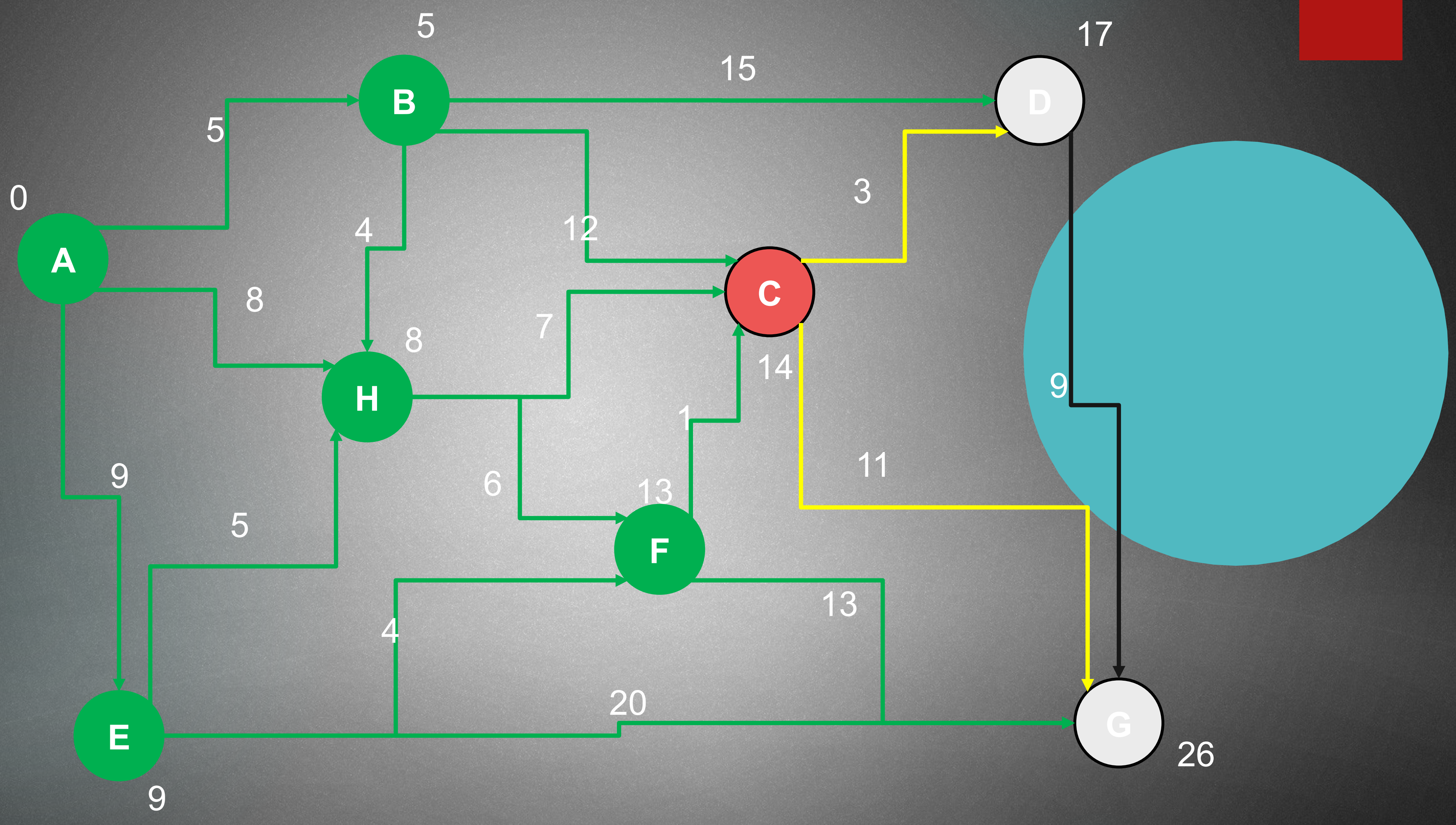
Node D: decide what is smaller 14+3 or 20 ... 17 is smaller so UPDATE

Heap content: D – 20 ; G – 26



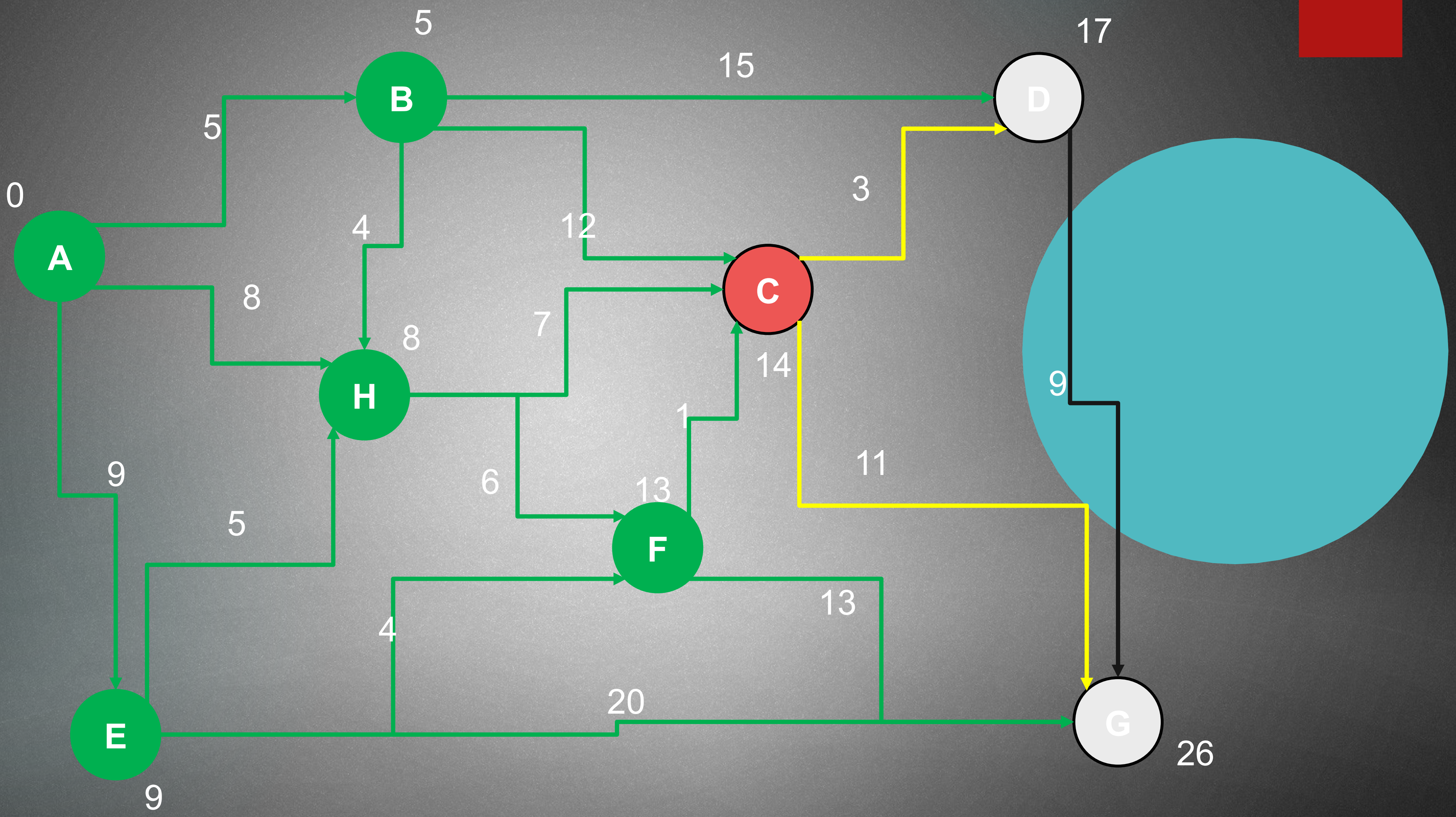
Node D: decide what is smaller 14+3 or 20 ... 17 is smaller so UPDATE

Heap content: D – 17 ; G – 26



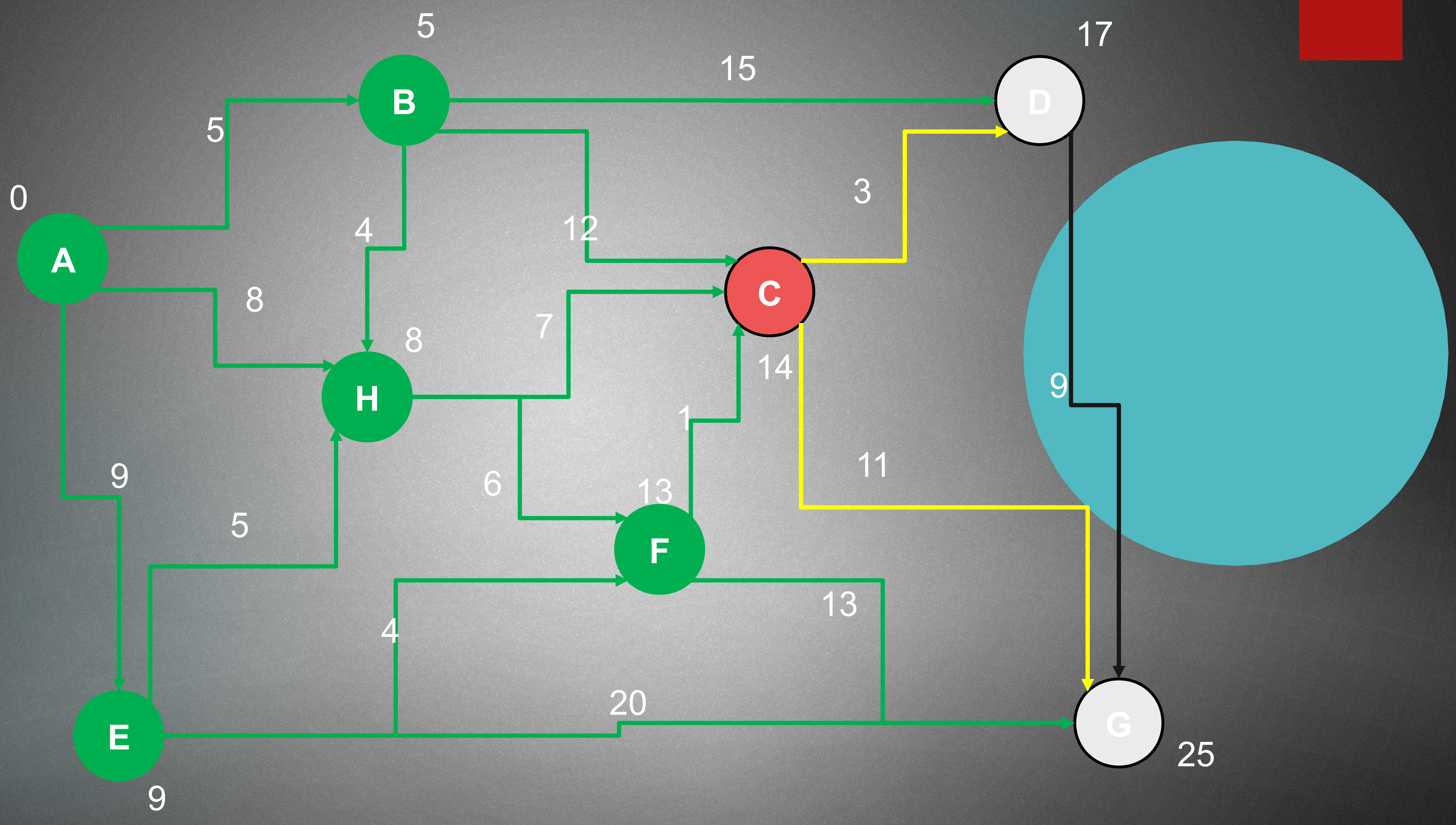
Node G: decide what is smaller 14+11 or 26 ... 25 is smaller so UPDATE

Heap content: D – 17 ; G – 26

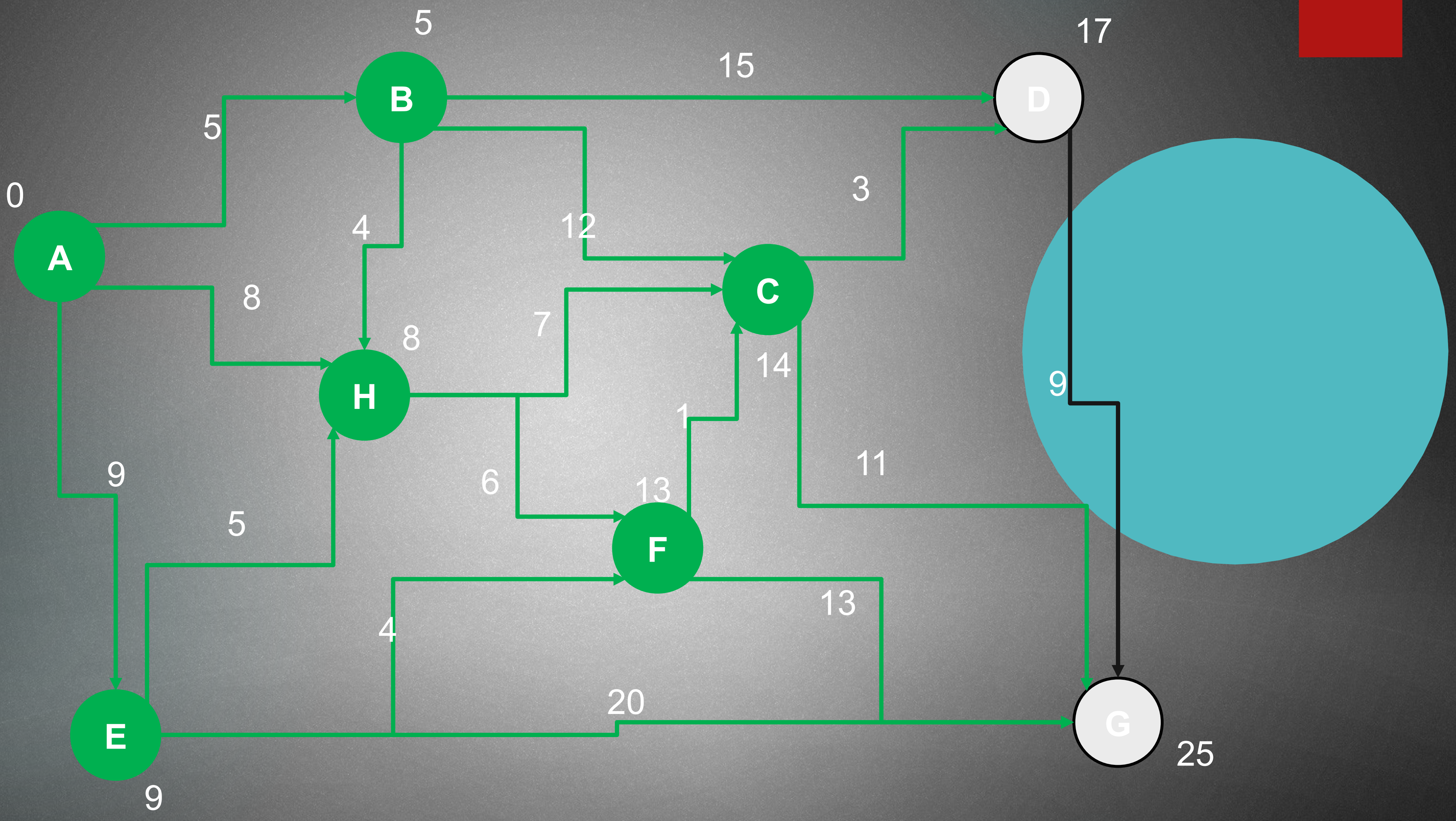


Node G: decide what is smaller 14+11 or 26 ... 25 is smaller so UPDATE

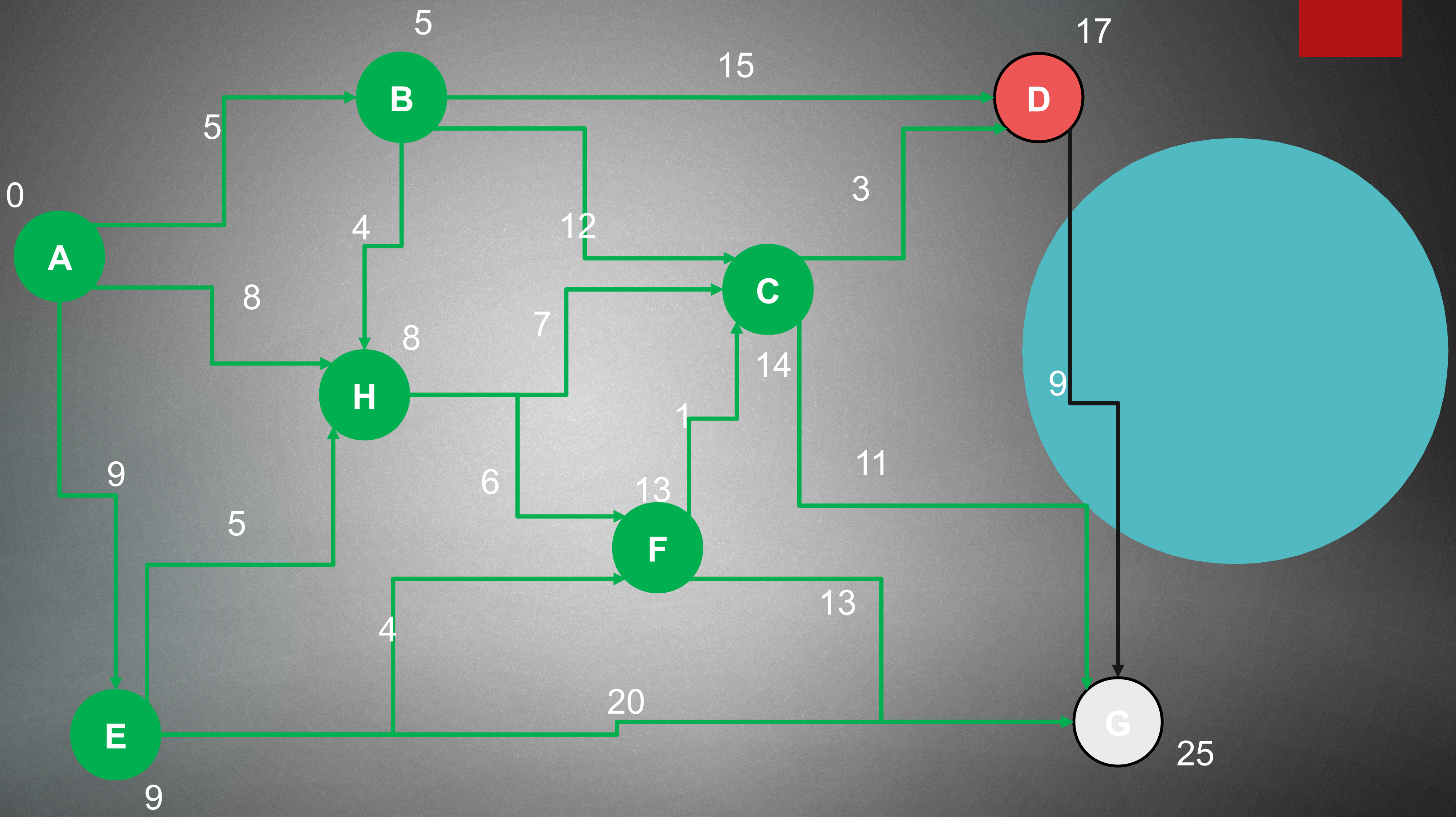
Heap content: D – 17 ; G – 25



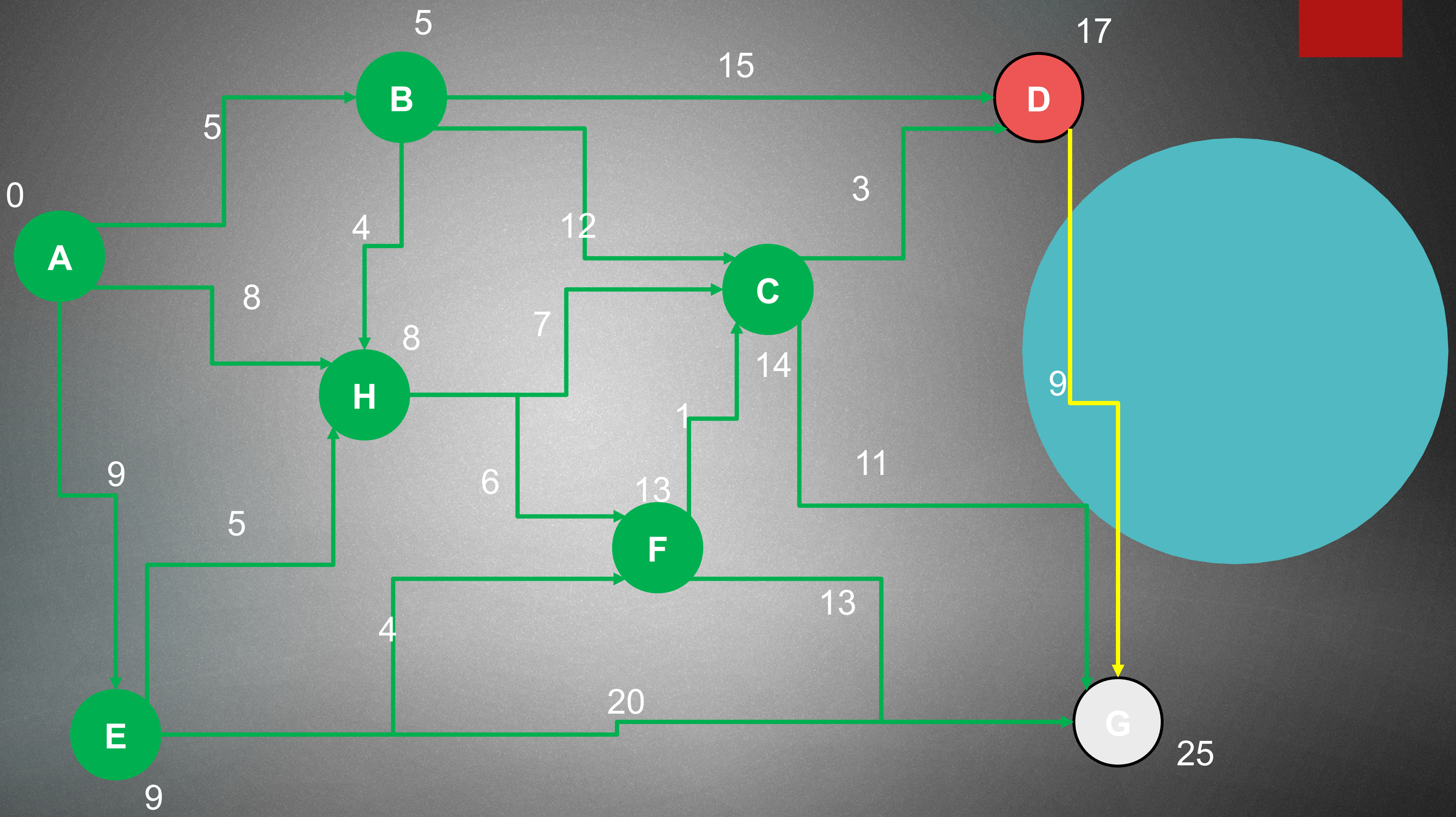
Heap content: D – 17 ; G – 25



Heap content: **D – 17** ; G – 25

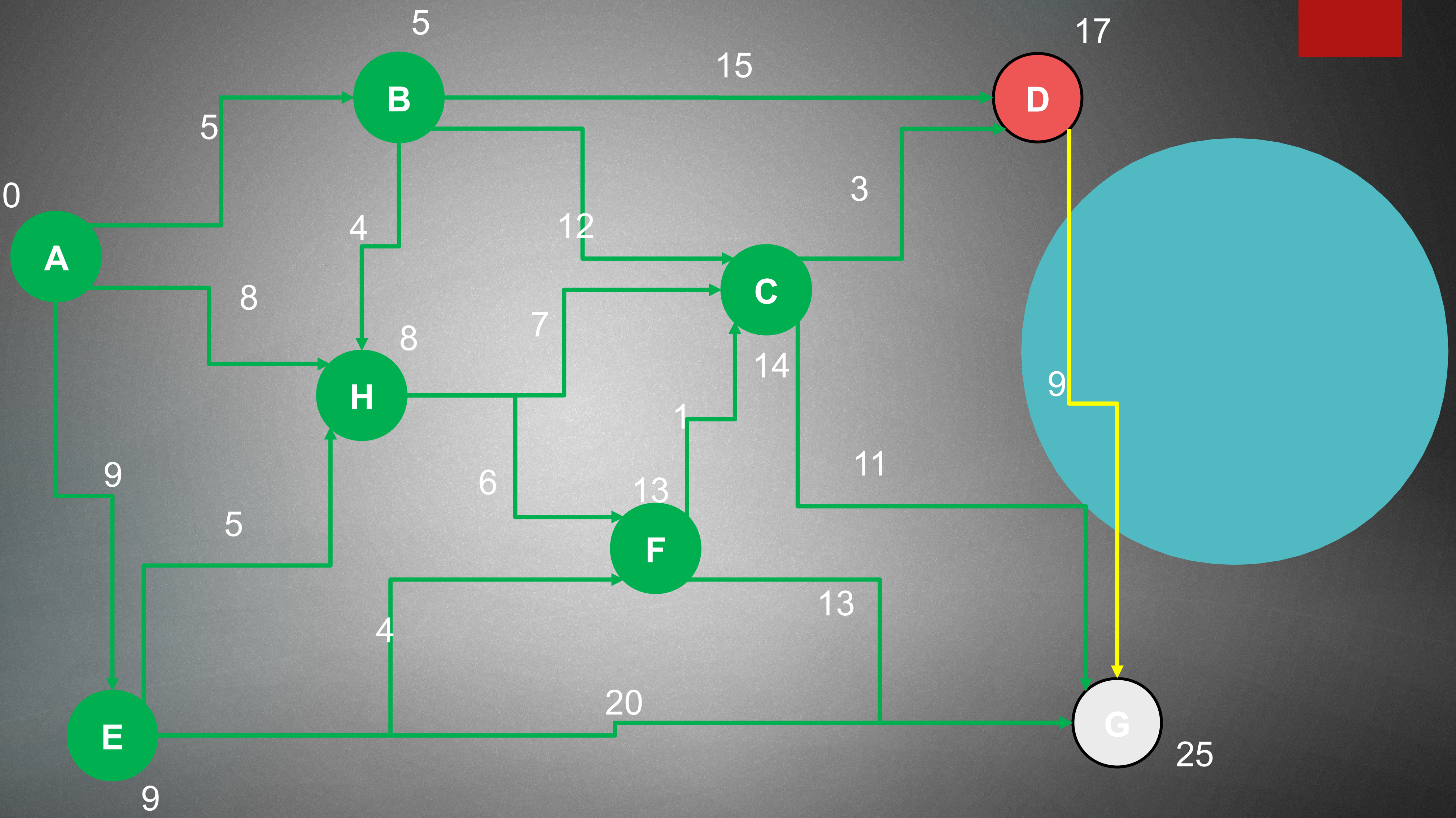


Heap content: G – 25

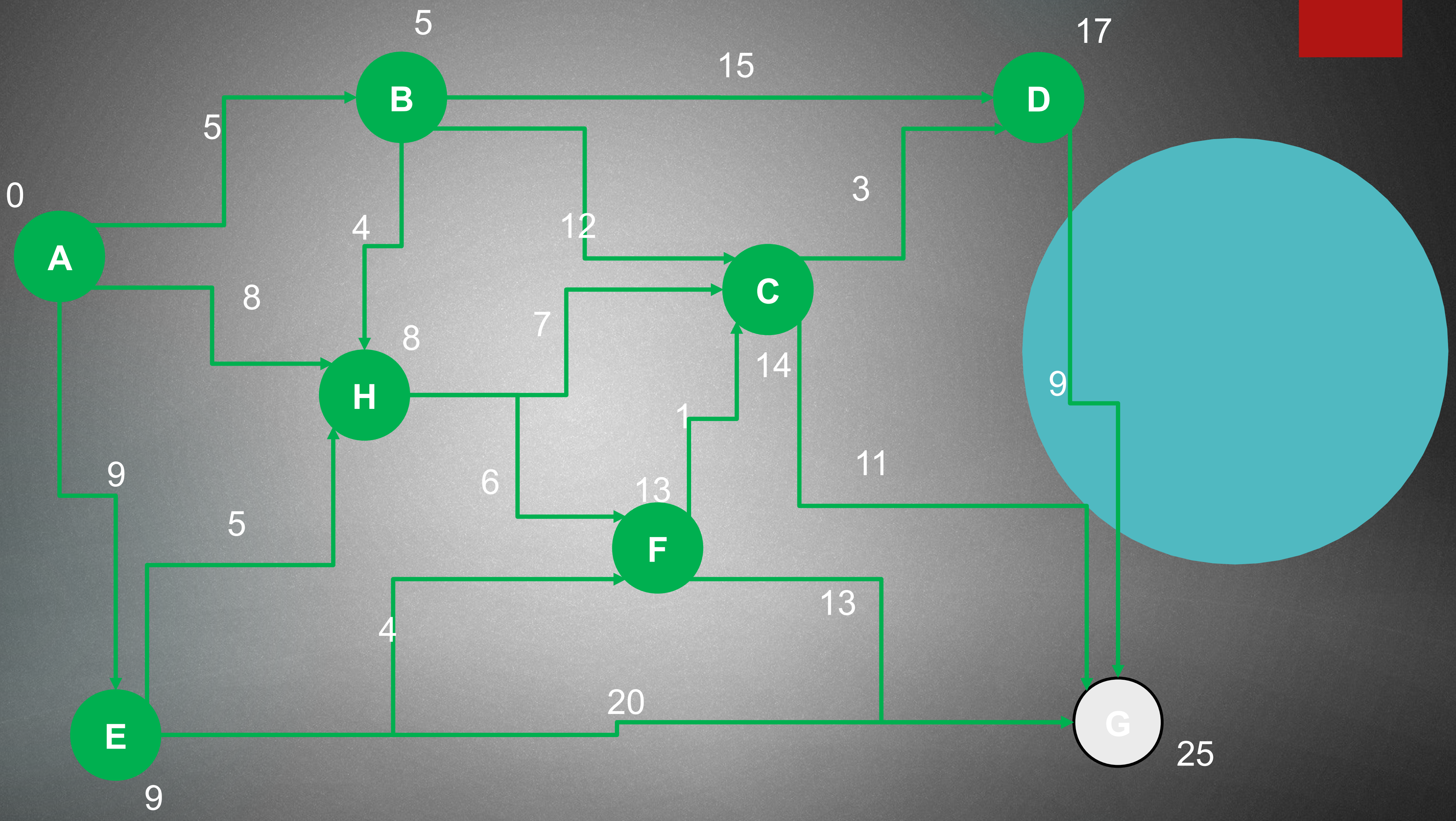


Node G: decide what is smaller 15+17 or 25 ... 25 is smaller so DO NOT UPDATE

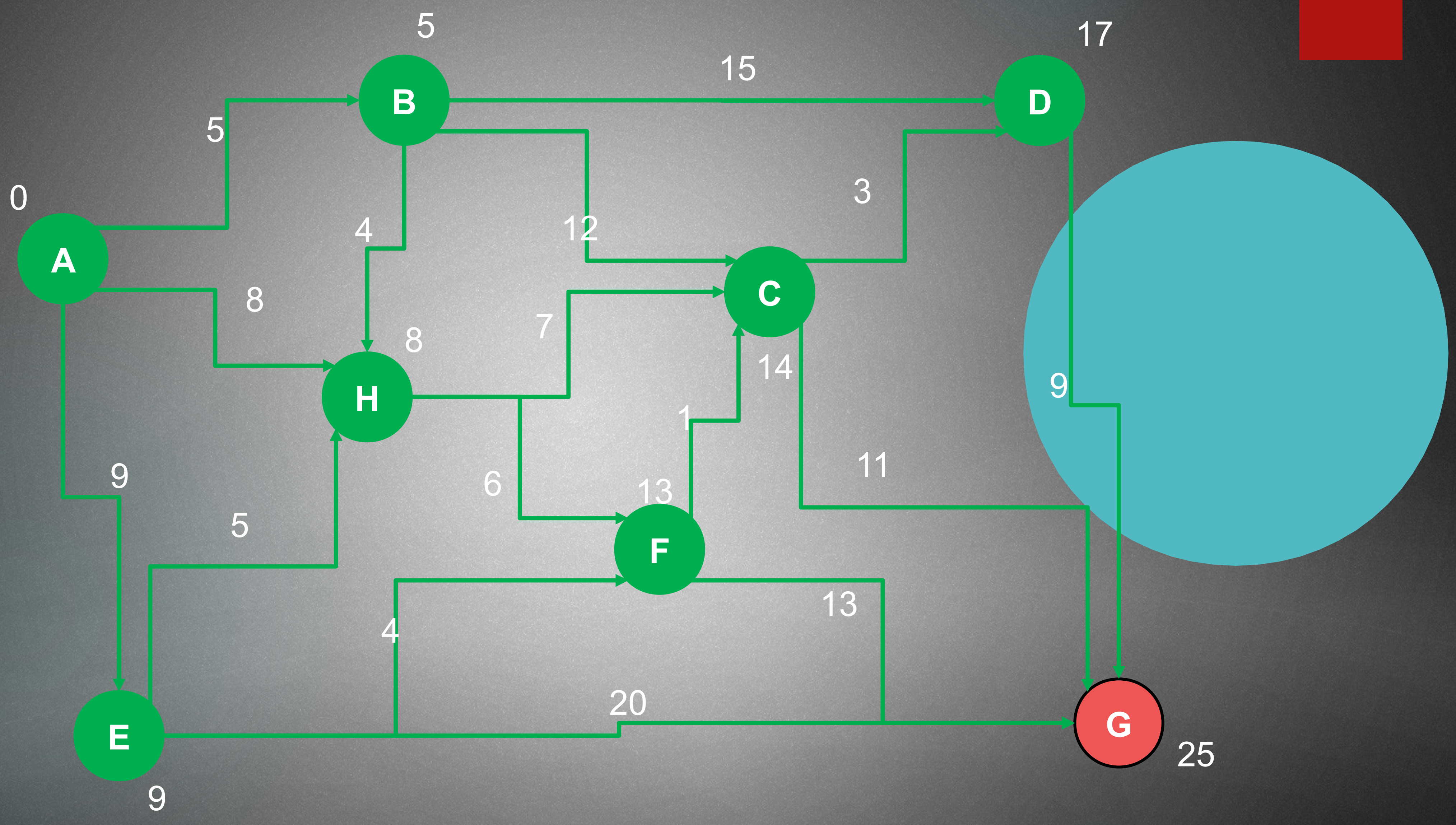
Heap content: G – 25



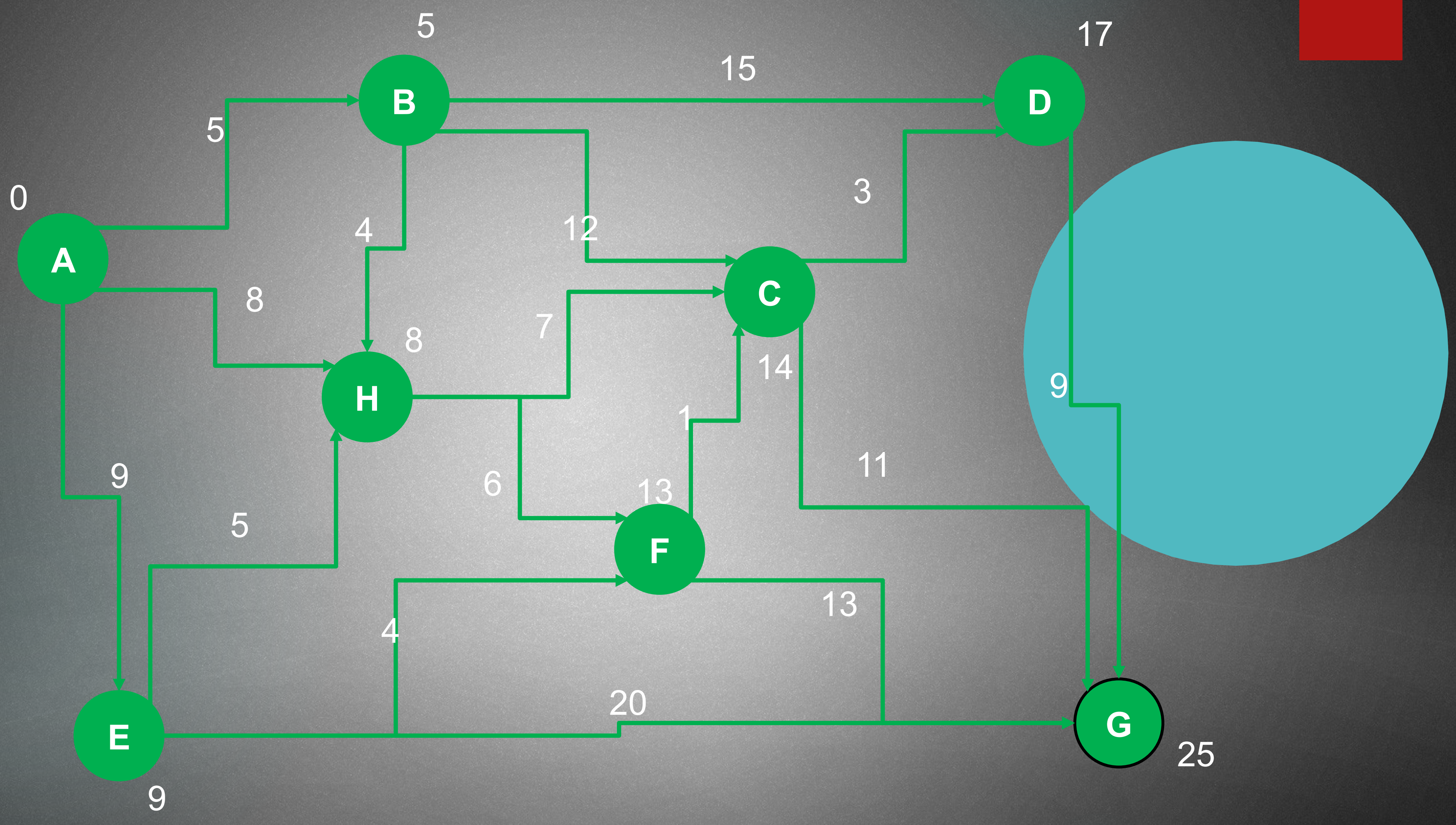
Heap content: G – 25



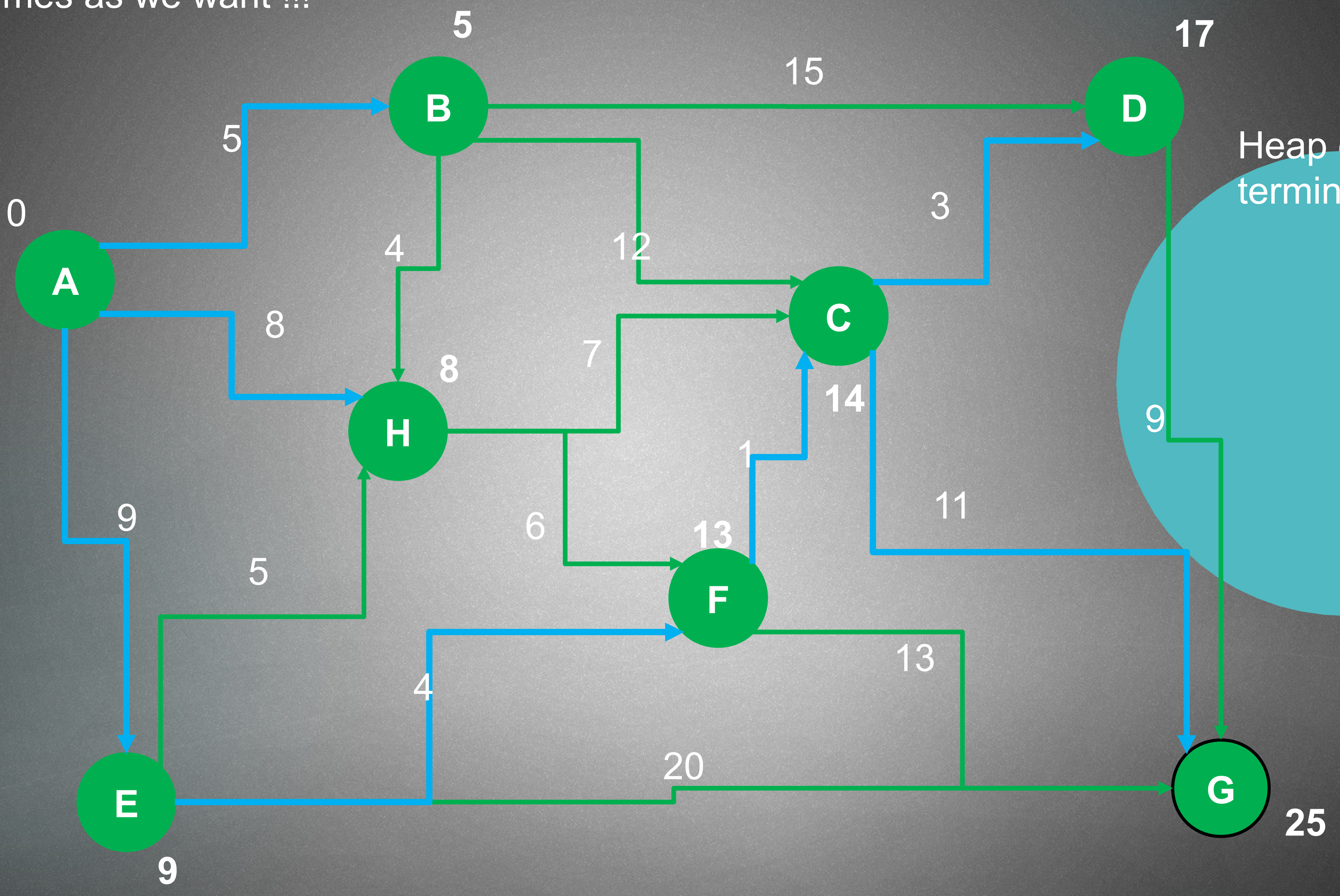
Heap content: **G – 25**



Heap content: empty so terminate the algorithm !!!



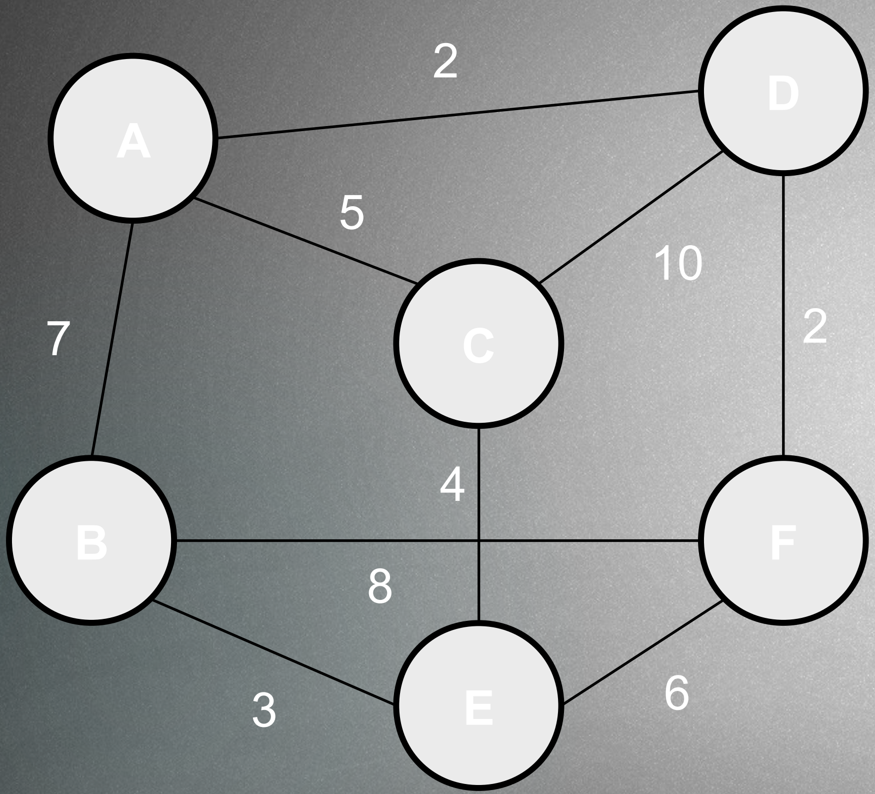
We have constructed the shortest path tree: we just have to calculated once, than reuse it as many times as we want !!!

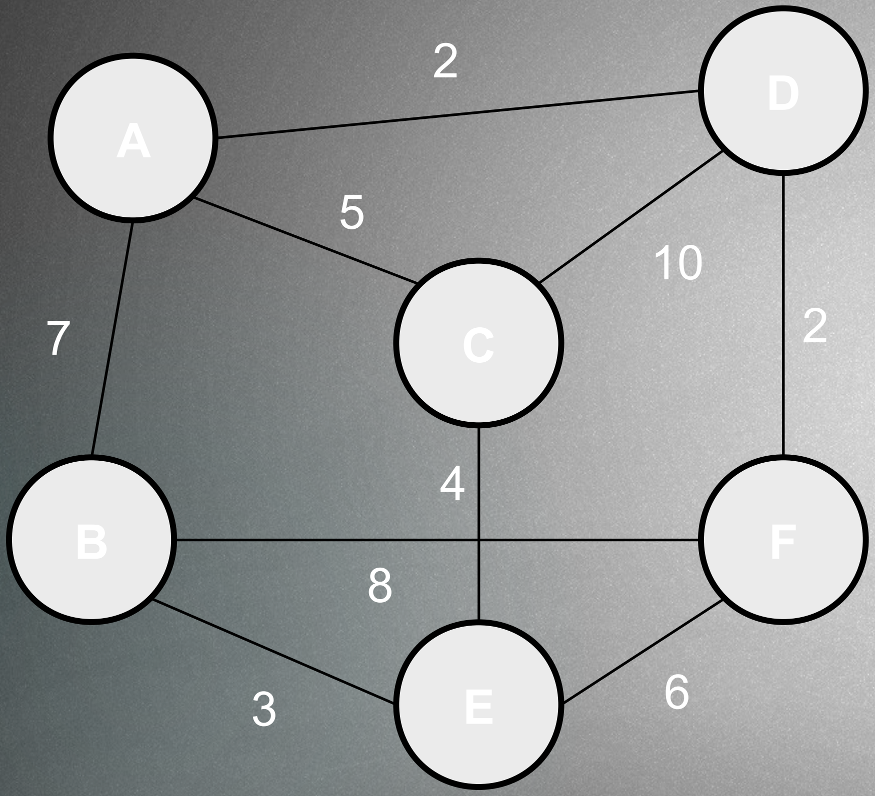


Heap content: empty so terminate the algorithm !!!

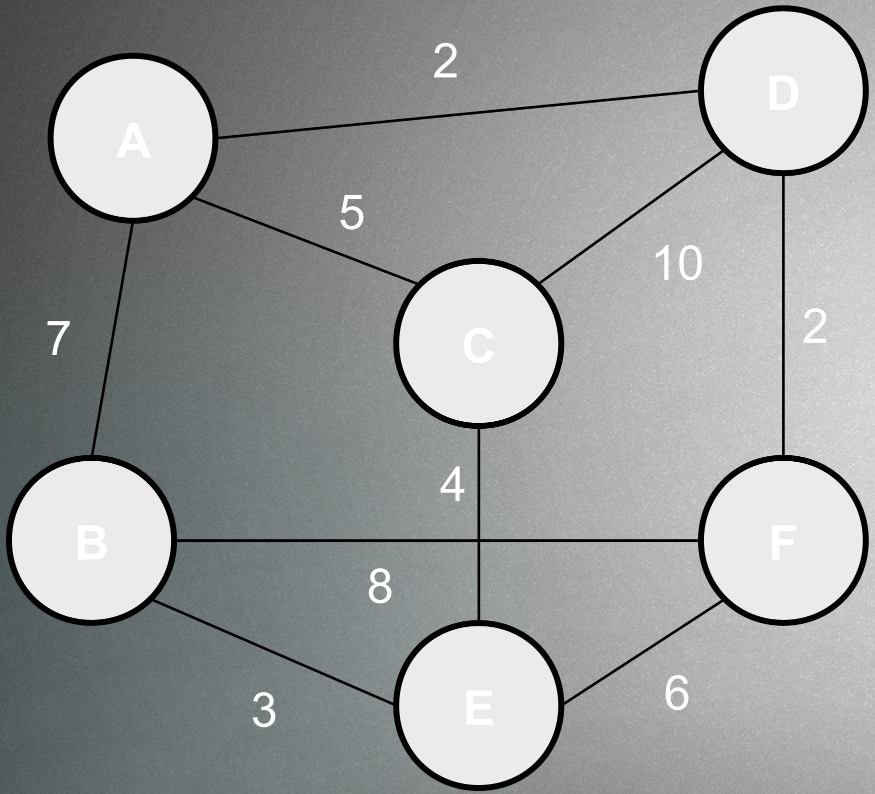
DIJKSTRA ALGORITHM

SHORTEST PATH – with adjacency matrix

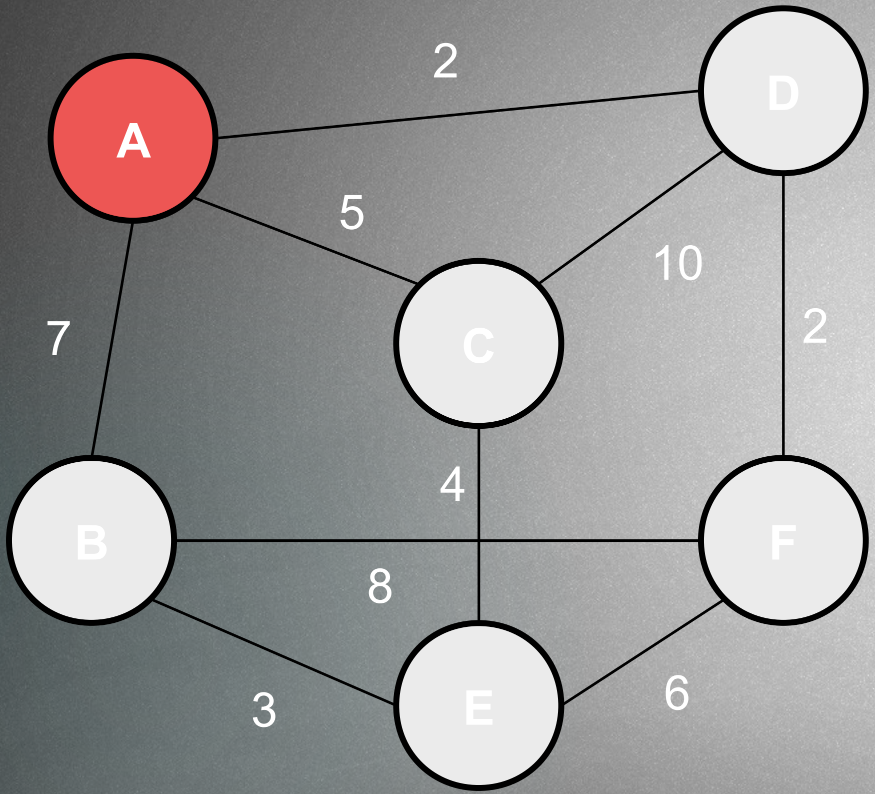

A
B
C
D
E
F
2
5
7
10
2
4
8
3
6



$$
\begin{array}{c c}
 & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} &
\left[\begin{array}{cccccc}
0 & 7 & 5 & 2 & 0 & 0 \\
7 & 0 & 0 & 0 & 3 & 0 \\
5 & 0 & 0 & 10 & 4 & 0 \\
2 & 0 & 10 & 0 & 0 & 2 \\
0 & 3 & 4 & 0 & 0 & 6 \\
0 & 8 & 0 & 2 & 6 & 0
\end{array}\right]
\end{array}
$$

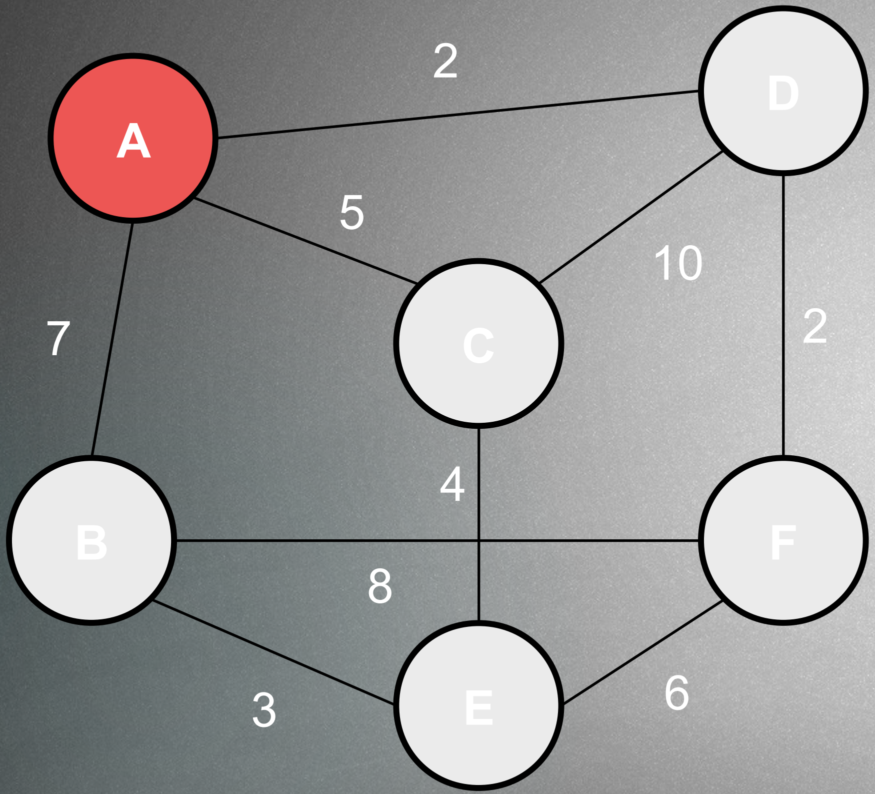


v	A	B	C	D	E	F



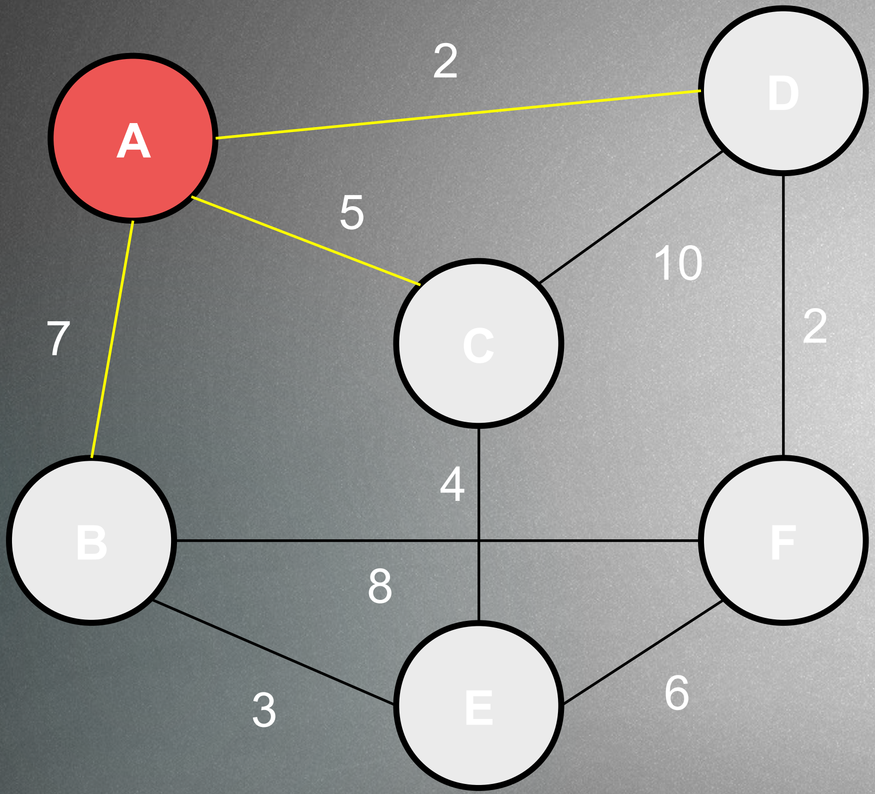
v	A	B	C	D	E	F
A	0	inf	inf	inf	inf	inf

The starting vertex is node A + initialize all the other distances to be infinity
We track: the minimum distance + where did we come here (predecessor)



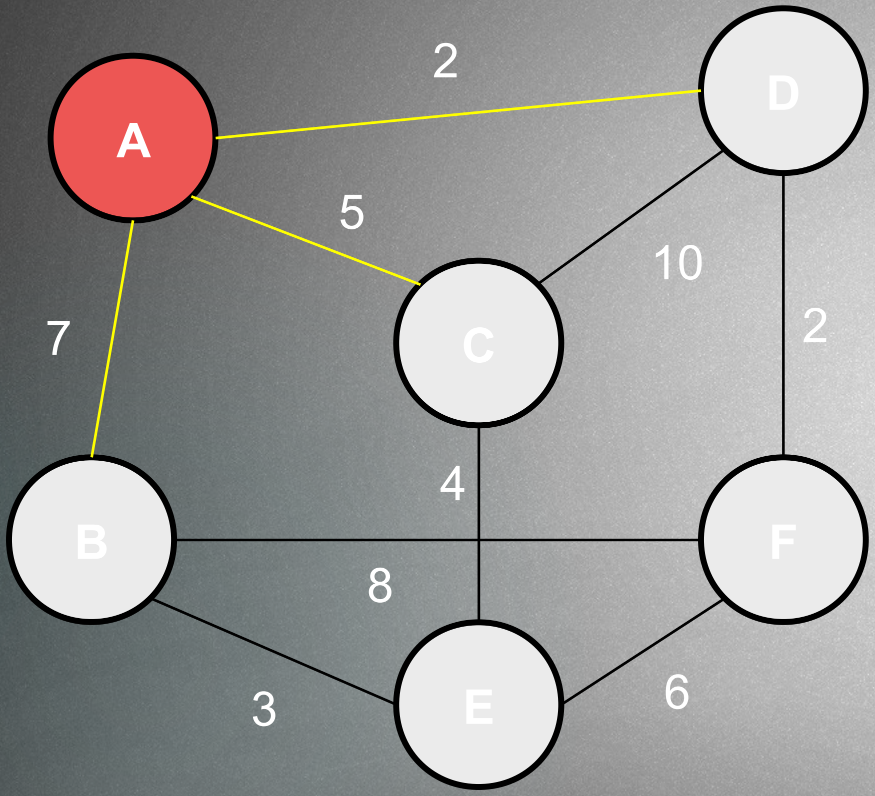
v	A	B	C	D	E	F
A	0	inf	inf	inf	inf	inf

On every iteration we consider the possible routes we are able to take



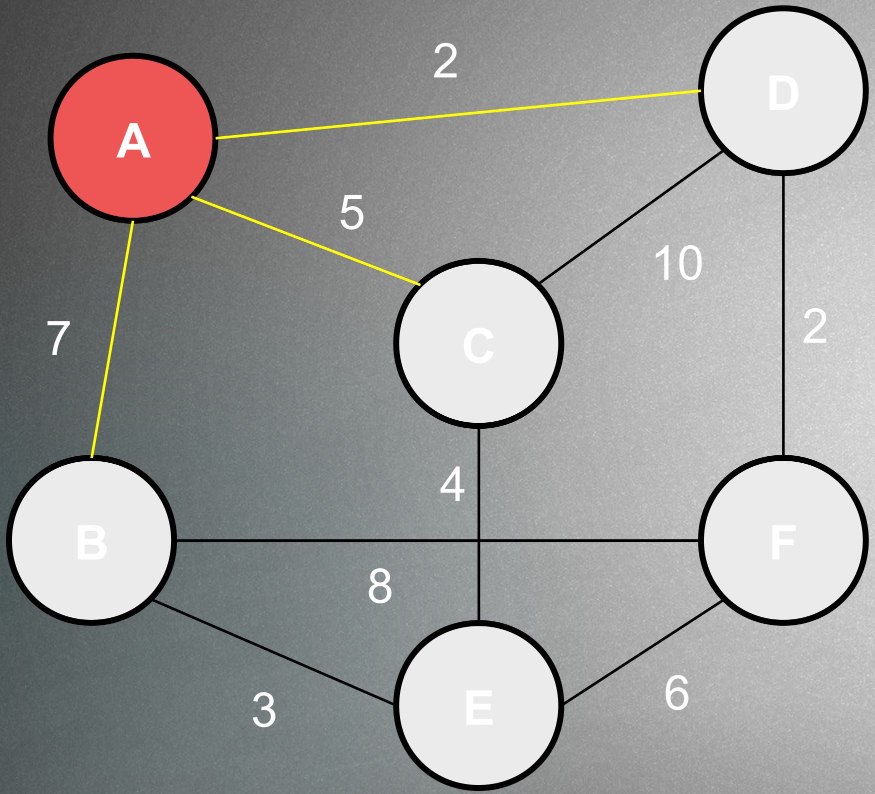
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf

On every iteration we consider the possible routes we are able to take



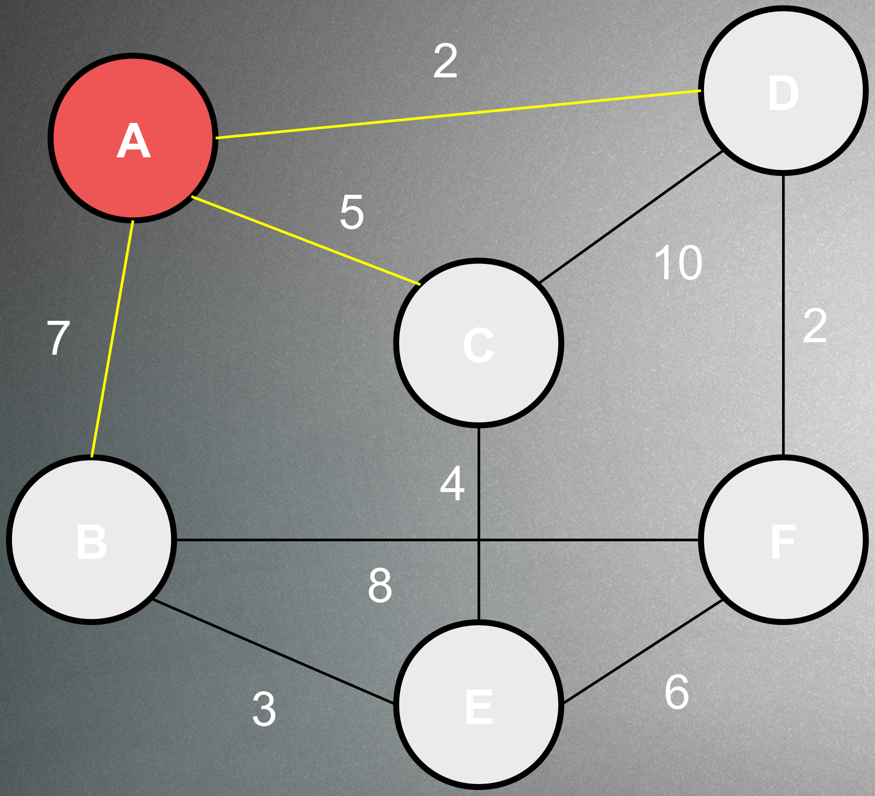
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A						

On every iteration we consider the possible routes we are able to take



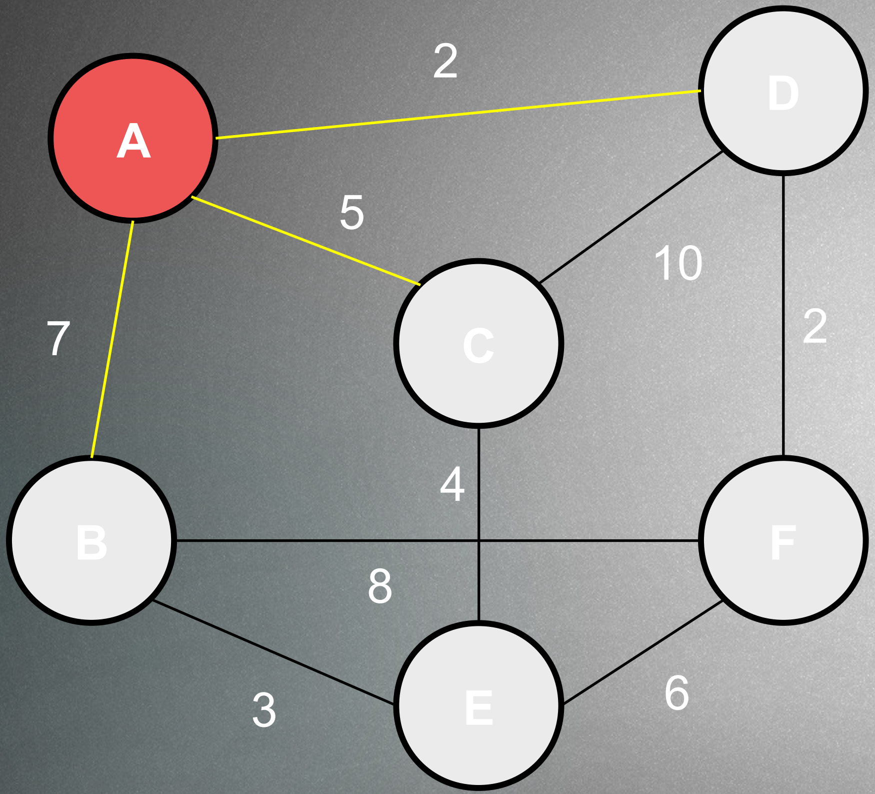
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7				

We have to colculate: min(inf,7) for node B



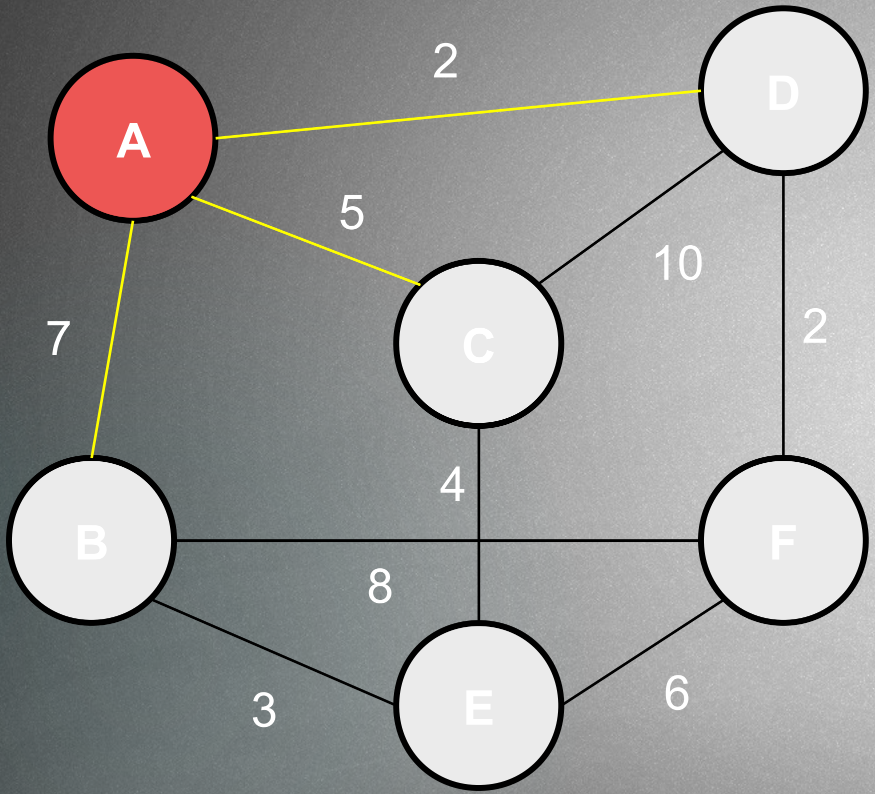
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5			

We have to colculate: min(inf,5) for node C



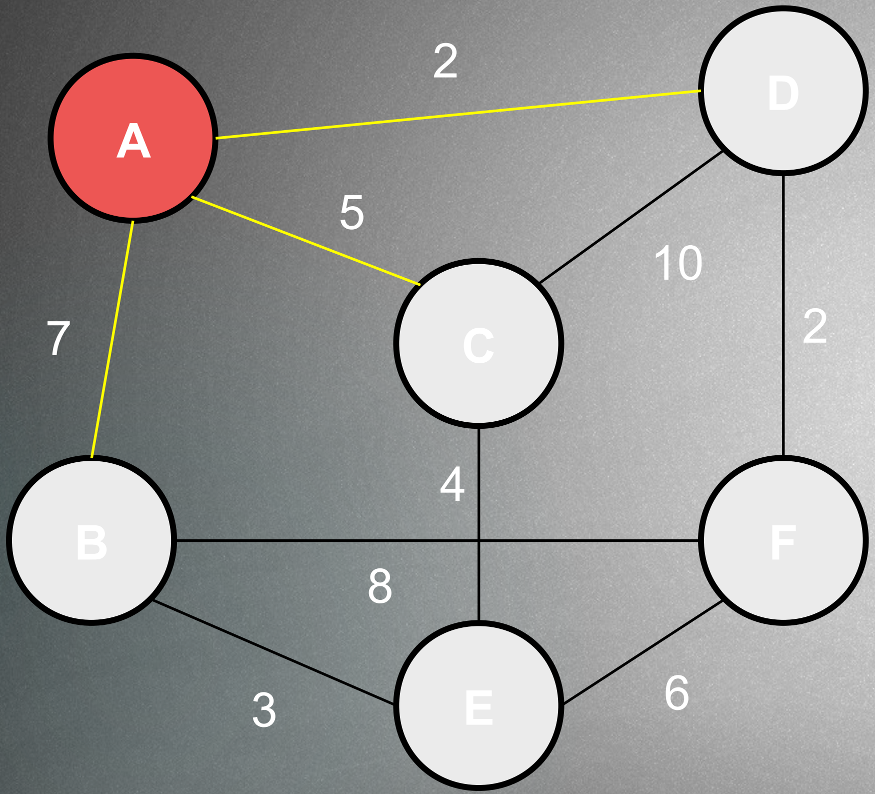
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	2		

We have to colculate: min(inf,2) for node D



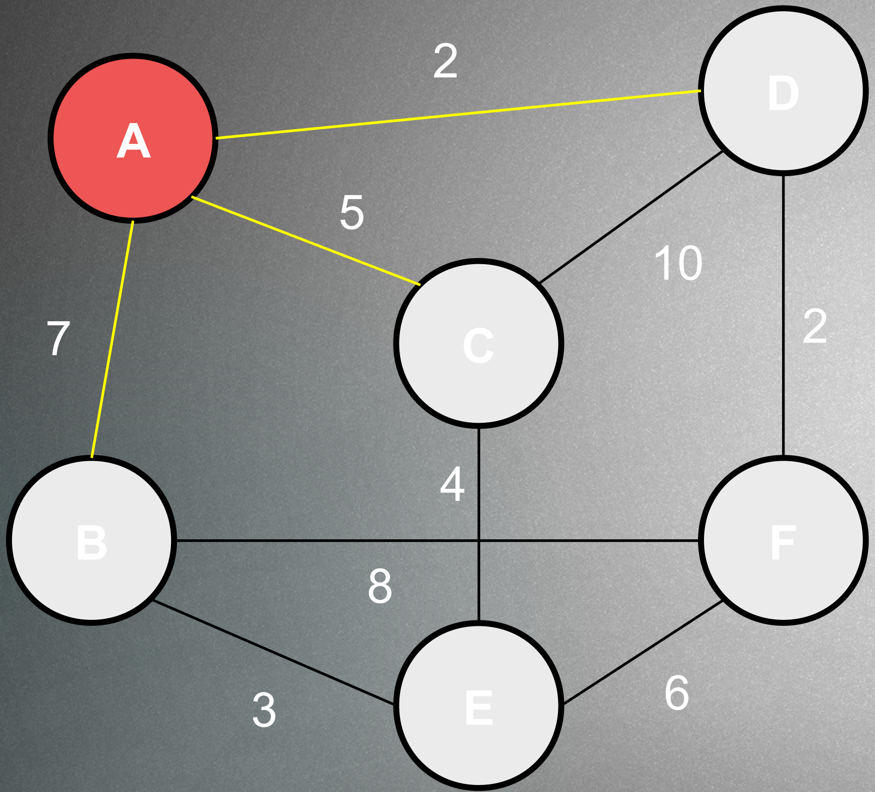
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	2	inf	inf

We can not reach E and F at the moment: they are infinitely far away



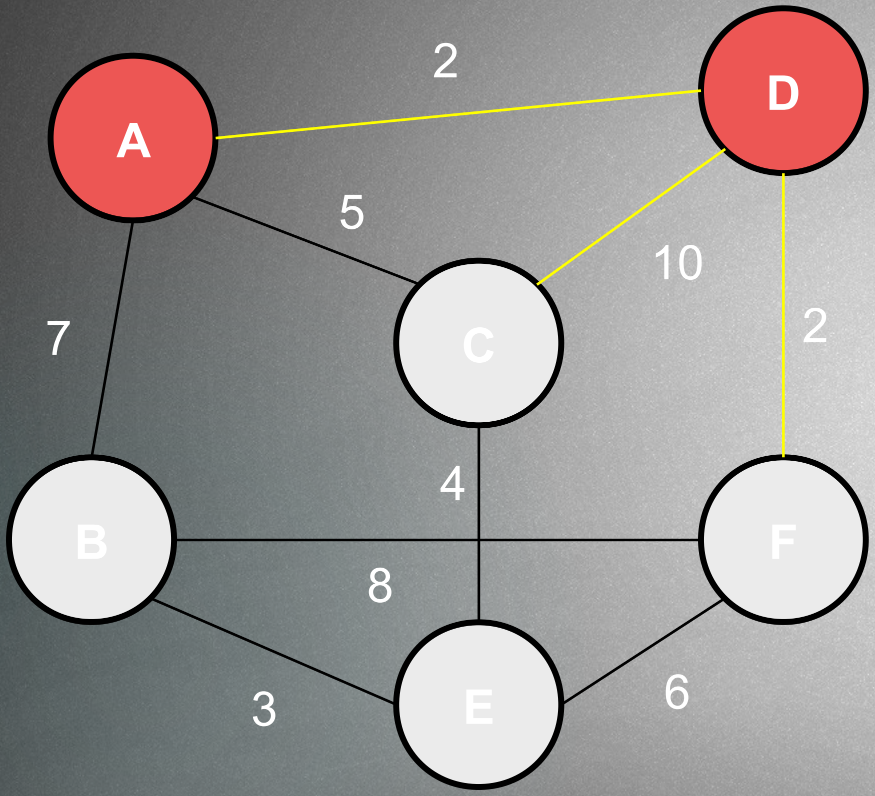
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf

On every iteration we consider the possible routes we are able to take
+ we calculate the minimum value in every row



v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf

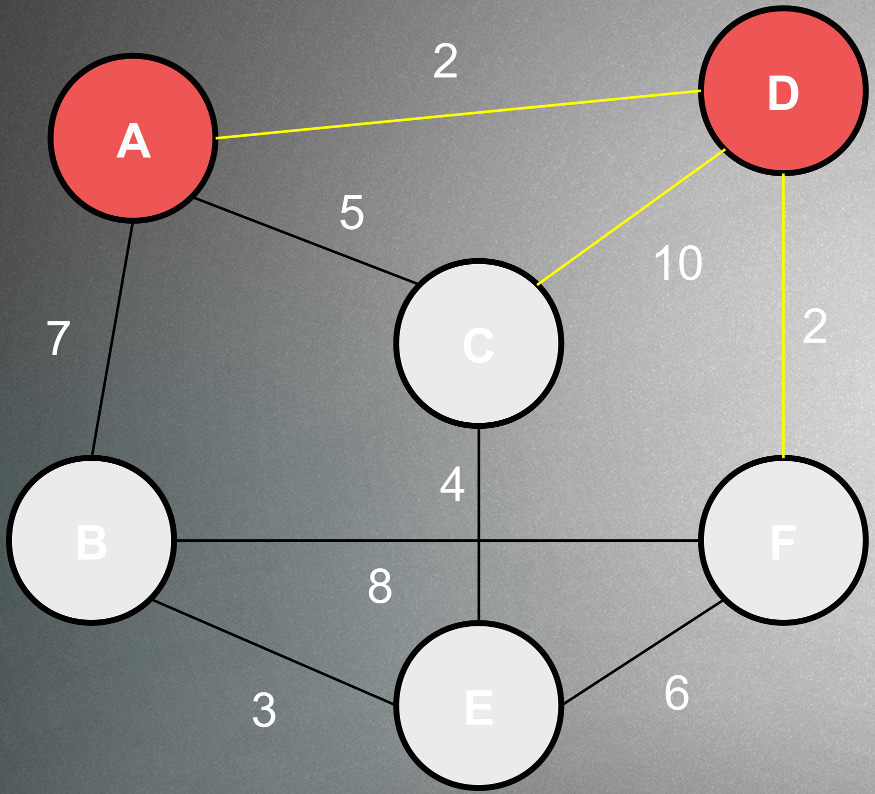
On every iteration we consider the possible routes we are able to take
+ we calculate the minimum value in every row – we hop there



v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D						

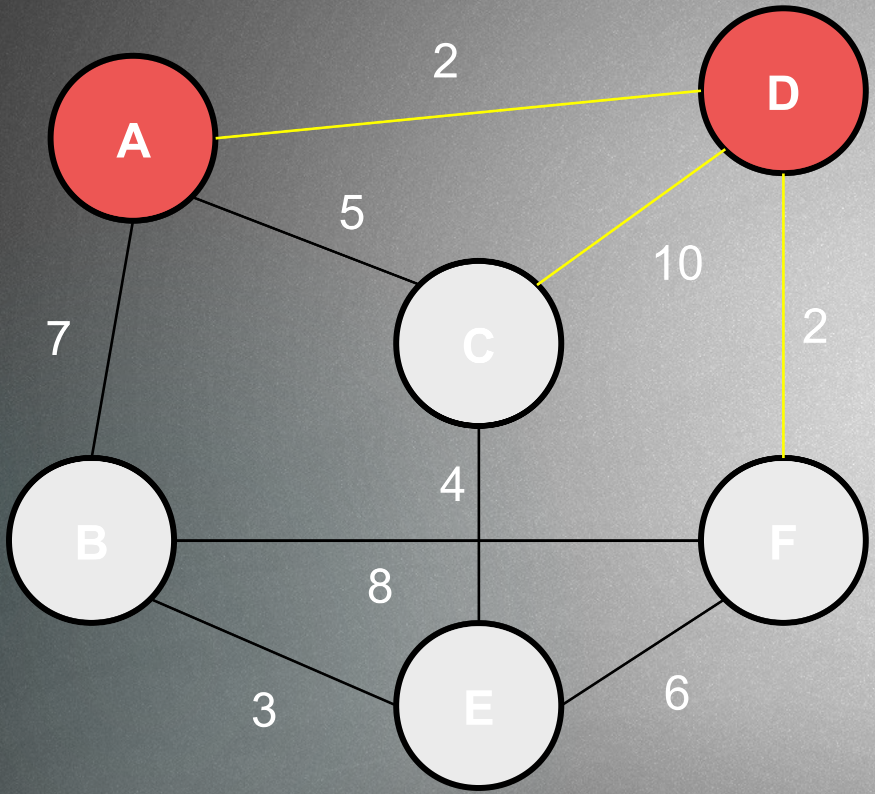
IMPORTANT: it takes cost 2 to get to D so we have to add this value from now on

From D: we can get to A (already visited) and C and F



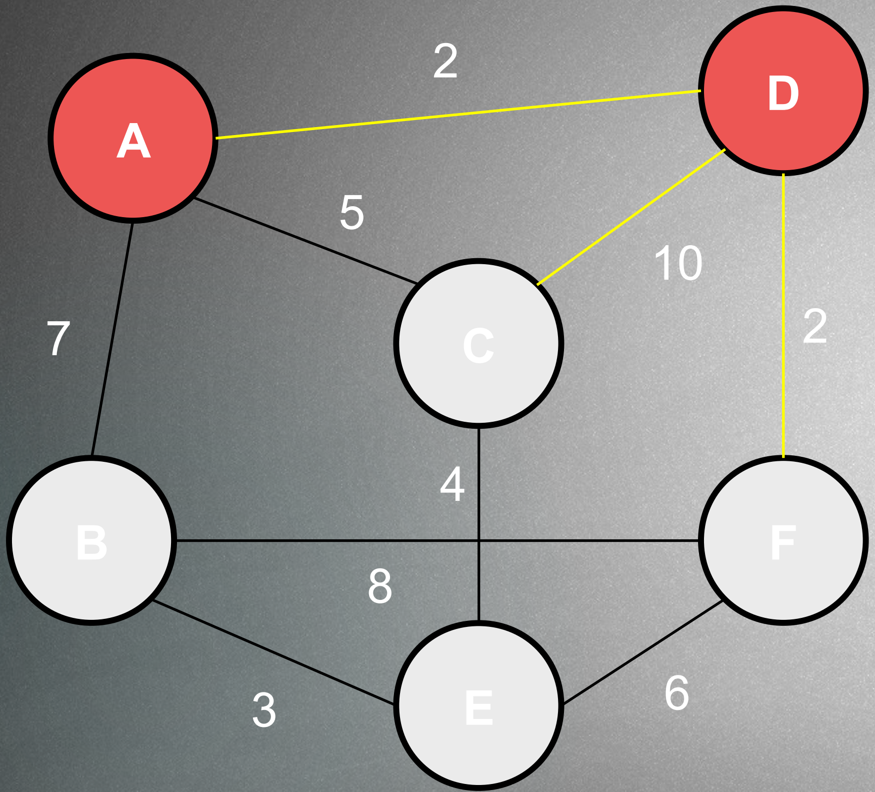
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D		5			

Math.min(10+2;5) = 5 do not change column C



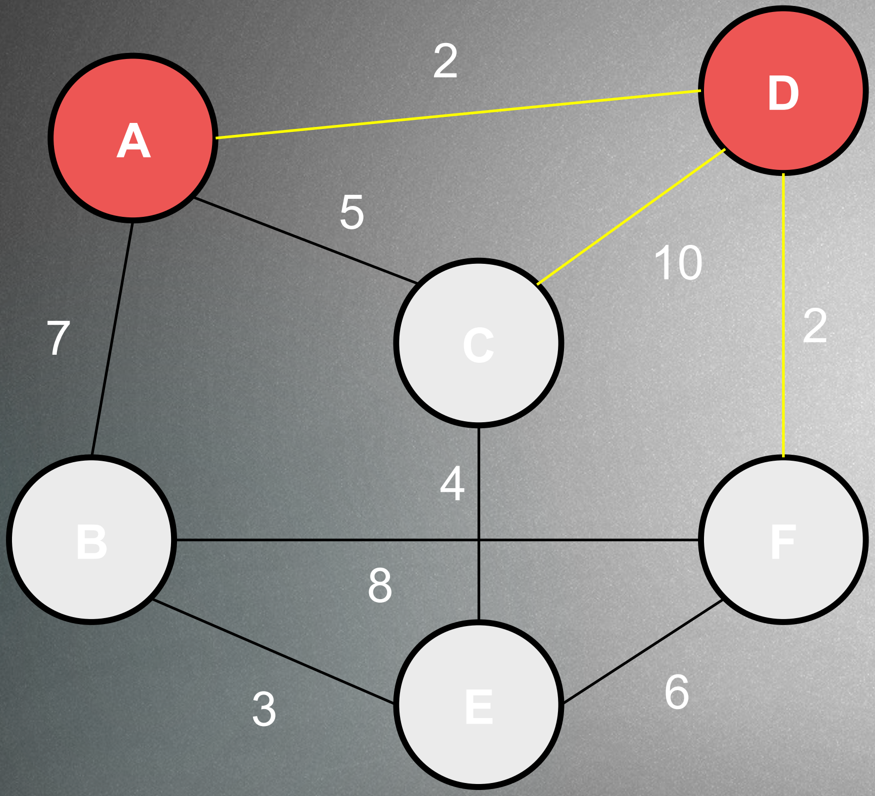
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D		5			4

Math.min(inf, 4) = 4 change column F



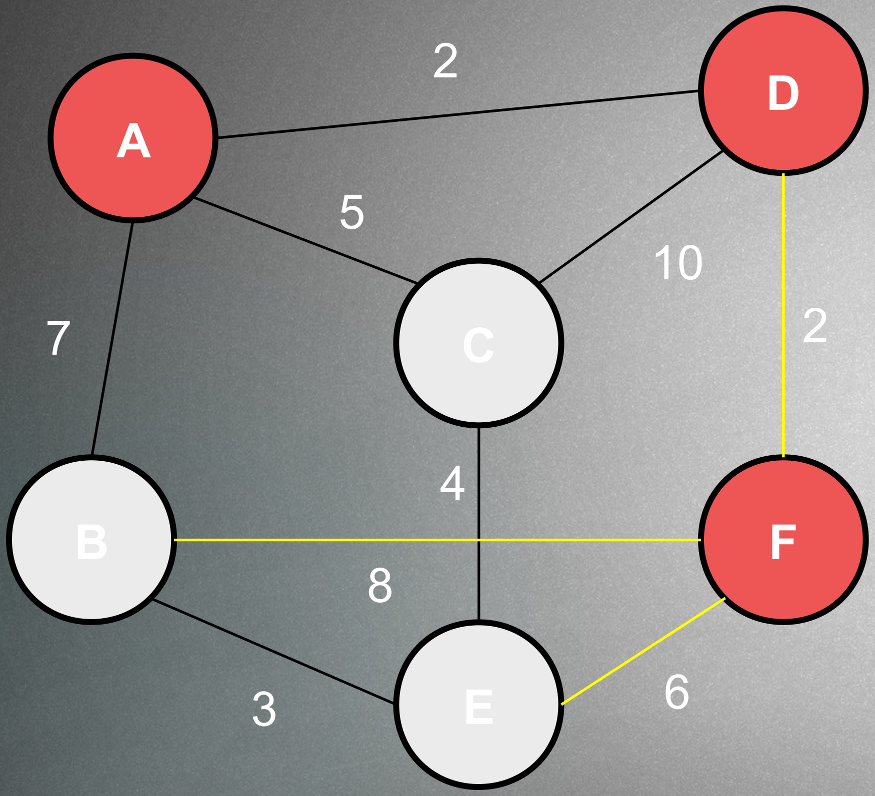
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	4

Copy all the values from the row above for nodes we have not visited yet



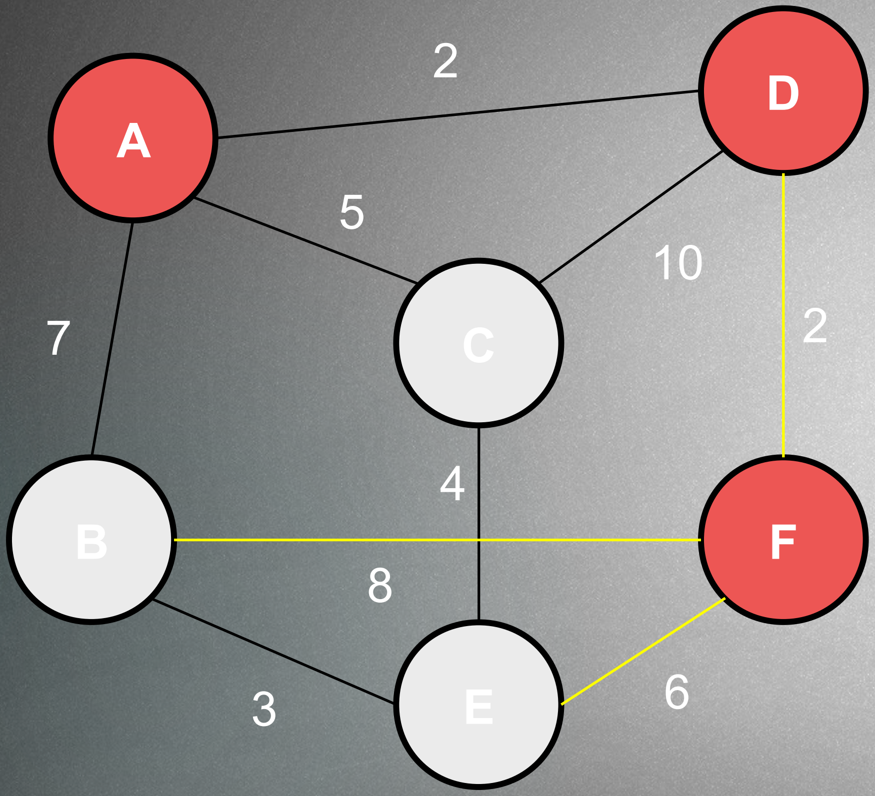
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**

Get the minimum again from the last row → so we visit node F



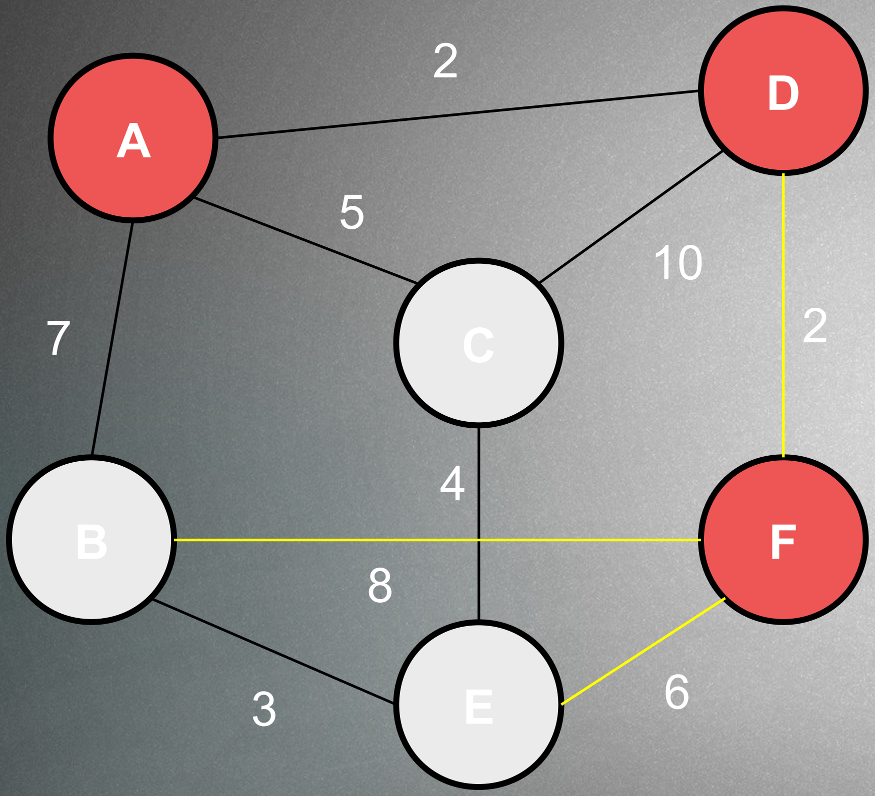
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**
F						

Get the minimum again from the last row → so we visit node F



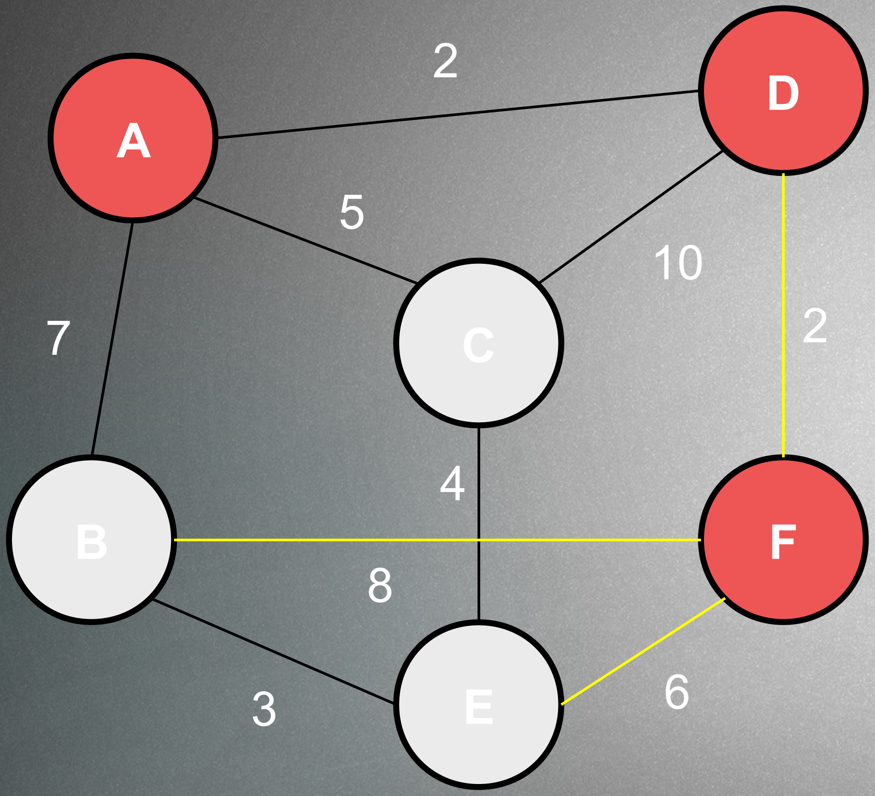
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**
F						

Node F connects to: B, E, D (we have already visited D)



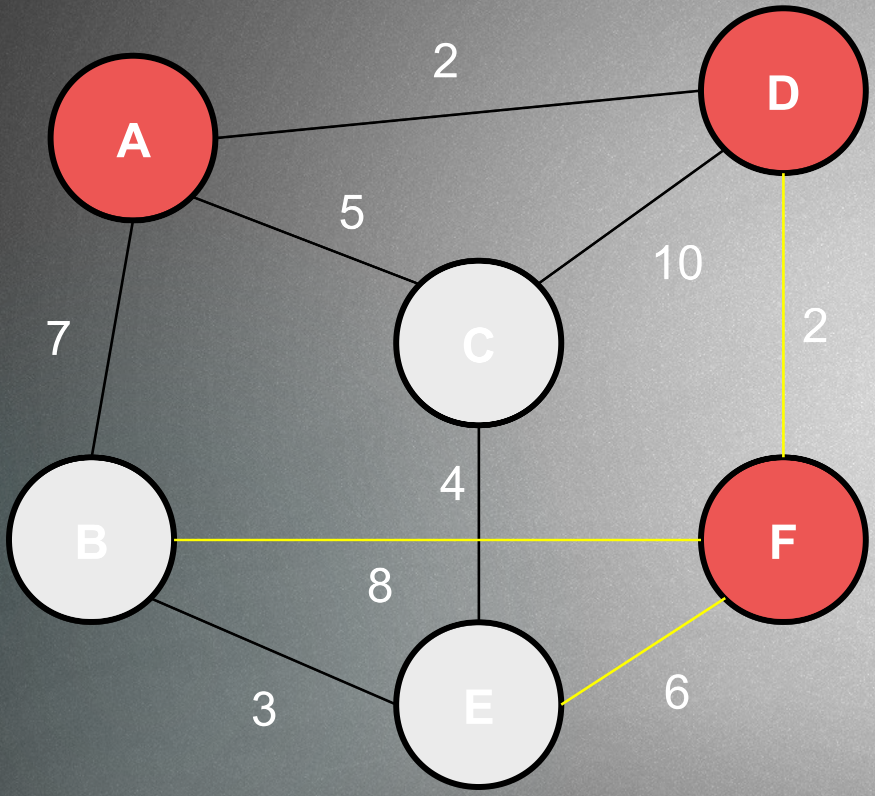
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	**4**
	F	7				

We can get to B: min(7,8+4) = 7



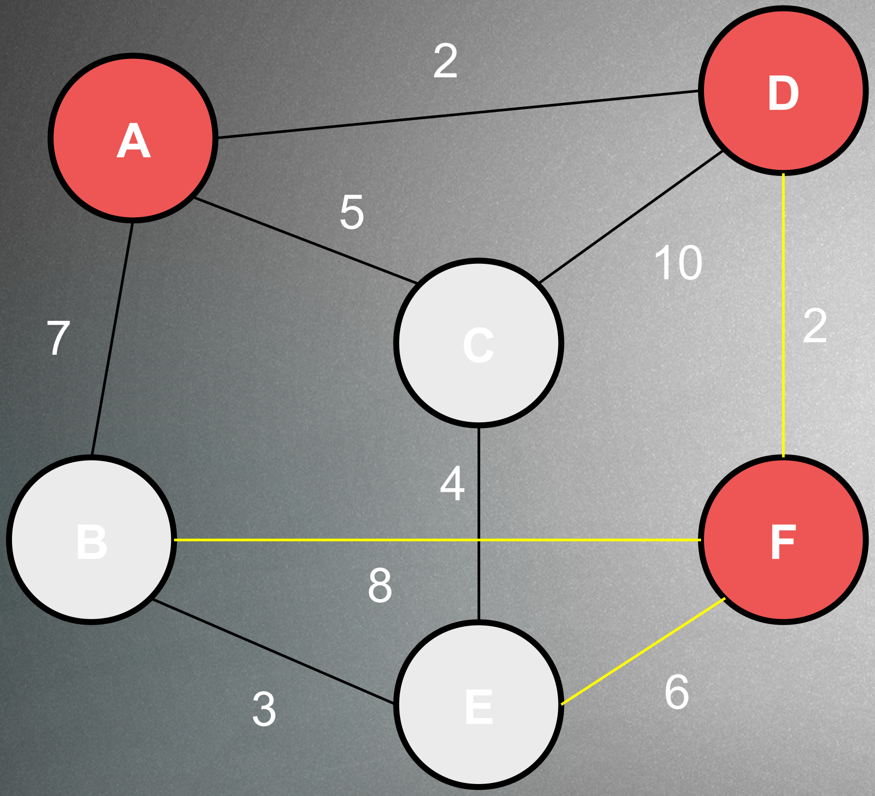
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**
F		7			10	

We can get to E: min(inf,4+6) = 10



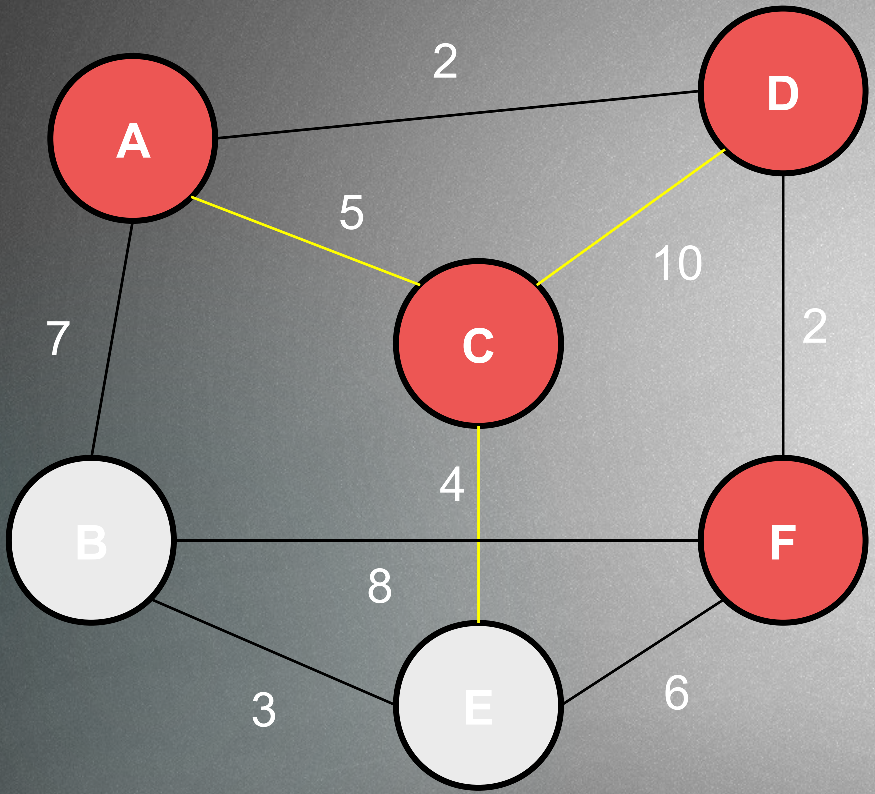
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	**4**
	F	7	5		10	

Copy all the values from the row above



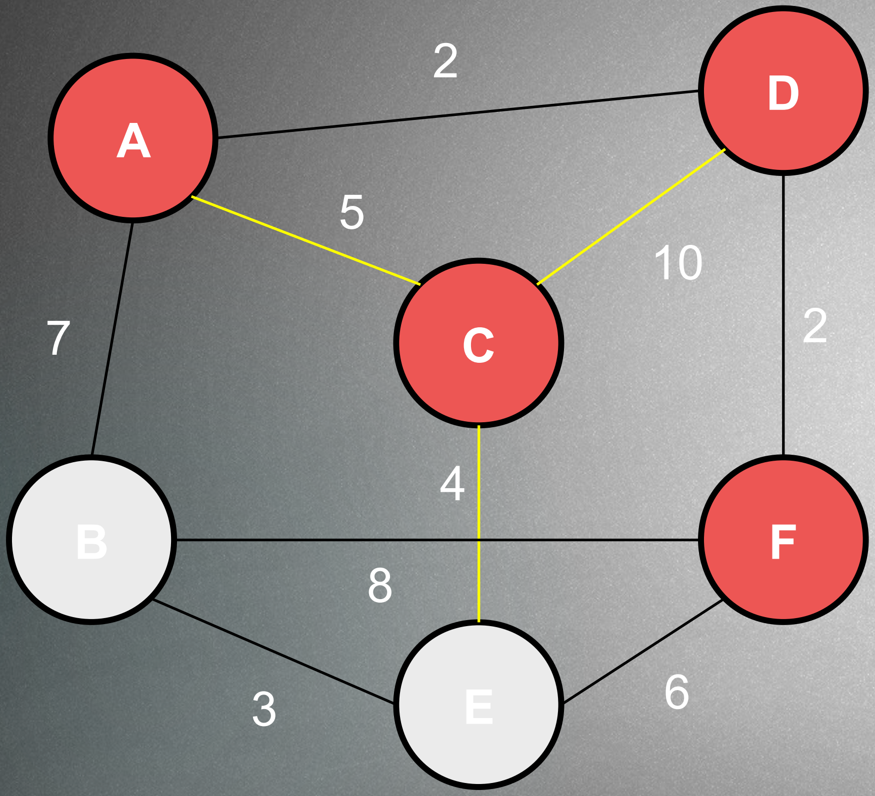
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	**4**
	F	7	5		10	

Calculate the minimum value in the last row: it is 5 so node C



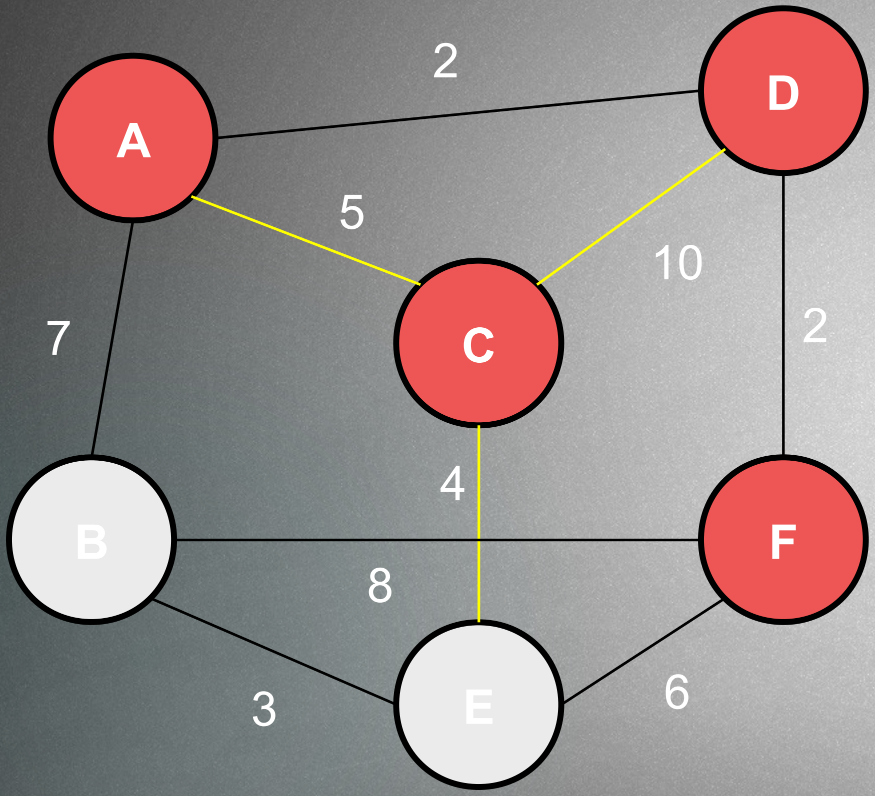
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**
F		7	**5**		10	
C						

Calculate the minimum value in the last row: it is 5 so node C



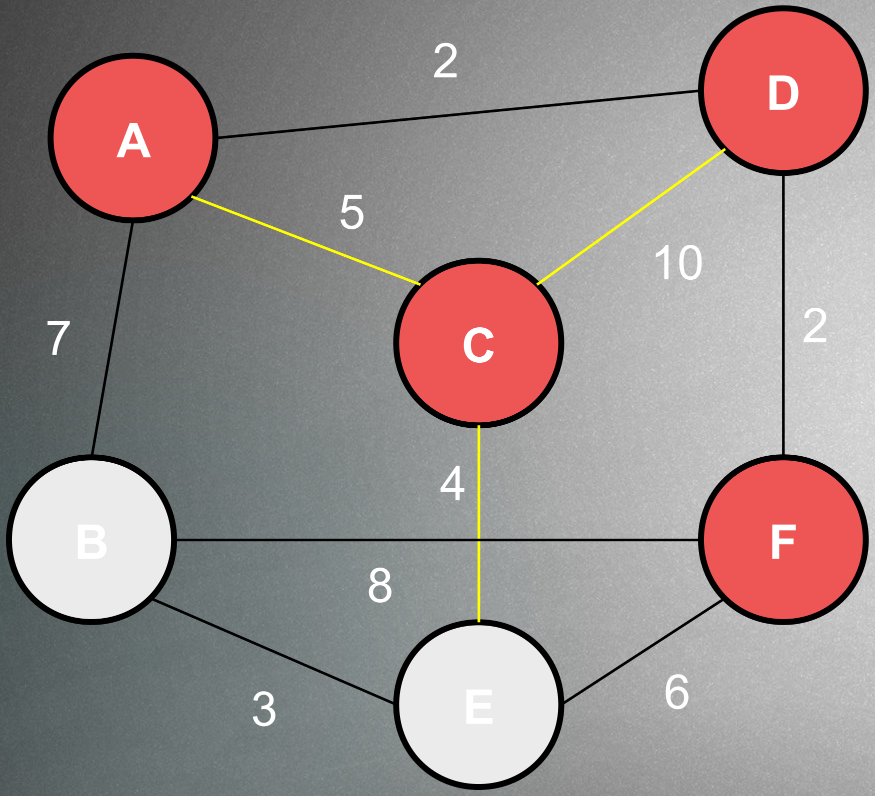
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**
F		7	**5**		10	
C						

We have already visited node A and B, so E is the only one



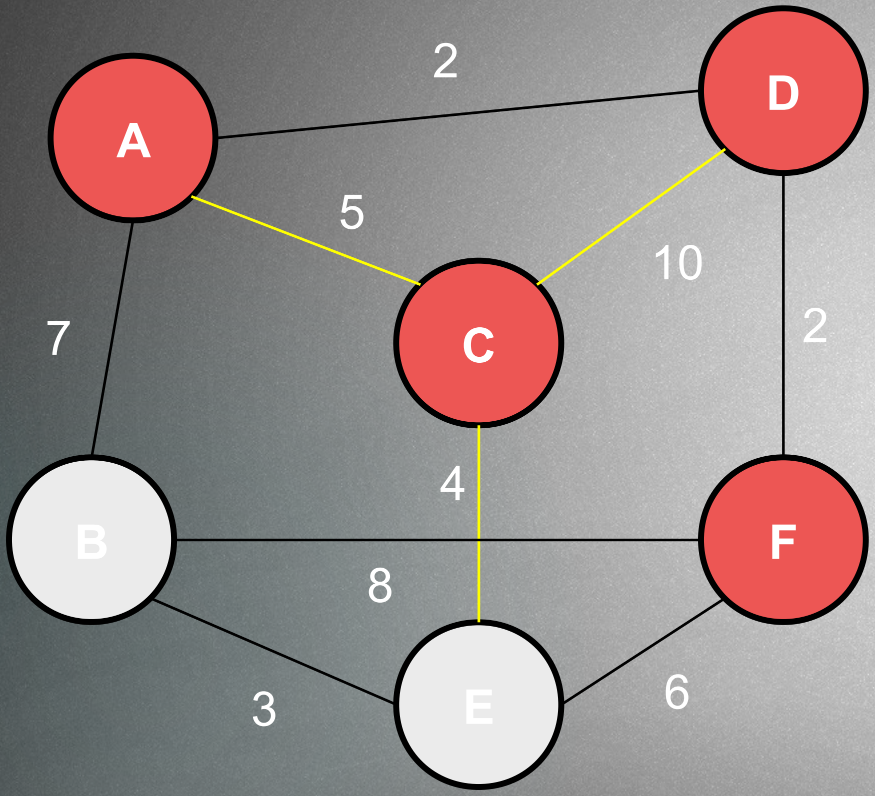
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	**4**
	F	7	**5**		10	
C					9	

min(10,5+4) = 9 we have found a shorter path



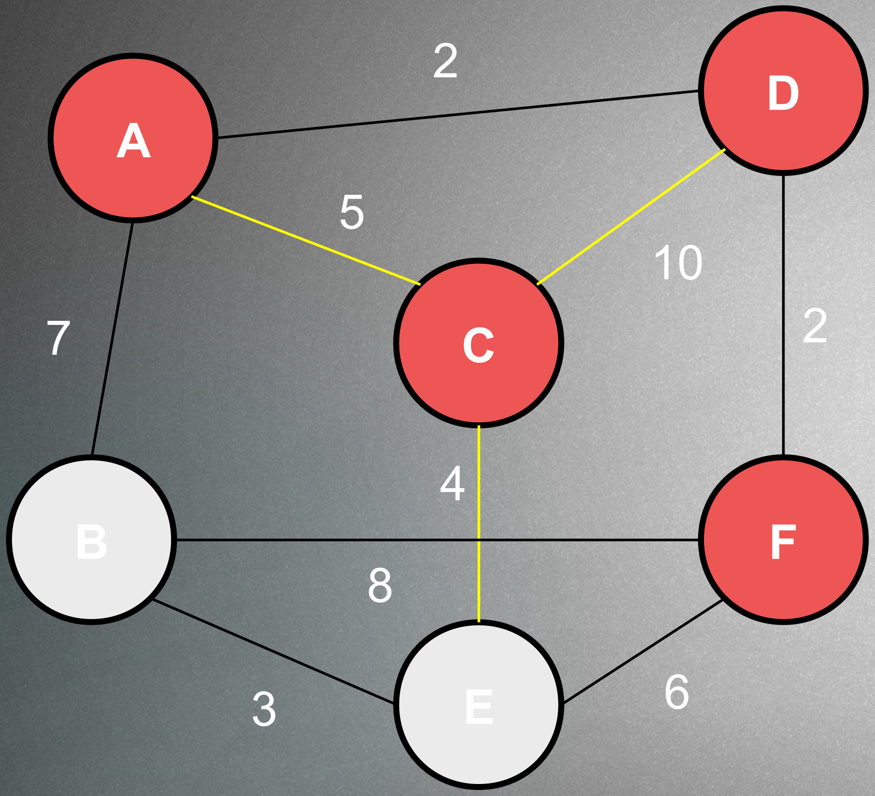
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	**4**
	F	7	**5**		10	
C					9	

Copy the values from the row above that has not been visited / ready



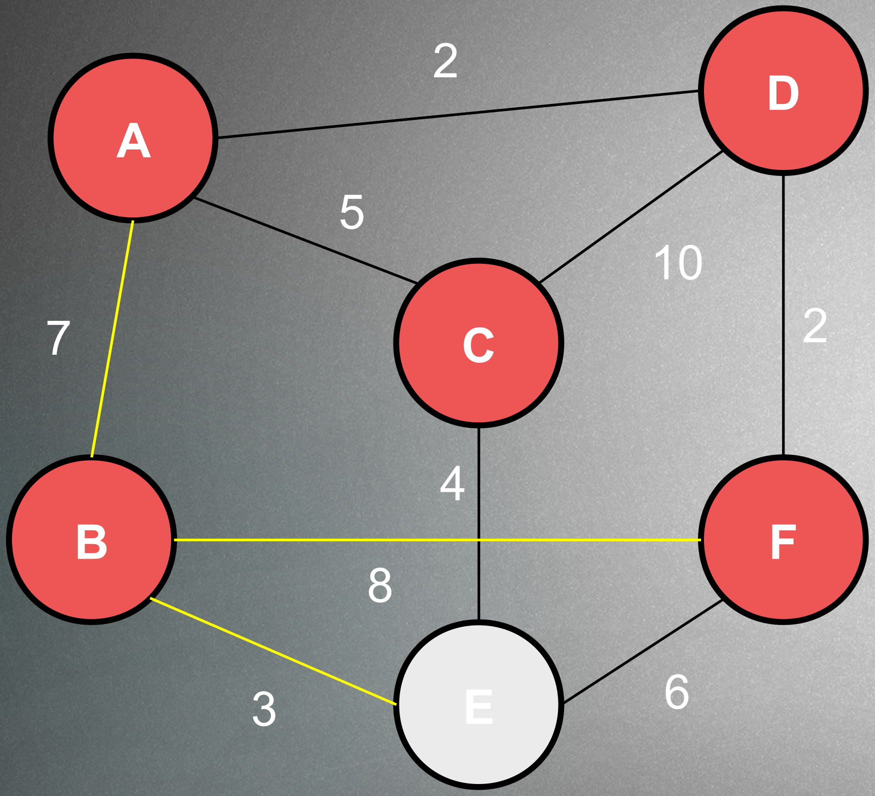
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	**4**
	F	7	**5**		10	
C		7			9	

Copy the values from the row above that has not been visited / ready



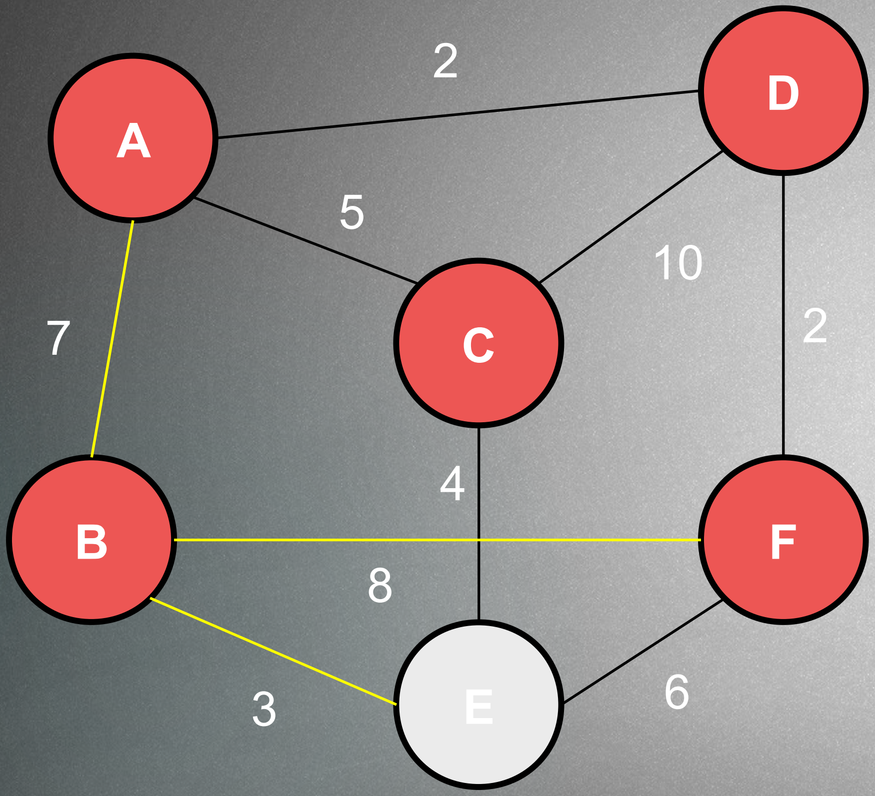
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	**4**
	F	7	**5**		10	
C		**7**			9	

Calculate the minimum: it is node B → so we consider node B



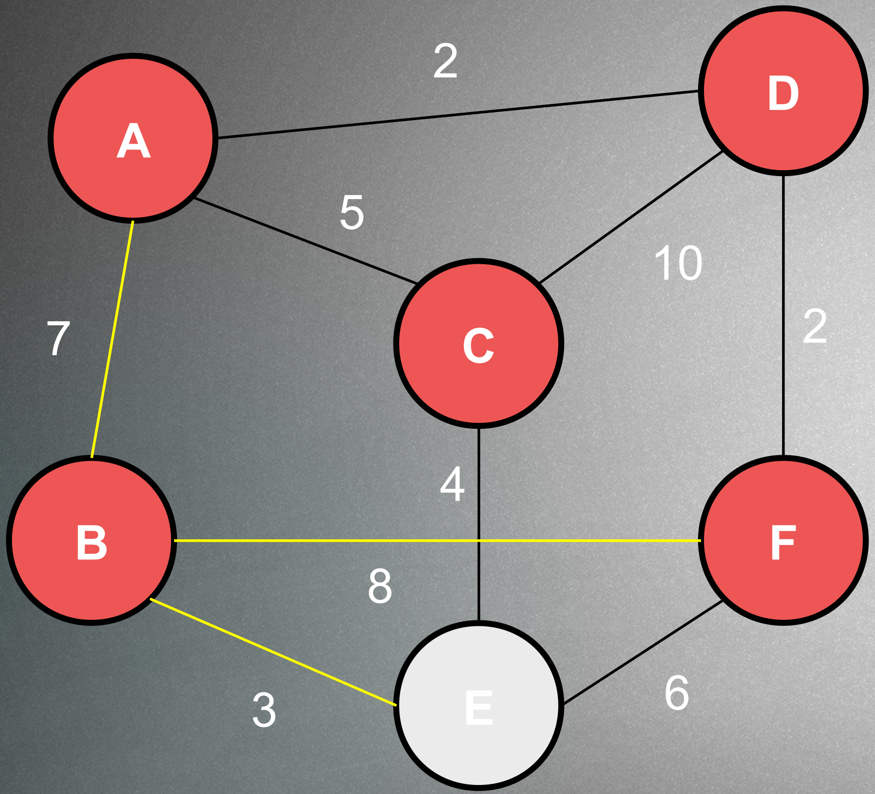
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
	D	7	5		inf	**4**
	F	7	**5**		10	
C		**7**			9	
B						

Calculate the minimum: it is node B → so we consider node B



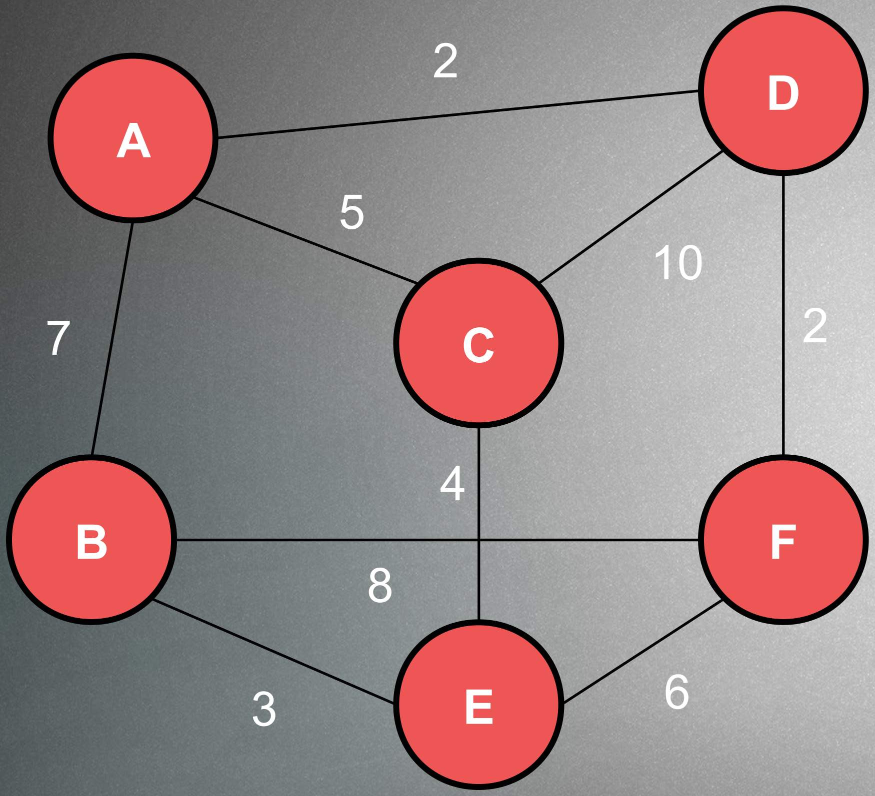
v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**
F		7	**5**		10	
C		**7**			9	
B						

We have considered every node except for the node E



v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**
F		7	**5**		10	
C		**7**			9	
B					**9**	

min(9,7+3) = 9 so no better path found



v	A	B	C	D	E	F
A	**0**	inf	inf	inf	inf	inf
A		7	5	**2**	inf	inf
D		7	5		inf	**4**
F		7	**5**		10	
C		**7**			9	
B					**9**	

Conclusion: red values represent what are the shortest path values from A to the given node
If we want the path itself: we have to „backtrack", have to store predecessors