26/12/2022
Algorithms and Data Structure
Week 7-12 portfolios
Mohamad Subhi Taha
U2180946

Practical 11

Report:

- The "**Student"** class is a simple class that stores the ID of a student and provides a "**compareTo"** method for total ordering of students. It also has a "**toString"** method for creating a human-friendly representation of the class.
- The "**Module"** class stores the ID of a module, an array of enrolled students, and the current size of the class list. It has methods for enrolling and unenrolling students, maintaining the class list in ascending order based on student ID, and creating a human-friendly representation of the class.
- The "**StudentSupportOffice"** class creates three "**Module"** objects, and five "**Student**" objects and enrols and unenrolls students to modules as described in the prompt.

Code: is in the IntelliJ project file named practical11.

Practical 12

Report:

- This code defines three classes: **Team, FootballLeague, and Dashboard.**
- The "**Team**" class is responsible for storing information about a team in a football league, including its name, number of wins, draws, losses, and number of points. It also has a method to calculate the number of points based on the number of wins, draws, and losses, and a method to create a human-readable representation of the team (i.e., a "toString" method). The Team class also implements the Comparable interface, which allows teams to be compared and sorted based on their points.
- The "**FootballLeague"** class stores a list of teams in the league and provides methods for inserting and removing teams from the list, as well as updating a team's position in the league based on its wins, draws, and losses. It also has a method to relegate the three teams with the least number of points at the end of the season. The FootballLeague class also has a method to print the league table, which is the list of teams in the league.
- The "**Dashboard**" class creates an instance of the FootballLeague class and provides methods for inserting, updating, removing, and relegating teams in the league.

Code: is in the IntelliJ project file named practical12.

Practical 13

Report:

- Class called "**Calculator**" that is responsible for evaluating a postfix expression, which is a type of mathematical expression that is evaluated by processing the operands and operators from left to right. The Calculator class uses a stack to hold the operands as it processes the expression. It also can print output that shows the steps it takes to evaluate the expression, including the method being called, the return value, and the contents of the stack.
- To evaluate the postfix expression, the Calculator class processes each character in the expression from **left to right**. If the character is an operand, it is pushed onto the **stack**. If the character is an operator, the class pops the appropriate number of operands off the stack, performs the operation, and pushes the result back onto the stack. When the end of the expression is reached, the final value on the stack is the result of the expression. The class can also handle errors such as attempting to perform an operation with insufficient operands on the stack.

Code: is in the IntelliJ project file named practical13.

Practical 14

- **Priority queue**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
A	A	A																											
B																													
		C	C	C																									
			D	D	D	D	D	D	D	D	D	D	D																
			E	E	E																								
			F	F	F	F	F	F																					
							G	G	G																				
											H	H	H	H	H	H	H												
												I	I	I															
															J	J	J	J	J										

Waiting time (Priority Queue):

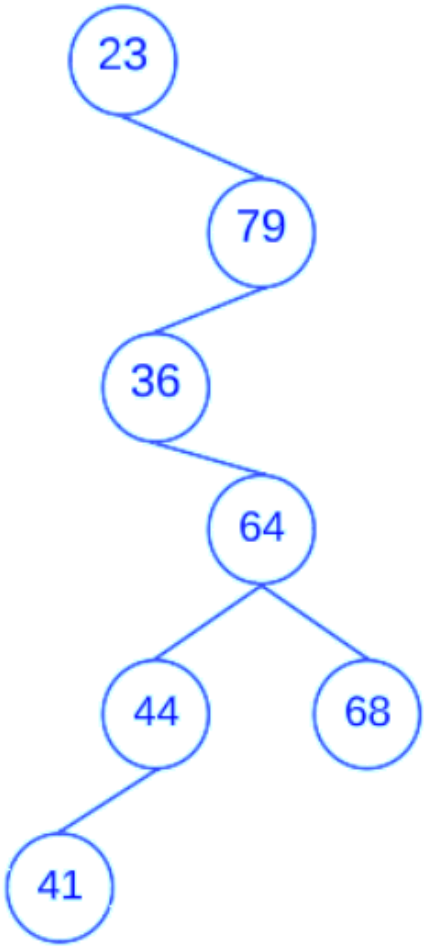
- Program A: 1-time units
- Program B: 0-time units
- Program C: 1-time units
- Program D:7-time units
- Program E: 2-time units
- Program F: 3-time units
- Program G: 2-time units
- Program H: 4-time units
- Program I: 2-time units
- Program J: 2-time units
- Total: 0+2+1+2+6+7+6+3+4+2=**24** time units
- Average: **2.4-time** units per program

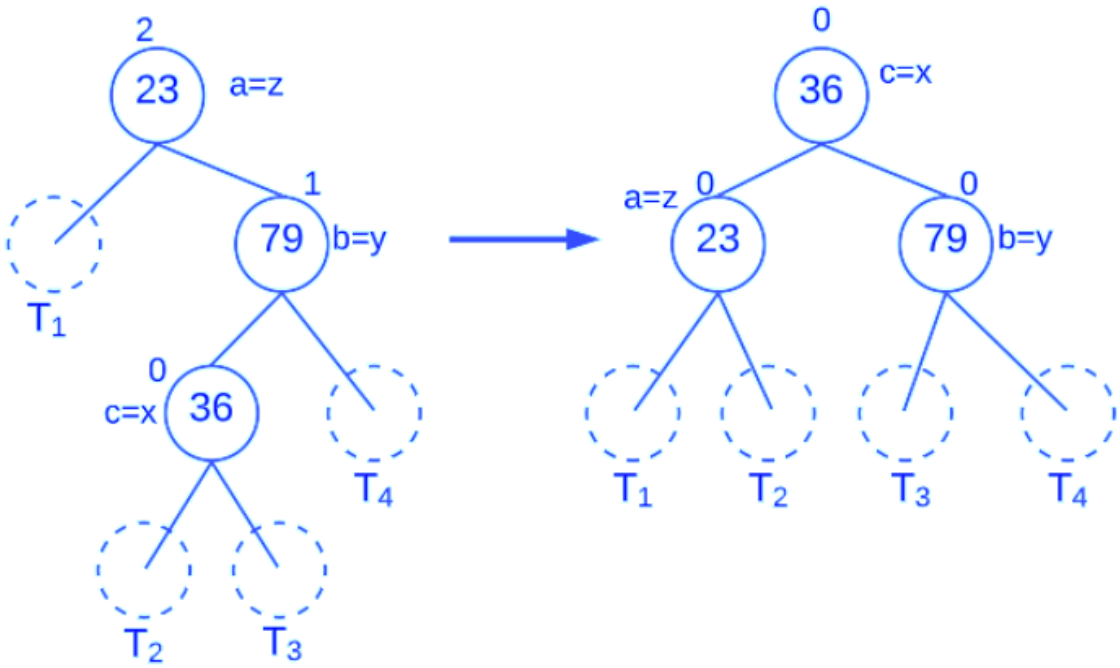
The state of the Priority queue after each event takes place is depicted below:

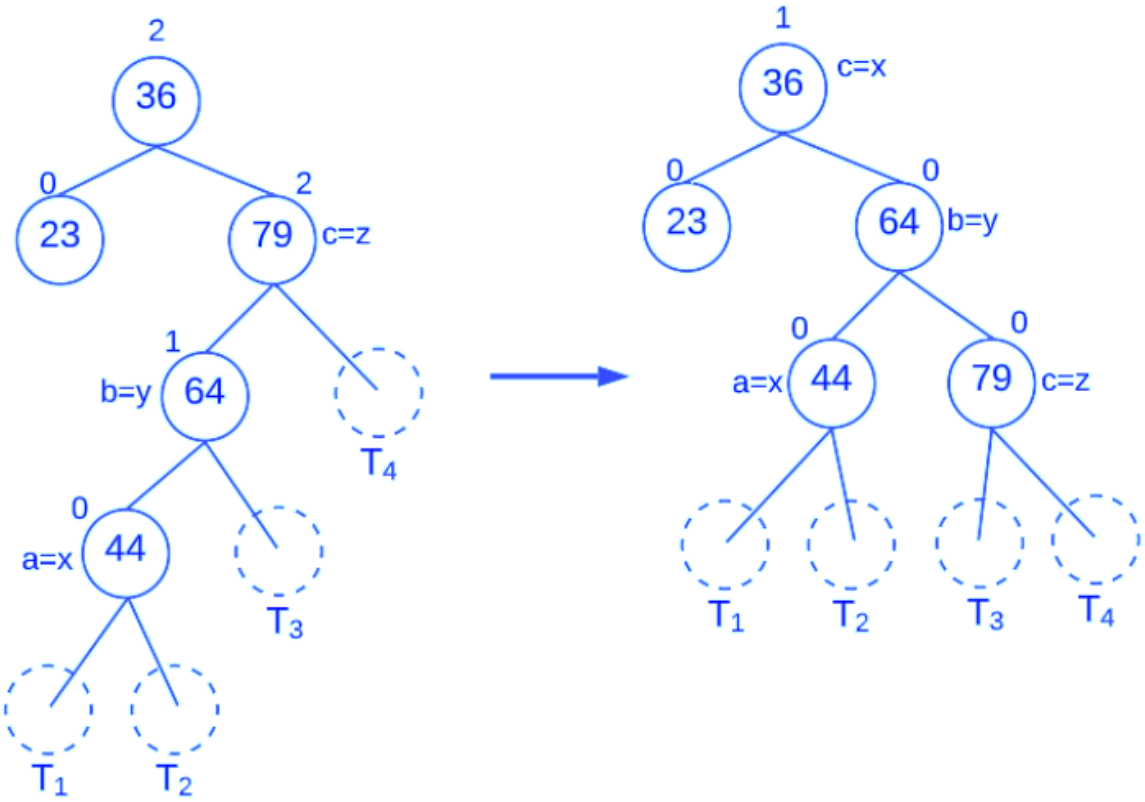
Time Unit	**Running**	**Method**	**Return Value**	**first ← Q ← last**
0	B	insert (2, A) insert (1, B) remove Min ()	--- (1, B)	{(2, A) (1, B)}
1	A	removeMin()	{(2, A)}	
2	A	insert (2, C)		(2, C)
3	E	Insert (4, D) Insert (1, E) Insert (3, F) removeMin ()	----- ------ ------ {(1, E)}	{(2, C), (4, D), (1, E), (3, F)} {(2, C), (4, D), (3, F)}
4	C	removeMin ()	{2, C)}	{(4, D), (3, F)}
5	C	--	-	{(4, D), (3, F)}
6	F	removeMin ()	(3, F)	{(4, D)}
7	F	Insert(G,1)	-----	{(4, D), (1,G)}
8	F	-------	-------	{(4, D), (1, G)}
9	G	removeMin ()	(1, G)	{(4, D)}
10	D	removeMin()	(4, D)	

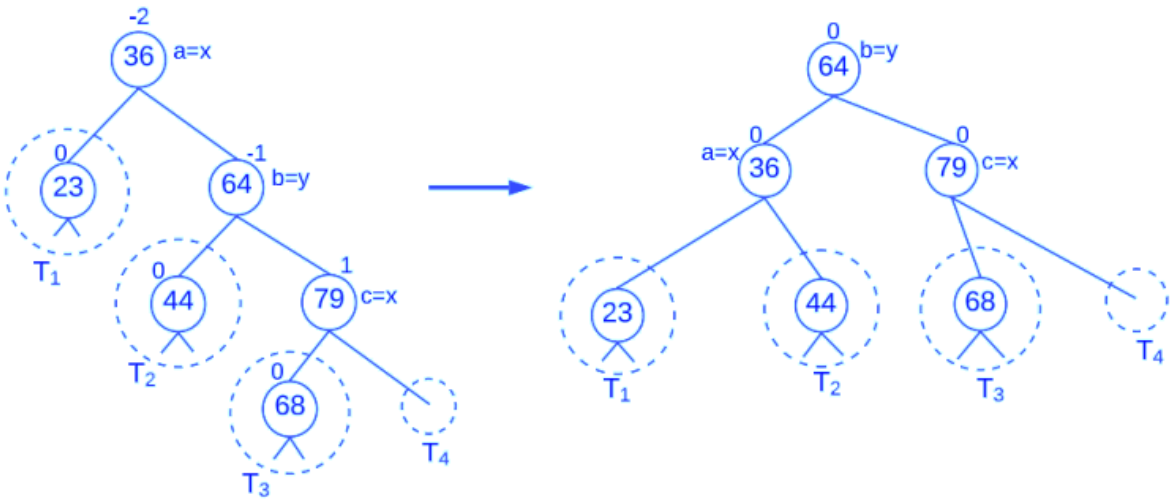
11	D	Insert (2, H)	------	{(2, H)}
12	D	Insert (1, I)	-----	{(2, H), (1, I)}
13	D	-	-	{(2, H), (1, I)}
14	I	removeMin ()	(1, I)	{(2, H)}
15	H	Insert (3, J) removeMin ()	(2, H)	{(2, H), (3, J)} {(3, J)}
16	H	-	--	{(3, J)}
17	J	removeMin ()	(3, J)	{ }
18	J	--		{ }
19	J	--		{ }

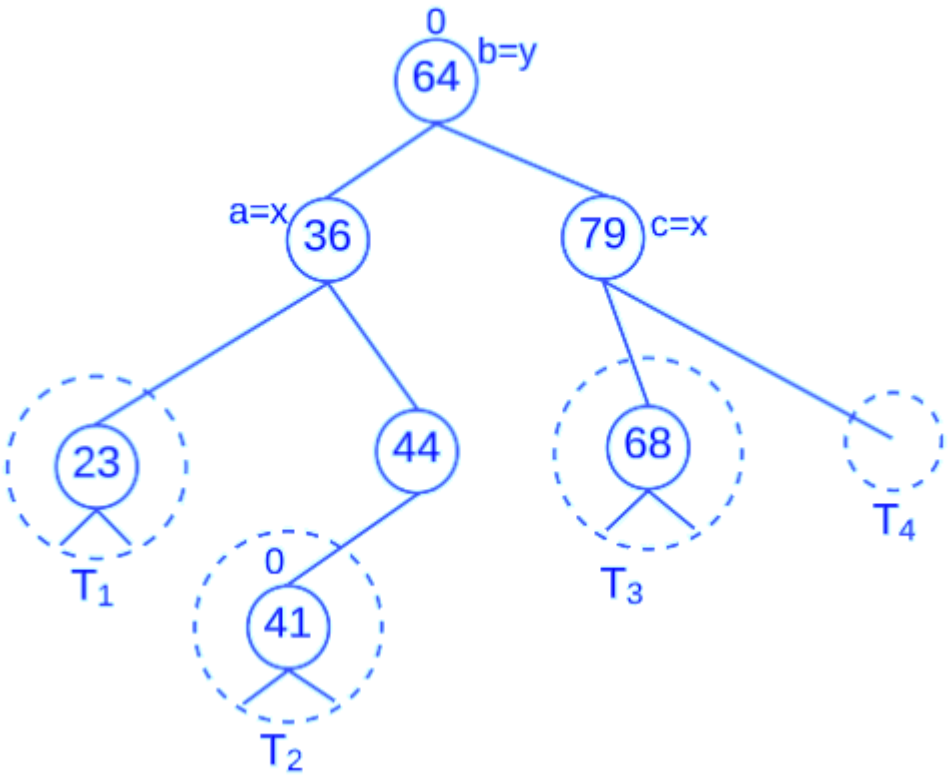
Practical 15




2
23
a=z
1
79
b=y
T1
0
c=x
36
T4
T2
T3
0
36
c=x
a=z
0
0
23
79
b=y
T1
T2
T3
T4


2
36
0
23
2
79
c=z
1
b=y
64
T4
0
a=x
44
T3
T1
T2
1
36
c=x
0
23
0
64
b=y
0
a=x
44
0
79
c=z
T1
T2
T3
T4


-2
36
a=x
0
23
-1
64
b=y
T1
0
44
1
79
c=x
T2
0
68
T3
T4
0
64
b=y
0
a=x
36
0
79
c=x
23
44
68
T1
T2
T3
T4



Practical 16

Array after 35 is inserted:

35						
0	1	2	3	4	5	6

Array after 12 is inserted:

12	35					
0	1	2	3	4	5	6

Array after 73 is inserted:

12	35	73				
0	1	2	3	4	5	6

Array after 61 is inserted:

12	35	73	61			
0	1	2	3	4	5	6

Array after 23 is inserted:

12	23	73	61	35		
0	1	2	3	4	5	6

Array after 57 is inserted:

12	23	57	61	35	73	
0	1	2	3	4	5	6

Array after 29 is inserted:

12	23	29	61	35	73	57
0	1	2	3	4	5	6

Max-heap

Array after 35 is inserted:

35						
0	1	2	3	4	5	6

Array after 12 is inserted:

35	12					
0	1	2	3	4	5	6

Array after 73 is inserted:

73	12	35				
0	1	2	3	4	5	6

Array after 61 is inserted:

73	61	35	12			
0	1	2	3	4	5	6

Array after 23 is inserted:

73	61	35	12	23		
0	1	2	3	4	5	6

Array after 57 is inserted:

73	61	57	12	23	35	
0	1	2	3	4	5	6

Array after 29 is inserted:

73	61	57	12	23	35	29
0	1	2	3	4	5	6

Initial array after **35**, **12**, **73**, **61**, **23**, **57**, and **29** are inserted:

73	61	57	29	23	35	12
0	1	2	3	4	5	6

Array after 73 is sorted:

61	29	57	23	35	12	73
0	1	2	3	4	5	6

Array after 61 is sorted:

57	29	35	23	12	73	61
0	1	2	3	4	5	6

Array after 57 is sorted:

35	29	12	23	73	61	57
0	1	2	3	4	5	6

Array after 35 is sorted:

29	23	12	73	61	57	35
0	1	2	3	4	5	6

Array after 29 is sorted:

23	12	73	61	57	35	29
0	1	2	3	4	5	6

Array after 23 is sorted:

12	73	61	57	35	29	23
0	1	2	3	4	5	6

Array after 12 is sorted:

73	61	57	35	29	23	12
0	1	2	3	4	5	6

Practical 17

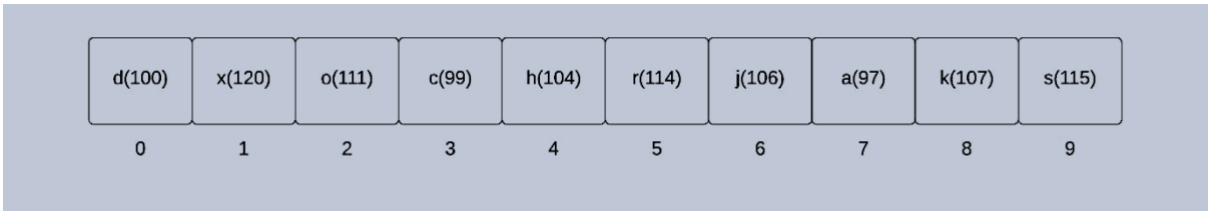
Report:

- three classes: **PhoneBookEntry, PhoneBook, and User.**
  The PhoneBookEntry class is responsible for storing information about a single phone book entry, including the full name (surname and name), phone number, email, and address. It also has a method to create a human-readable representation of the entry (i.e., a "toString" method).
- The "**PhoneBook"** class stores phone book entries in a map, where the key is the full name of the entry and the value is the PhoneBookEntry object. The PhoneBook class provides methods for inserting, updating, and deleting entries in the phone book, as well as for printing the details of a specific entry or all entries in the phone book.
- The "**User"** class creates an instance of the PhoneBook class and provides methods for inserting, updating, removing, and printing phone book entries.

Code: is in the IntelliJ project file named practical17.

Practical 18

A hashtable with keys 'd', 'h', 'x', 'r', 'a', 'j', 'o', 'k', 's' and 'c' in that order based on linear probing is as follows:

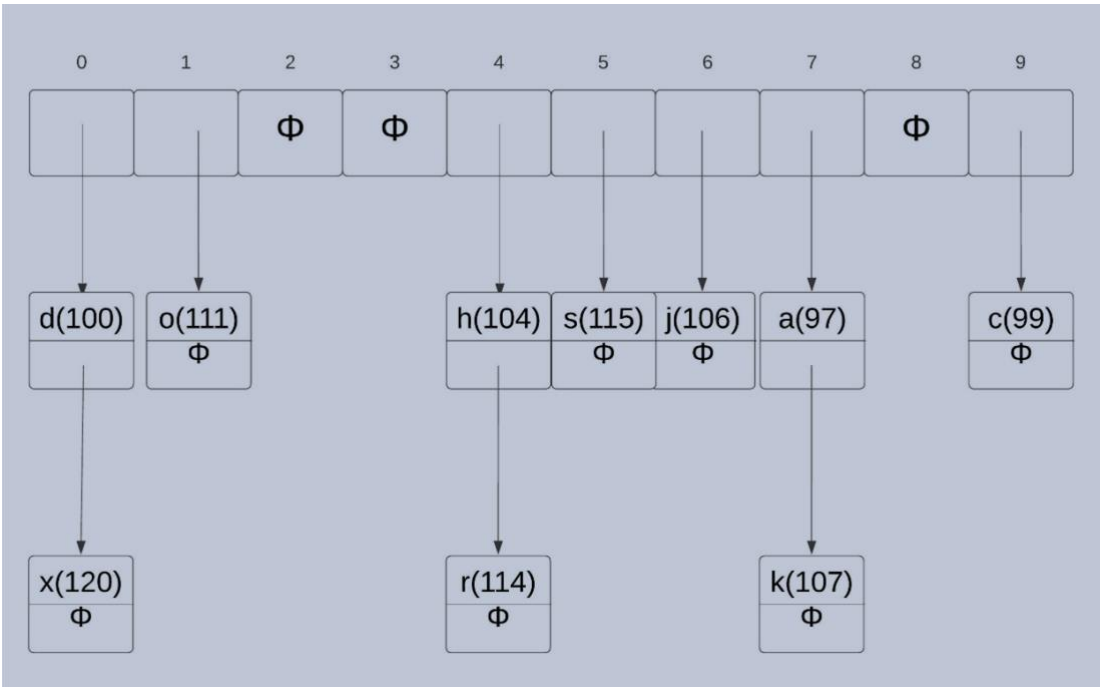


The calculations for linear probing are:

- 'd': hash('d') = ascii('d') = 100, trimming gives 100 % 10 = 0, so 'd' goes into index 0 in the array
- 'h': hash('h') = ascii('h') = 104, trimming gives 104 % 10 = 4, so 'h' goes into index 4 in the array
- 'x': hash('x') = ascii('x') = 120, trimming gives 120% 10 = 0 (collision). Linear probing: 0→ 1, so 'x' goes into index 1 in the array
- 'r': hash('r) = ascii('r') = 114, trimming gives 114 % 10 = 4, (collision). Linear probing: 4 → 5, so 'r' goes into index 5 in the array

- ‘a’: hash(‘a’) = ascii(‘a’) = 97, trimming gives 97 % 10 = 7 , so ‘a’ goes into index 7 in the array
- ‘j’: hash(‘j’) = ascii(‘j’) = 106, trimming gives 106% 10 = 6 so ‘j’ goes into index 6 in the array
- ‘o’: hash(‘o’) = ascii(‘o’) = 111, trimming gives 111 % 10 = 1 (collision). Linear probing: 1 → 2, so ‘o’ goes into index 2 in the array
- ‘k’: hash(‘k’) = ascii(‘k’) = 107, trimming gives 107 % 10 = 7 (collision). Linear probing: 7 → 8, so ‘k’ goes into index 8 in the array
- ‘s’: hash(‘s’) = ascii(‘s’) = 115, trimming gives 115 % 10 = 5 (collision). Linear probing: 5 → 6 → 7 → 8 → 9, so ‘s’ goes into index 9 in the array
- ‘c’: hash(‘c’) = ascii(‘c’) = 99, trimming gives 99% 10 = 9 (collision). Linear probing: 9 → 0 → 1 → 2 → 3, so ‘c’ goes into index 3 in the array

A hashtable with keys ‘d’, ‘h’, ‘x’, ‘r’, ‘a’, ‘j’, ‘o’, ‘k’, ‘s’ and ‘c’, in that order based on chaining is as follows:



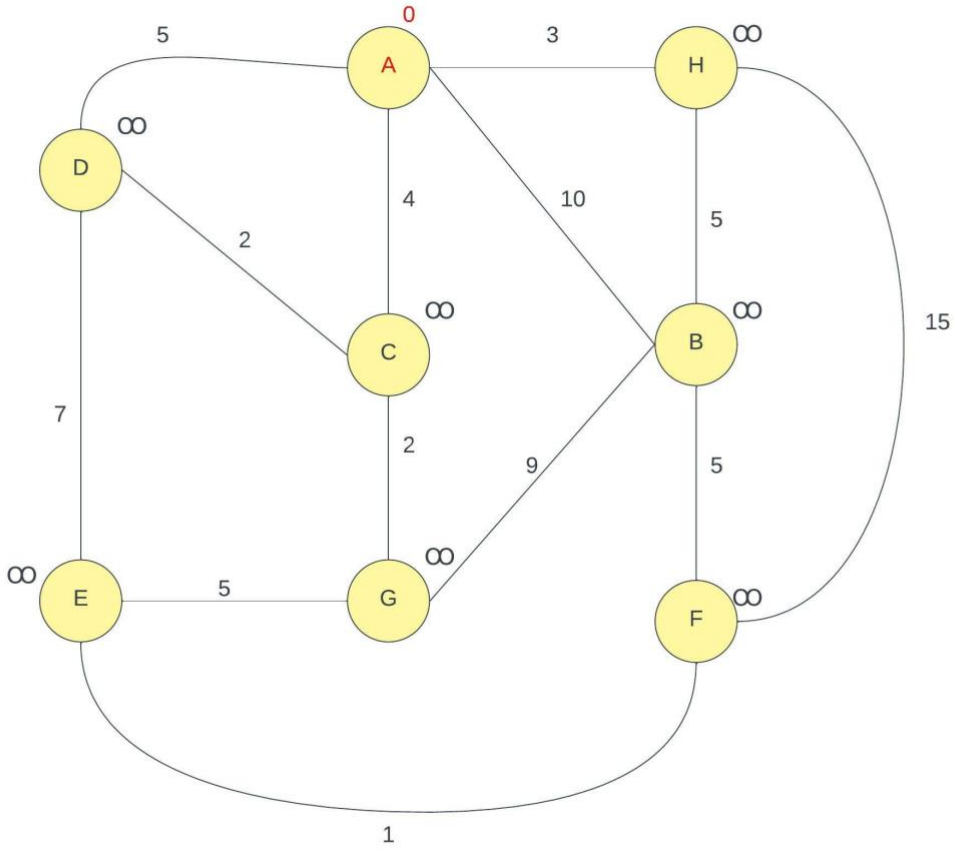
The calculations for chaining are:

- ‘d’: hash(‘d’) = ascii(‘d’) = 100, trimming gives 100 % 10 = 0, so a new linked list is created at index 0
- ‘h’: hash(‘h’) = ascii(‘h’) = 104, trimming gives 104 % 10 = 4, so a new linked list is created at index 4
- ‘x’: hash(‘x’) = ascii(‘x’) = 120, trimming gives 120% 10 = 0 (collision), add ‘x’ to the linked list at index 0
- ‘r’: hash(‘r) = ascii(‘r’) = 114, trimming gives 114 % 10 = 4, (collision),), add ‘r’ to the linked list at index 4

- ‘a’: hash(‘a’) = ascii(‘a’) = 97, trimming gives 97 % 10 = 7 so a new linked list is created at index 7

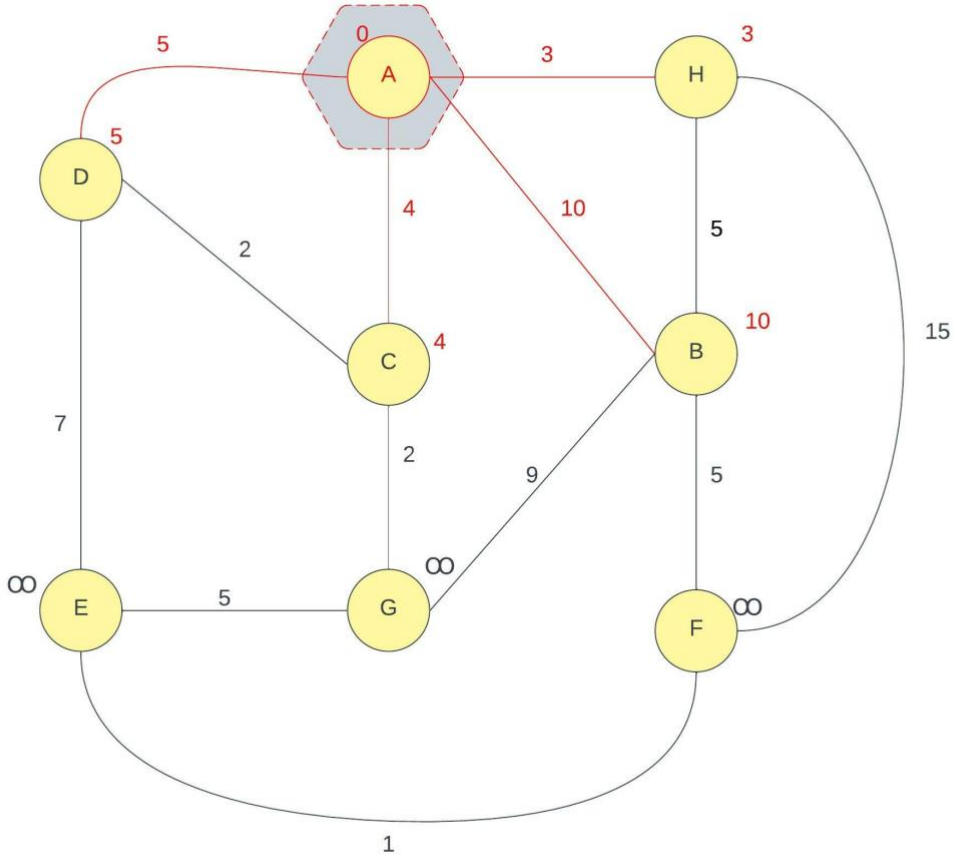
Practical 19

Appling Dijkstra’s algorithm on the following graph, **starting from node ‘A’**:



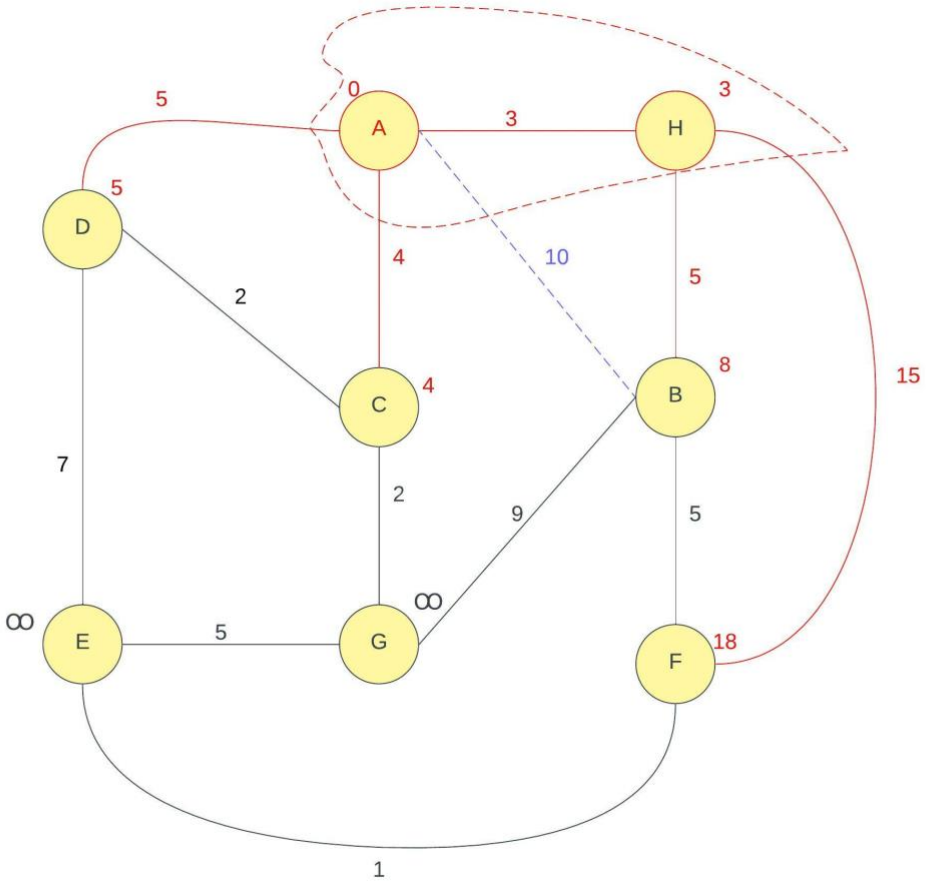
Add node A to “cloud”, perform relaxation of neighbouring nodes outside the “cloud” (i.e., nodes D, H, B and C)

- D[D] = 0+5 = 5 < ∞
- D[H] = 0+3 = 3 < ∞
- D[B] = 0+10 = 10 < ∞
- D[C] = 0+4 = 4 < ∞



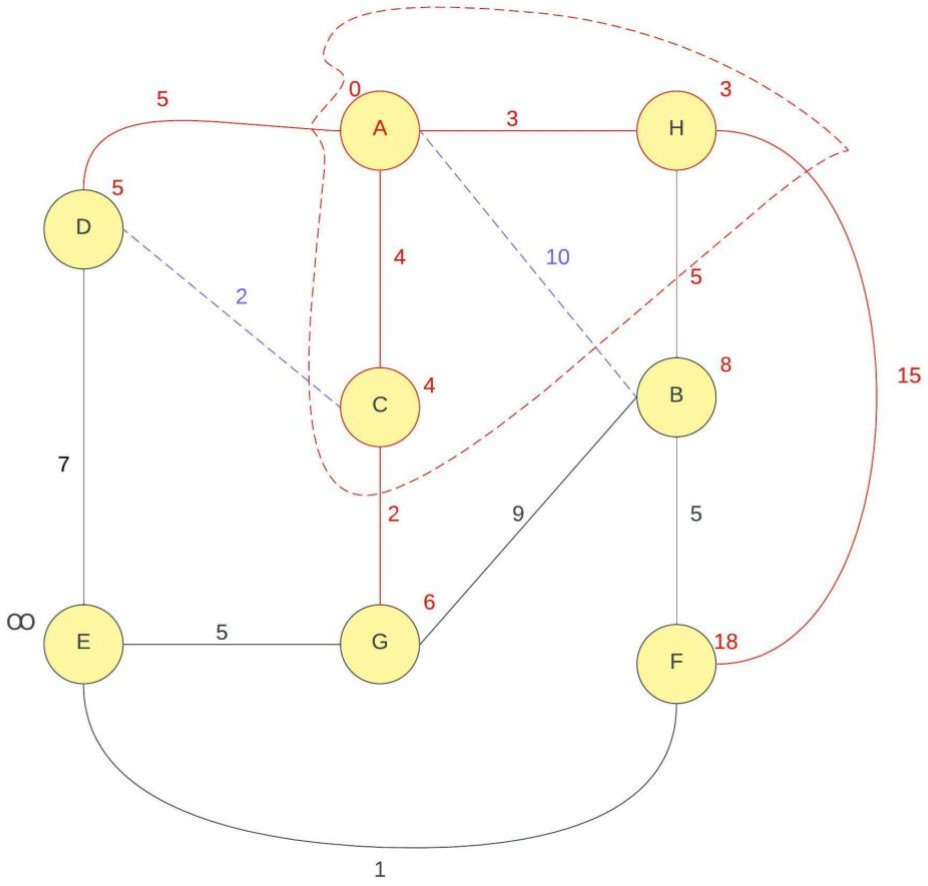
Add node H to “cloud”, perform relaxation of neighbouring nodes outside the “cloud” (i.e., nodes B and F)

- D[B]= 3+5= 8<10
- D[F]= 3+15=18< ∞



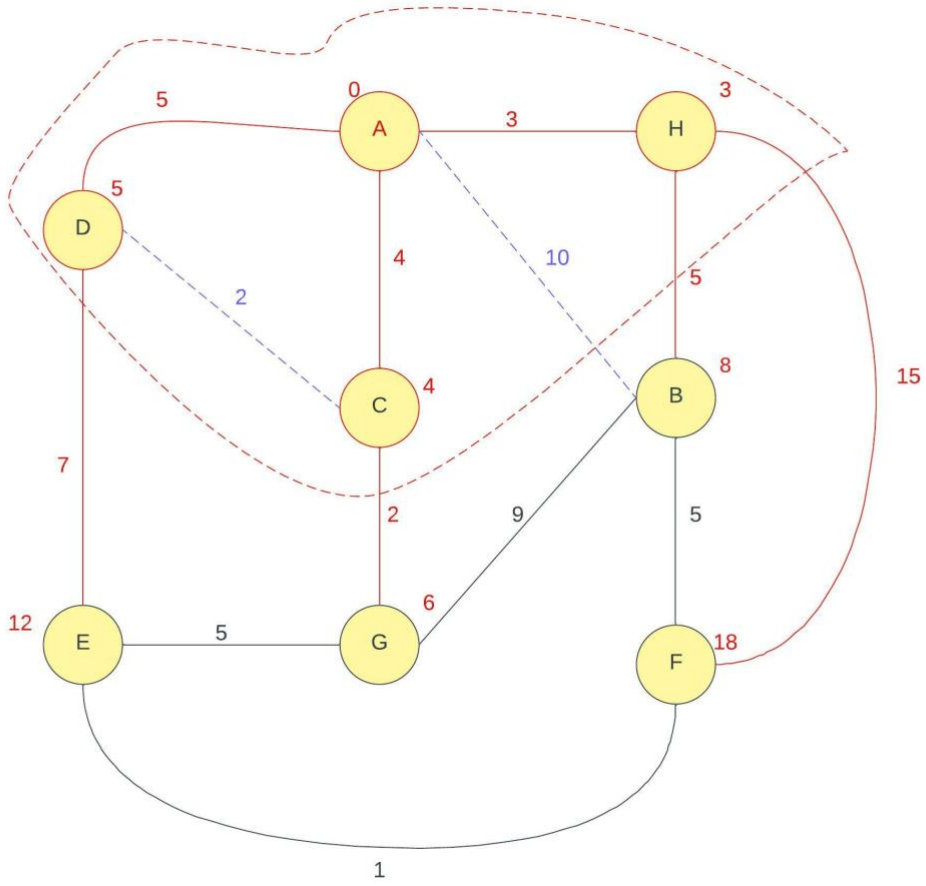
Add node C to “cloud”, perform relaxation of neighbouring nodes outside the “cloud” (i.e., nodes G and D)

- D[D] = 4+2 = 6 > 5
- D[G] = 4+2 = 6 < ∞



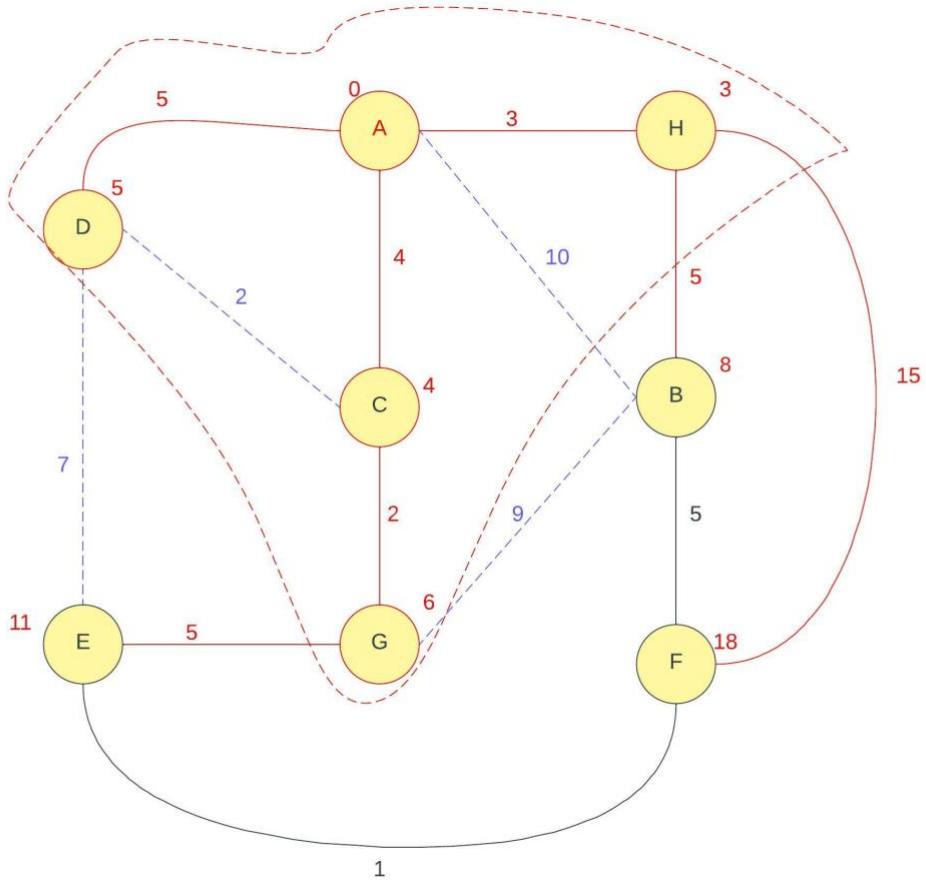
Add node D to “cloud”, perform relaxation of neighbouring nodes outside the “cloud” (i.e., nodes E)

- D[E] = 5+7 = 12



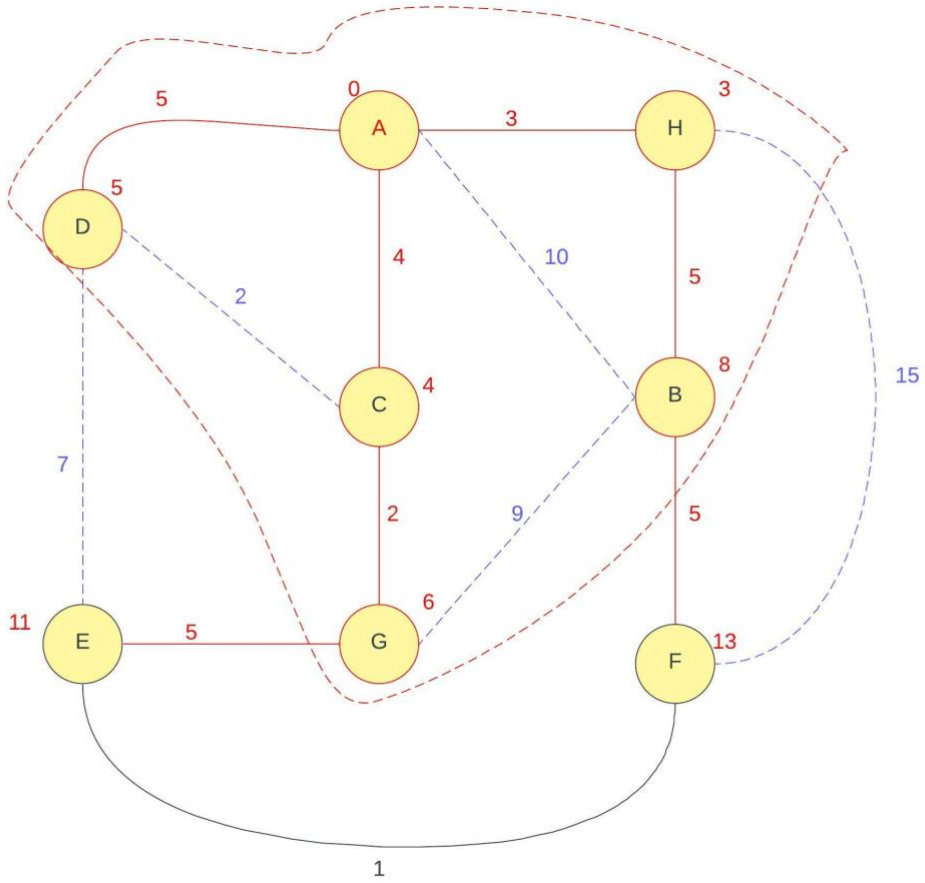
Add node G to “cloud”, perform relaxation of neighbouring nodes outside the “cloud” (i.e., nodes B and E)

- D[B] = 6+9 =15 > 8
- D[E] = 6+5 = 11 < 12



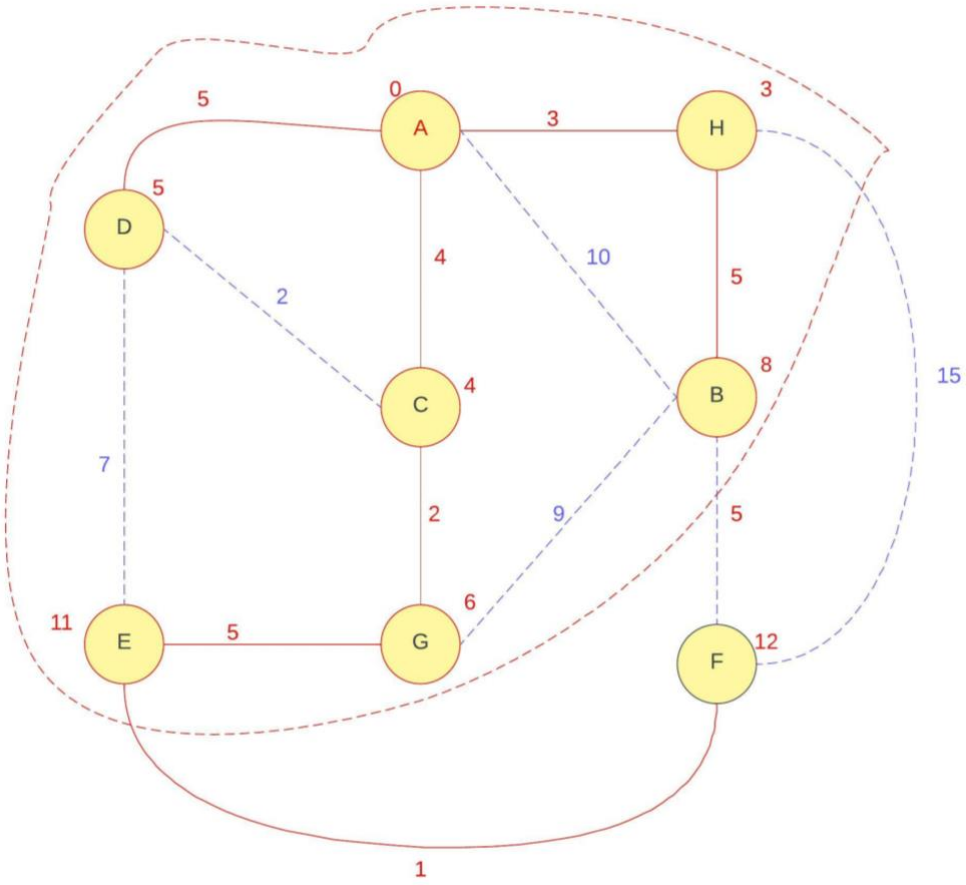
Add node B “cloud”, perform relaxation of neighbouring nodes outside the “cloud” (i.e., node B)

- D[F] = 8+5 = 13 < 18

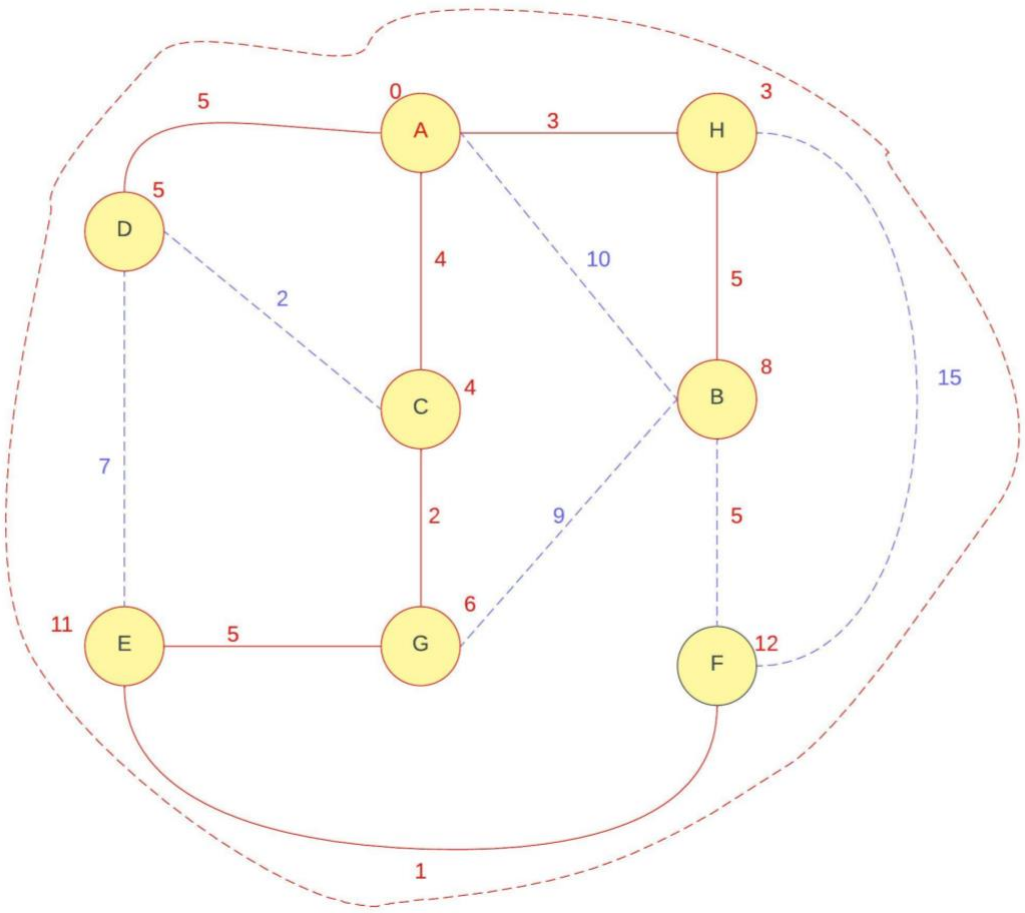


Add node E “cloud”, perform relaxation of neighbouring nodes outside the “cloud” (i.e., node F)

- D[F] = 11+1 = 12 < 13



Add node F to “cloud”, perform relaxation of neighbouring nodes outside the “cloud” (no neighbouring nodes outside the “cloud”)



Practical 20

Report:

- A class called ”**SierpinskiCarpet**” is responsible for generating a Sierpinski carpet, which is a fractal pattern formed by recursively removing smaller squares from a larger square.
- The ”**SierpinskiCarpet**” class has a member variable called "board" that is a 2D array of characters, as well as a variable to store the dimension size of the array. The class also has a method to initialize the board with the character '*', and a recursive function that removes sub-arrays of characters from the board to form the Sierpinski carpet pattern. The recursive function takes the 2D array as input, and can also take additional parameters to facilitate the recursion. The class has a method to print the resulting Sierpinski carpet.

Code: is in the IntelliJ project file named practical 20.