7.2 Sorting, searching, and Graphs

- types of sorting: internal and external;
- Insertion and selection sort; Exchange sort;
- Merge and Redix sort;
- Shell sort; Heap sort as a priority queue;
- Big ‘O’ notation and Efficiency of sorting;
- Search technique; Sequential search, Binary search and Tree search;
- General search tree; Hashing: Hash function and hash tables, and Collision resolution technique,
- Undirected and Directed Graphs,
- Representation of Graph,
- Transitive closure of graph,
- Warshall’s algorithm,
- Depth First Traversal and Breadth First Traversal of Graph,
- Topological sorting (Depth first, Breadth first topological sorting),
- Minimum spanning trees (Prim’s, Kruskal’s and Round- Robin algorithms),
- Shortest-path algorithm (Greedy algorithm, and Dijkstra’s Algorithm)

Sorting :

- Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.
- if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ..... < A[N].
- For example, if we have an array that is declared and initialized as
- A[] = {21, 34, 11, 9, 1, 0, 22};
- Then the sorted array (ascending order) can be given as: A[] = {0, 1, 9, 11, 21, 22, 34;

Consider the data records given below:

Name	Department	Salary	Phone Number
Janak	Telecommunications	1000000	9812345678
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Divya	Computer Science	750000	9350123455

Now if we take department as the primary key and name as the secondary key, then the sorted order of records can be given as:

Name	Department	Salary	Phone Number
Divya	Computer Science	750000	9350123455
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Janak	Telecommunications	1000000	9812345678

Real Life Scenarios of Sorting

- Telephone directories in which names are sorted by location, category (business or residential), and then in an alphabetical order.
- In a library, the information about books can be sorted alphabetically based on titles and then by authors' names.
- Customers' addresses can be sorted based on the name of the city and then the street.

Methods of Sorting:

Internal Sorting:

- In it, all the data to be sorted is stored in main memory .
- It is performed when the data to be sorted is small enough to fit in main memory.(it is used when the size of input is small.)
- In it, the storage device used is only main memory(RAM).
- Examples : Insertion sort, Quick Sort, Bubble Sort, etc.

External Sorting :

- In it, data is stored outside the main memory like on disk and only loaded into memory in small chunks.
- It is usually applied when data can’t fit in main memory entirely. .(it is used when the size of input is large.)
- In it, the storage device used are main memory(RAM) and secondary memory(Hard Disk).
- Examples : External Merge Sort, External Radix Sort, Four Tape Sort

Sorting Algorithms

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Radix sort
- Shell sort
- Quick sort
- Heap sort

Bubble Sort/ Exchange sort

- a simple sorting algorithm which
  - repeatedly steps through the list to be sorted
  - compares each pair of adjacent items
  - swaps them if they are in the wrong order
  - The passing through the list is continued until the swapping is not required (i.e. the list sorted)
- it is a comparison sort
- It is called Bubble Sort because the data gradually bubbles up in its proper position
- In each pass at least one data is bubbled up in its proper position

Bubble Sort (Algorithm-pseudocode)

Declare and Initialize necessary variables

n => number of data items

a[n] => an array holding all the data items to be sorted flag => for checking whether the swap has been done

```
Do
    flag=0;
    for i=1 to n-1
        if(a[i-1]>a[i])
            swap (a[i-1],a[i]);  flag=1;
        end if  end for  n=n-1;
while (flag!=0)
```

Bubble Sort

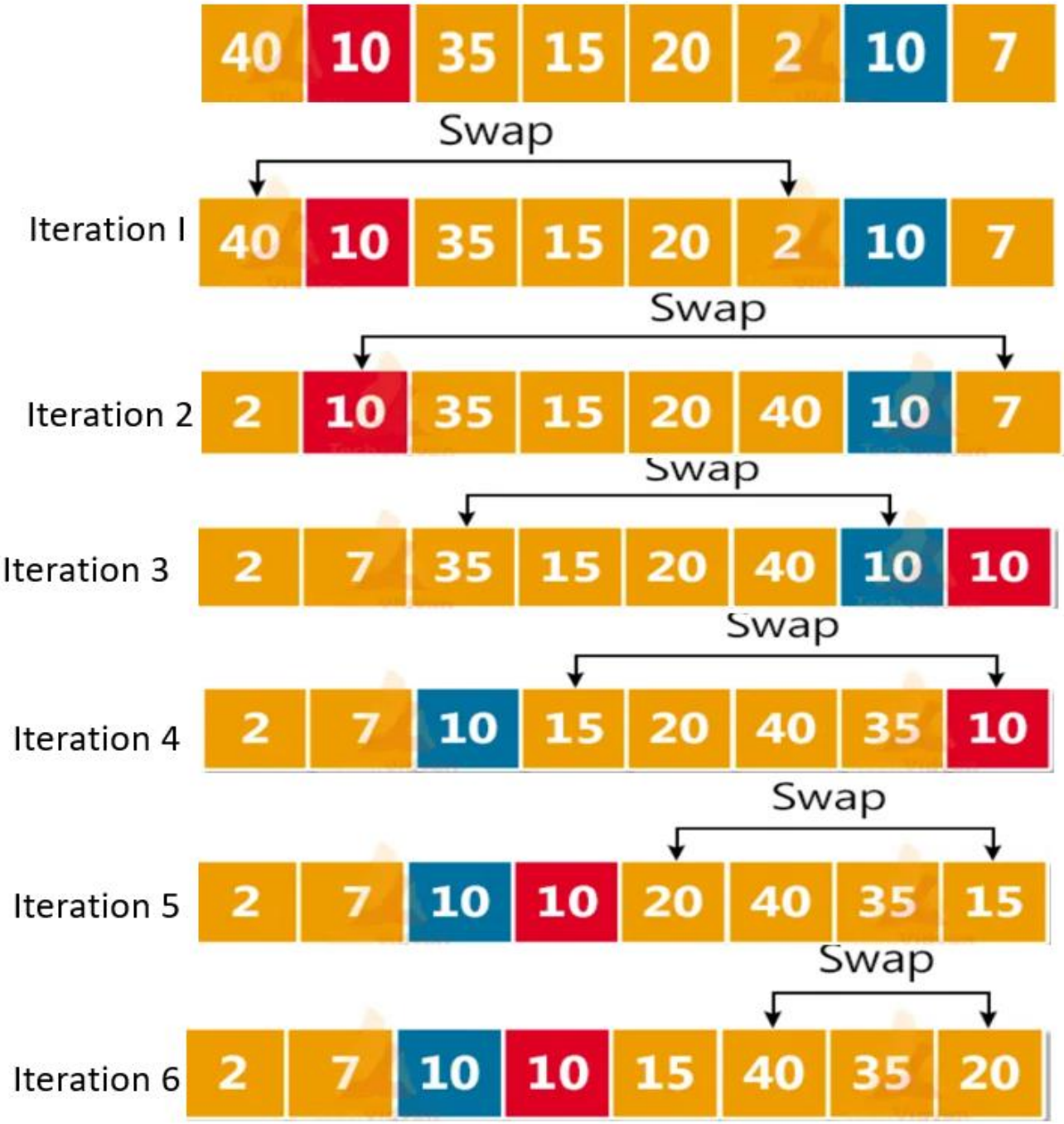
- Example : 6 1 3 2 7
- **First Pass:**
  (**6 1** 3 2 7) (**1 6** 3 2 7), Swap since 6 > 1.
  (1 **6 3** 2 7) (1 **3 6** 2 7), Swap since 6 > 3
  (1 3 **6 2** 7) (1 3 **2 6** 7), Swap since 6 > 2
  (1 3 2 **6 7**), does not swap
- **Second Pass:**
  (**1 3** 2 6 7)
  (1 **3 2** 6 7) (1 **2 3** 6 7), Swap since 3 > 2
  (1 2 **3 6** 7) (1 2 **3 6** 7)
  (1 2 3 **6 7**)
- list is already sorted, but our algorithm does not know. Hence one more pass to see if further swapping has to be done
- **Third Pass:**
- (**1 2 3 6 7**), No swap up to the last comparison, hence the list is sorted

Selection Sort:

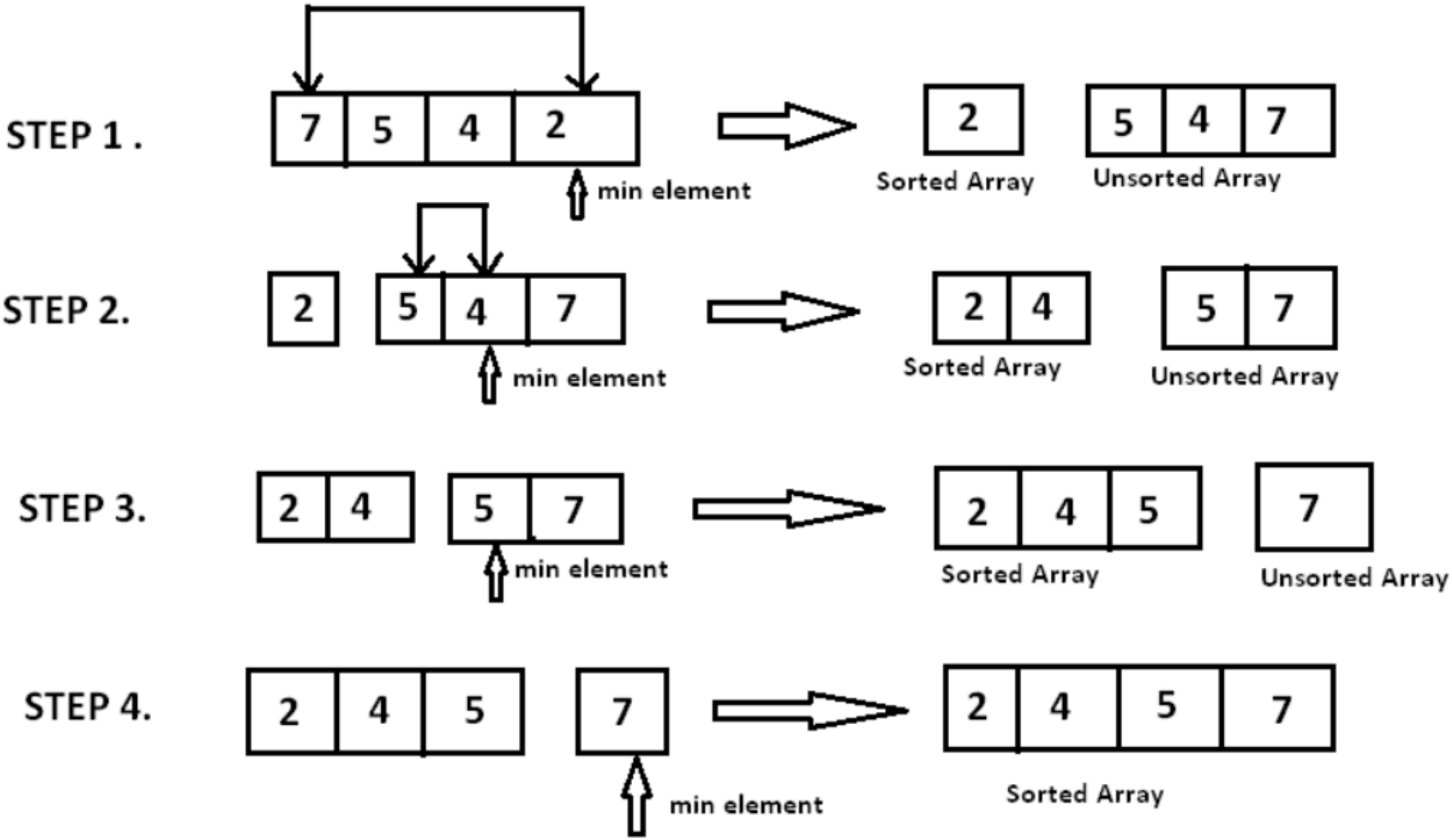
- A sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Steps involved in Selection Sort

1. Find the smallest element in the array and swap it with the first element of the array i.e. a[0].
2. The elements left for sorting are n-1 so far. Find the smallest element in the array from index 1 to n-1 i.e. a[1] to a[n-1] and swap it with a[1].
3. Continue this process for all the elements in the array until we get a sorted list.


40 10 35 15 20 2 10 7
Swap
Iteration I
40 10 35 15 20 2 10 7
Swap
Iteration 2
2 10 35 15 20 40 10 7
Swap
Iteration 3
2 7 35 15 20 40 10 10
Swap
Iteration 4
2 7 10 15 20 40 35 10
Swap
Iteration 5
2 7 10 10 20 40 35 15
Swap
Iteration 6
2 7 10 10 15 40 35 20


Iteration 7
2 7 10 10 15 20 35 40
Iteration 8
2 7 10 10 15 20 35 40
2 7 10 10 15 20 35 40


STEP 1 .
7
5
4
2
min element
2
Sorted Array
5
4
7
Unsorted Array
STEP 2.
2
5
4
7
min element
2
4
Sorted Array
5
7
Unsorted Array
STEP 3.
2
4
5
7
min element
2
4
5
Sorted Array
7
Unsorted Array
STEP 4.
2
4
5
7
min element
2
4
5
7
Sorted Array

Insertion Sort :

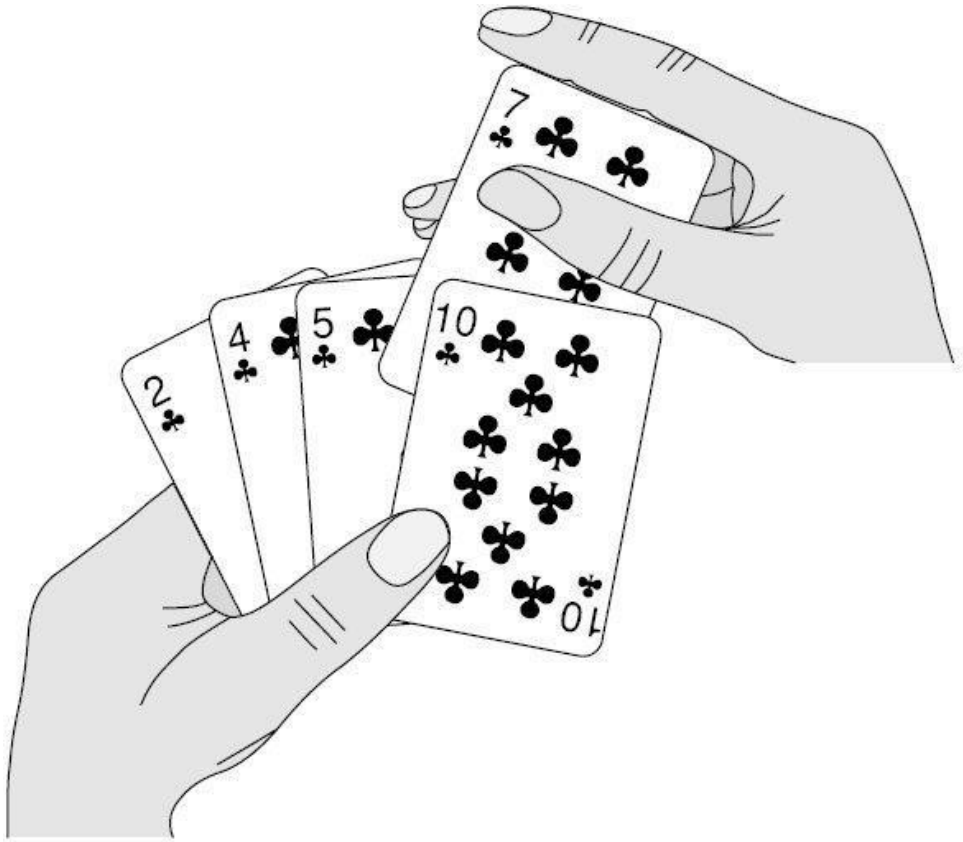
- simple sorting algorithm in which the sorted array (or list) is built one element at a time.
- We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards.
- efficient for smaller data sets, but very inefficient for larger lists.
- less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

Insertion Sort

- Insertion sort is implemented by inserting a particular data item in its proper position
- Any unsorted data item is kept on swapping with its previous data items until its proper position is not found
- The number of swapping makes the previous data items to shift for the new data item to take its position in order
- Once the new data item is inserted, the next data item after it is chosen for next insertion
- The process continues until all data items are sorted
- This method is highly efficient if the list is almost in sorted form

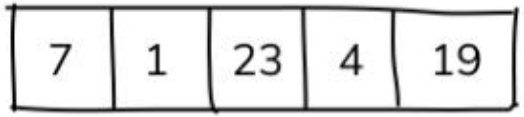
Insertion Sort

- Its similar to arrangement of cards during card game

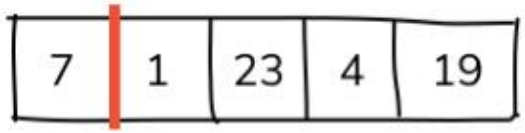


Let us consider an array with 5 elements, **A = [7, 1, 23, 4, 19]**.

Given unsorted array →

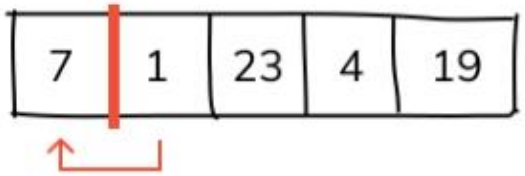


Consider the first element as sorted as there are no other elements on its left-hand side

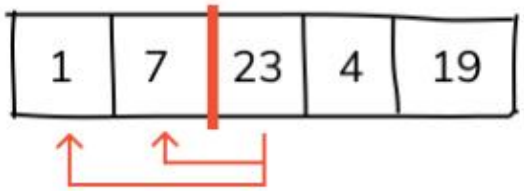


Look at the next unsorted element (7) & compare it with the sorted elements (1)

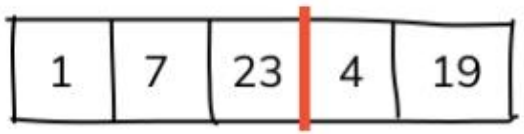
1 < 7 ?
Yes, so swap



23 < 7 ? ---> No
23 < 1 ? ---> No
So no swaps in this iteration

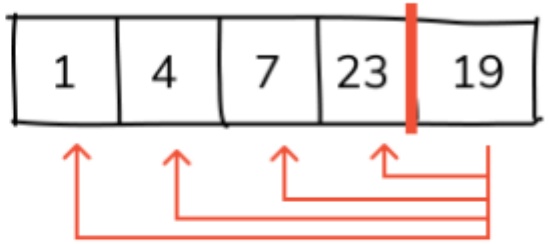


4 < 23 ? ---> Yes, so check the next number
4 < 7 ? ---> Yes, so check the next number
4 < 1 ? ---> No

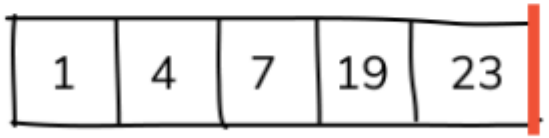


So the correct position of 4 is the current position of 7. **So shift all the elements from 7 to one position ahead.**

19 < 23 ? ---> Yes, so check the next number
19 < 7 ? ---> No



So the correct position of 19 is the current position of 23. **So shift the element 23 one position ahead.**



The entire array is now sorted.

Divide and Conquer Algorithm

- Merge Sort
- Quick Sort

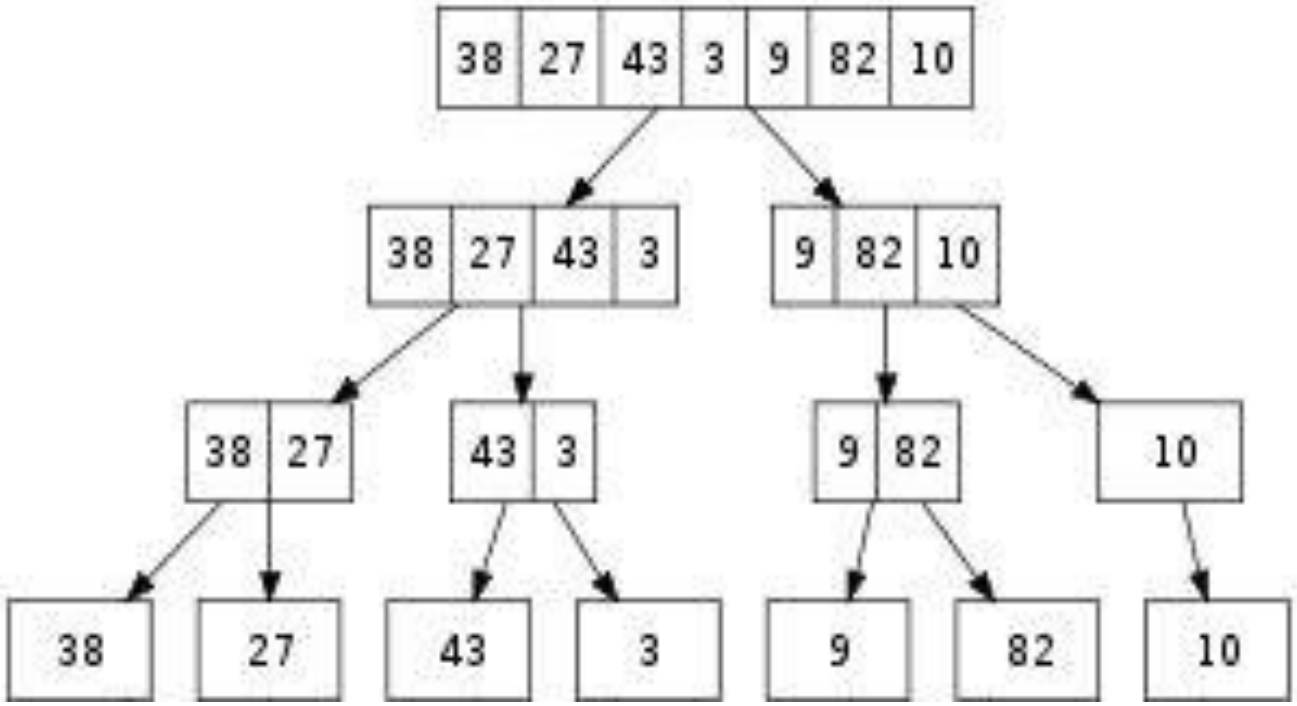
Merge Sort

- It is a divide and conquer algorithm
- At first we divide the given list of item
  - list is divided into two parts from middle
  - The process is repeated until each sublist contain exactly 1 item
- Now is the turn for sort and combine (conquer)
  - A list with a single element is considered sorted automatically
  - Pair of list is sorted and merged into one (i.e. approx. n/2 sublists of size 2)
  - The sort and merge is keep on repeated until a single list of size n is found
- The overall dividing and conquering is done recursively

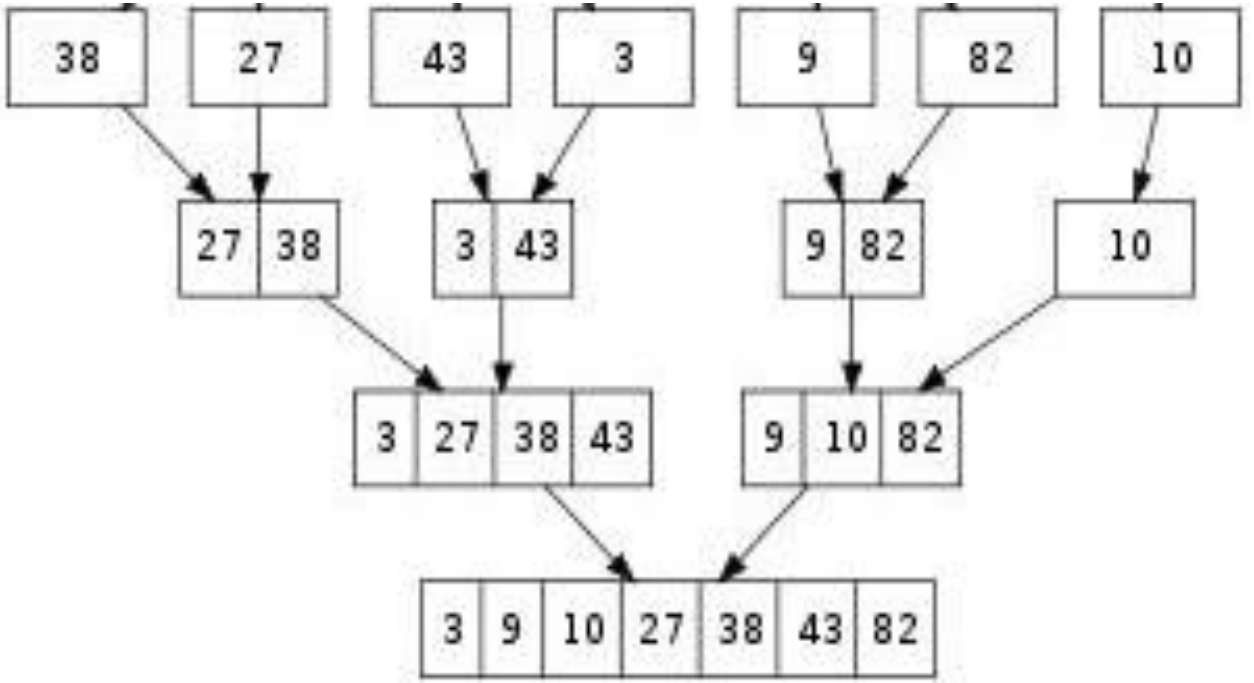
Merge Sort

- To sort $A[p \ldots r]$: (p=starting index , r=ending index)
- 1. **Divide Step**
  - If a given array A has zero or one element, simply return; it is already sorted.
  - Otherwise, split $A[p \,..\, r]$ into two subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$, each containing about half of the elements of $A[p \,..\, r]$. That is,
  - q is the halfway point of $A[p \,..\, r]$.
- 2. **Conquer Step**
  - Conquer by recursively sorting the two subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$.
- 3. **Combine Step**
  - Combine the elements back in $A[p \,..\, r]$ by merging the two sorted subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ into a sorted sequence.
  - To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Merge Sort



Divide



Conquer

Radix Sort

- Radix sort is the linear sorting algorithm that is used for integers.
- In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.
- The process of radix sort works similar to the sorting of students names, according to the alphabetical order.

Algorithm:

1. Find largest element in the given array and number of digits in the largest element.
2. Define 10 queues each representing a bucket for each digit from 0 to 9.
3. Consider the least significant digit of each number in the list which is to be sorted.
4. Insert each number into their respective queue based on the least significant digit.
5. Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
6. Repeat from step 4 based on the next least significant digit.
7. Repeat from step 3 until all the numbers are grouped based on the most significant digit.

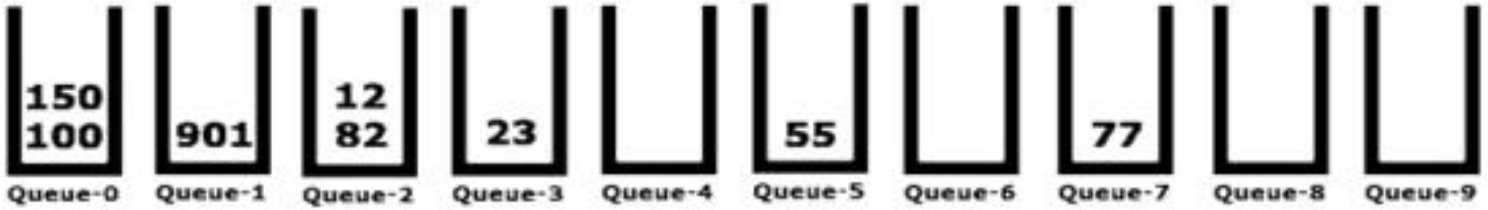
Consider the following list of unsorted integer numbers:

82,901,100,12,150,77,55,23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8<u>2</u>, 90<u>1</u>, 10<u>0</u>, 1<u>2</u>, 15<u>0</u>, 7<u>7</u>, 5<u>5</u> & 2<u>3</u>

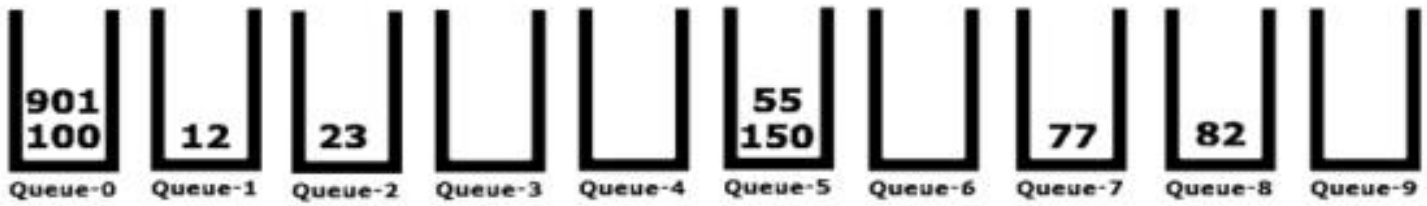


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1**0**0, 1**5**0, 9**0**1, **8**2, **1**2, **2**3, **5**5 & **7**7

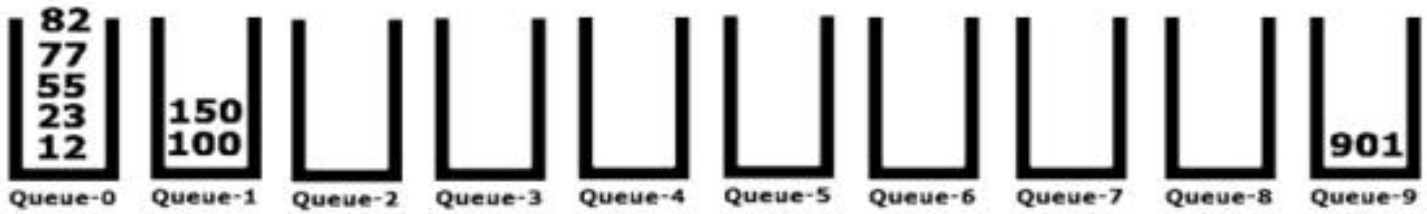


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, **9**01, 12, 23, **1**50, 55, 77 & 82



Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

Shell Sort:

- Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.
- It can be shown as a generalization of either exchange bubble sorting or insertion sorting.
- It first sorts elements that are far apart from each other by swapping and successively reduces the gap between the elements to be sorted. This gap is called as interval. The interval between the elements is reduced based on the sequence used.
- Shell's original sequence: N/2 ,N/4 , … , 1
- Knuth's Formula = h ∗ 3 + 1 where − h is interval with initial value 1

Algorithm:

for the size of array ‘N’:

1. Divide the list into smaller sub-list of interval N/2.
2. Sort these sub-lists using insertion sort.
3. Repeat until complete list is sorted.

Shell Sort(a, n) // 'a' is the given array, 'n' is the size of array

for ($interval$ = n/2; $interval$ >= 1; $interval$ /= 2)

for (j = $interval$; j < n; j + +)

for (i = j – $interval$; i >= 0; i –= $interval$)

if (a[i + $interval$] > a[i]) o $break$

$otherwise$ o $swap$ (a[i + $interval$], a[i])

$End\ Shell\ Sort$

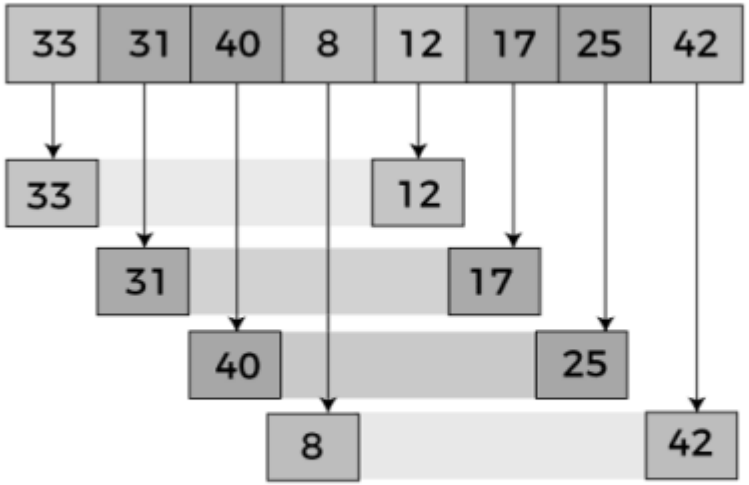
Example:

33	31	40	8	12	17	25	42

We will use the original sequence of shell sort, i.e., N/2, N/4...1 as the intervals. Here, in the first loop, the element at the 0th position will be compared with the element at 4th position.

If the 0th element is greater, it will be swapped with the element at 4th position. Otherwise, it remains the same. This process will continue for the remaining elements

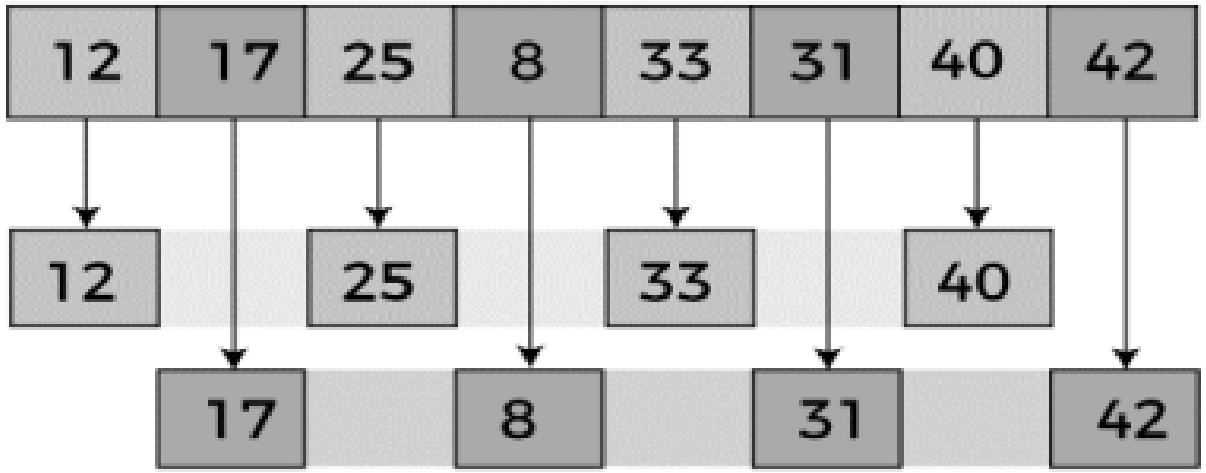
At the interval of 4, the sub lists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.



After comparing and swapping, the updated array will look as follows –

12	17	25	8	33	31	40	42

In the second loop, elements are lying at the interval of 2 (n/4 = 2), where n = 8.



After comparing and swapping, the updated array will look as follows –

In the third loop, elements are lying at the interval of 1 (n/8 = 1), where n = 8.

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Heap Sort:

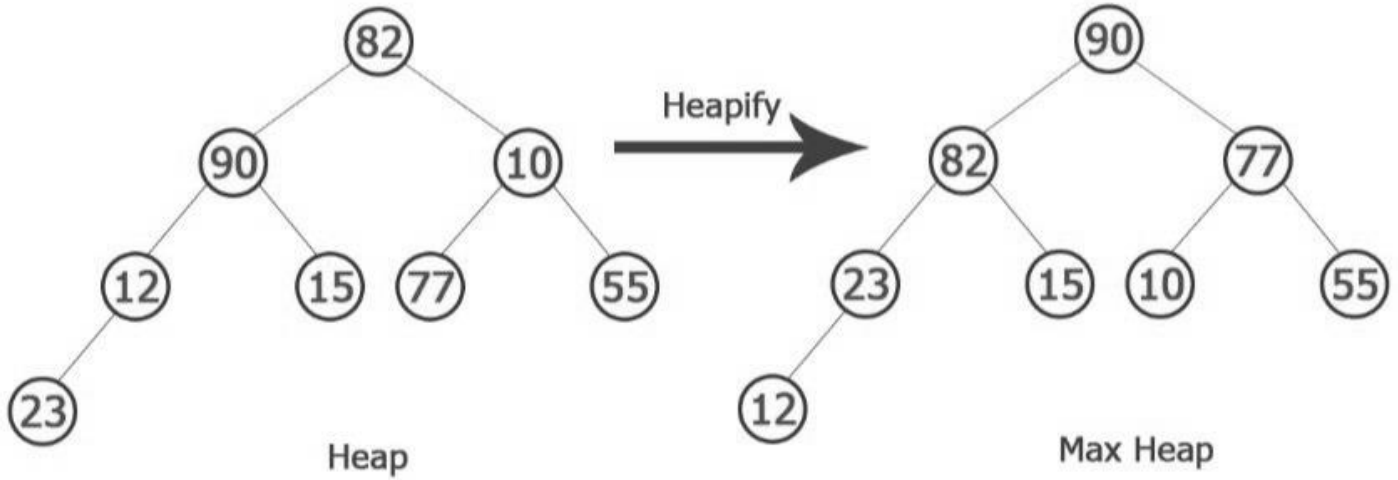
- Heap sort is one of the sorting algorithms used to arrange a list of elements in order.
- Heapsort algorithm uses one of the tree concepts called **Heap Tree**.
- In this sorting algorithm, we use **max Heap** to arrange list of elements in Descending order and **min Heap** to arrange list elements in Ascending order.

Algorithm:

1. Construct a Binary Tree with given list of Elements.
2. Transform the Binary Tree into Max Heap.
3. Delete the root element from Max Heap using Heapify method.
4. Put the deleted element into the Sorted list.
5. Repeat the same until Max Heap becomes empty.
6. Display the sorted list.

Consider the following list of unsorted numbers which are to be sort using Heap sort
82,90,10,12,15,77,55,23

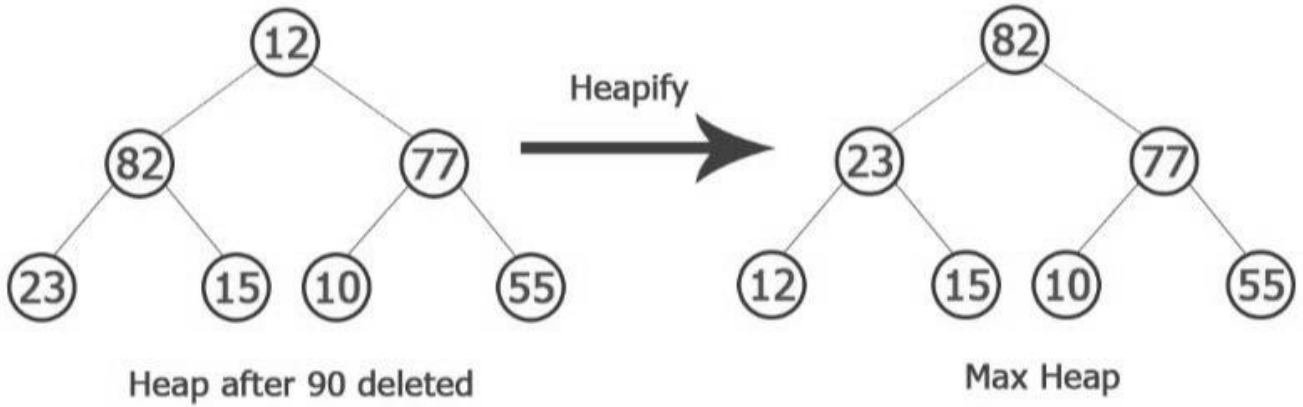
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

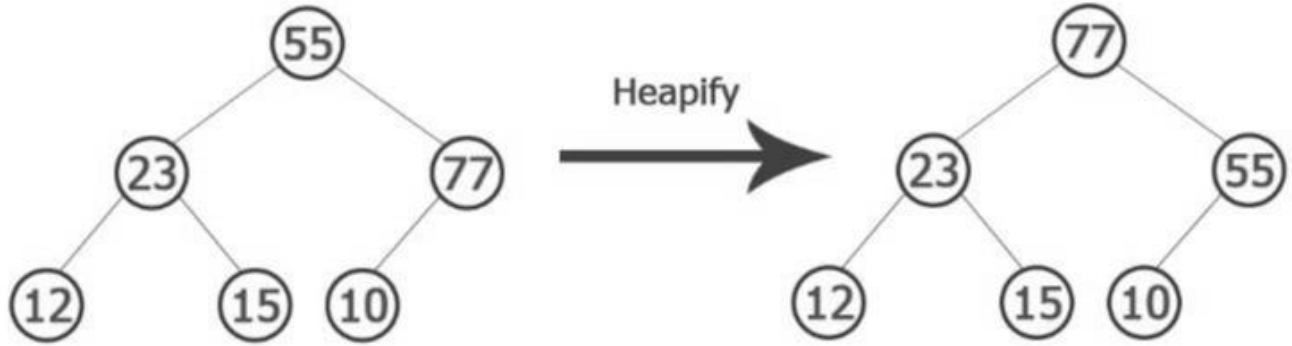
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



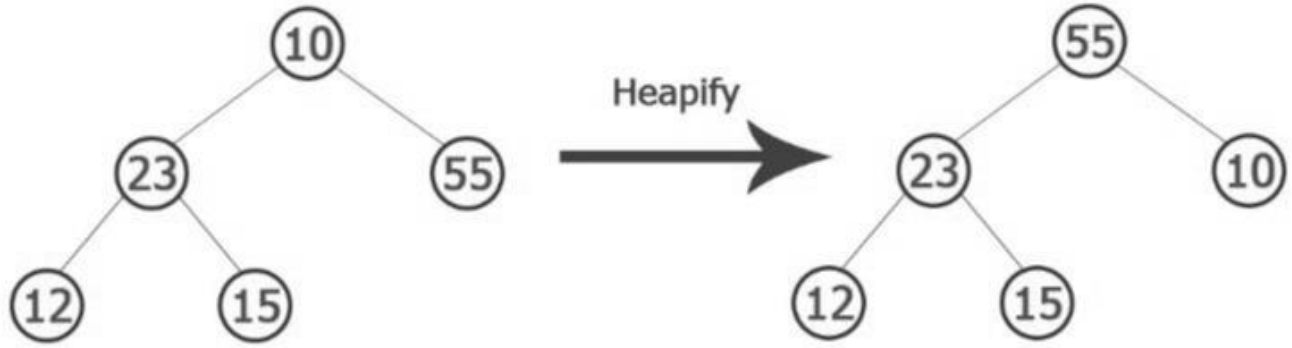
Heap after 82 deleted

Max Heap

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82**, **90**

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



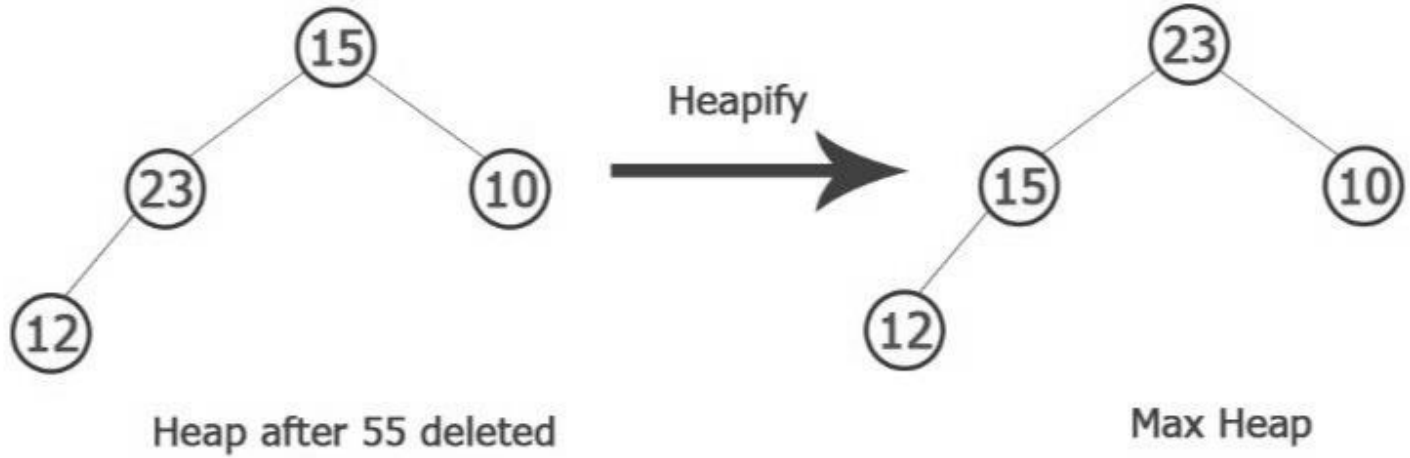
Heap after 77 deleted

Max Heap

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77**, **82**, **90**

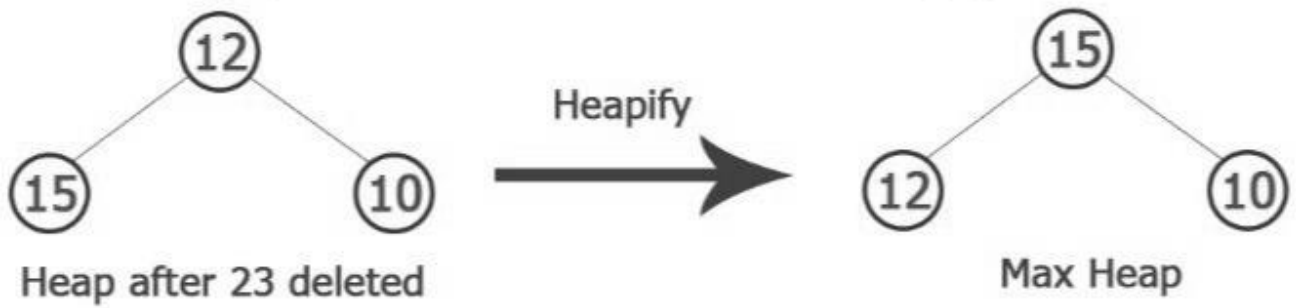
Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

12, 15, 10, 23, **55**, **77**, **82**, **90**

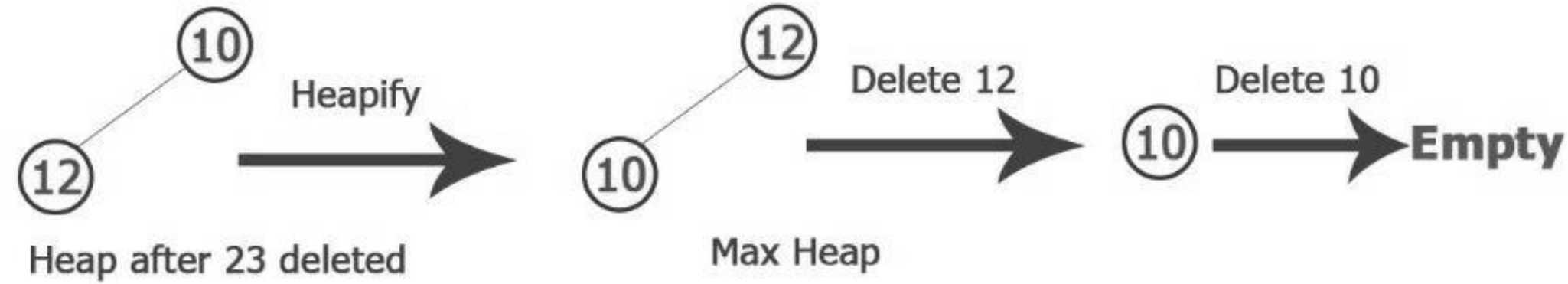
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

12, 15, 10, **23**, **55**, **77**, **82**, **90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Time Complexity of Sorting Algorithms

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

Searching :

- It is a process of finding an element within the list of elements stored in any order.
- It is not necessary that the data item we are searching for must be present in the list.
- If the searched item is present in the list then the searching algorithm (or program) can find that data item, in which case we say that the search is successful, but if the searched item is not present in the list, then it cannot be found and we say that the search is unsuccessful.

Types of Searching

1. Linear /Sequential Searching :

- It is the simplest technique to find out an element in an unordered list.
- We search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.
- Suppose there are ‘n’ elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison‘s to search an element. If search is not successful, you would need ‘n’ comparisons. **The time complexity of linear search is O(n).**

Steps :

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.
- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

- Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

Index	0	1	2	3	4	5	6	7	8	9	10	11
list	45	39	8	54	77	38	24	16	4	7	9	20

- 24 is compared with first element(45).If not matched, move to next element.
- 24 is compared with second element(39).If not matched, move to next element.
- 24 is compared with third element(8).If not matched, move to next element.
- 24 is compared with fourth element(54).If not matched, move to next element.
- 24 is compared with fifth element(77).If not matched, move to next element.
- 24 is compared with Sixth element(38).If not matched, move to next element.
- 24 is compared with seventh element(24).Matched.

Let us illustrate linear search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

- Searching for x = 7 Search successful, data found at 3rd position.
- Searching for x = 82 Search successful, data found at 7th positior
- Searching for x = 42 Search un-successful, data not found.

```
main()
{
        int number[25], n, data, i, flag = 0;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements:
        "); for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be Searched: ");
        scanf("%d", &data);
        for( i = 0; i < n; i++)
        {
                if(number[i] == data)
                {
                        flag = 1;
                        break;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", i+1);
        else
                printf("\n Data not found ");
}
```

Sequential search efficiency

- The number of comparisons of keys done in sequential search of a list of length n is

i. Unsuccessful search: n comparisons

ii. Successful search, best case: 1 comparison

iii. Successful search, worst case: n comparisons

iv. Successful search, average case: (n + 1)/2 comparisons v.

In any case, the number of comparison is O(n)

2. Binary Search :

- It is an extremely efficient algorithm.
- This search technique searches the given item in minimum possible comparisons.
- To do the binary search, first we have to sort the array elements.
- The logic behind this technique is given below.

i. First find the middle element of the array.

ii. Compare the middle element with an item.

iii. There are three cases:

a) If it is a desired element then search is successful,

b) If it is less than the desired item then search only in the first half of the array.

c) If it is greater than the desired item, search in the second half of the array.

4. Repeat the same steps until an element is found or search area is exhausted.

In this way, at each step we reduce the length of the list to be searched by half.

In this algorithm every time we are reducing the search area.

We can apply binary search technique **recursively** or **iteratively.**

As the list is divided into two halves, searching begins

- Now we check if the searched item is greater or less than the center element.
- If the element is smaller than the center element then the searching is done in the first half, otherwise it is done in the second half.
- The process is repeated till the element is found or the division of half parts gives one element.

Requirements :

i. The list must be ordered.

ii. Rapid random access is required, so we cannot use binary search for a linked list.

Binary search efficiency

i. In all cases, the no. of comparisons in proportional to n

ii. Hence, no. of comparisons in binary search is O (log n), where n is the no of items in the list

iii. Obviously binary search is faster then sequential search, but there is an extra overhead in maintaining the list ordered

iv. For small lists, better to use sequential search

v. Binary search is best suited for lists that are constructed and sorted once, and then repeatedly searched

Algorithm :

Given a table k of n elements in searching order searching for value x.

1.Initialize : low ← 0, high ←n-1

2. Perform Search : Repeat through step 4 while low <= high .

3. Obtain index of midpoint of interval : mid ←(low + high)/2

4.Compare :

if X<k[mid] then high ←mid -1

Else if X>k[mid] then low ←mid -1

Else Write (“Search is unsuccessful”)

Return (mid)

5. (“Search is unsuccessful”)

Return

6. Finished

0	1	2	3	4	5	6	7	8	9
22	43	68	100	120	330	420	555	560	570

Search 43

Insertion	Low	High	Mid	Remarks
1.	0	9	4	X<k[4]
2.	0	3	1	X=k[1]

Data found in Location 1.

Search 555

Insertion	Low	High	Mid	Remarks
1.	0	9	4	X>k[4]
2.	5	9	7	X=k[7]

Data found in location 7.

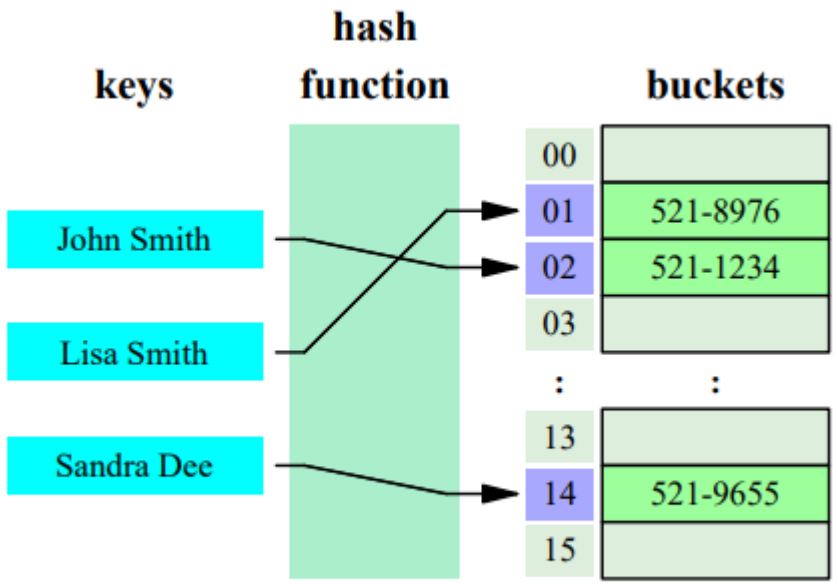
Search 333

Insertion	Low	High	Mid	Remarks
1.	0	9	4	X>k[4]
2.	5	9	7	X<k[7]
3.	5	6	5	X>k[5]
4.	6	6	6	X<k[6]
5.	6	5	5	Low>High

Search Value 333 is not found.

Hashing :

- It is the technique of representing longer records by shorter values called keys.
- technique used for storing and retrieving information as quickly as possible.
- The keys are placed in a table called hash table where the keys are compared for finding the roots.



Hash Table

Hash Tables :

- A hash table is a data structure where data is stored in an associative manner. The data is mapped to array positions by a hash function that generates a unique value from each key.

Hash function :

- It is a mathematical formula which, when applied to a key, produces a value which can be used as an index for the key in the hash table.
- The main aim of a hash function is that elements should be uniformly distributed. It produces a unique set of integers within some suitable range in order to reduce the number of collisions.
- In practice, there is no hash function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

Characteristics of Good Hash Function

A good hash function should have the following characteristics:

- Minimize collision
- Be easy and quick to compute
- Distribute key values evenly in the hash table
- Use all the information provided in the key
- Have a high load factor for a given set of keys

Different Hash Functions

1. Folding Method:

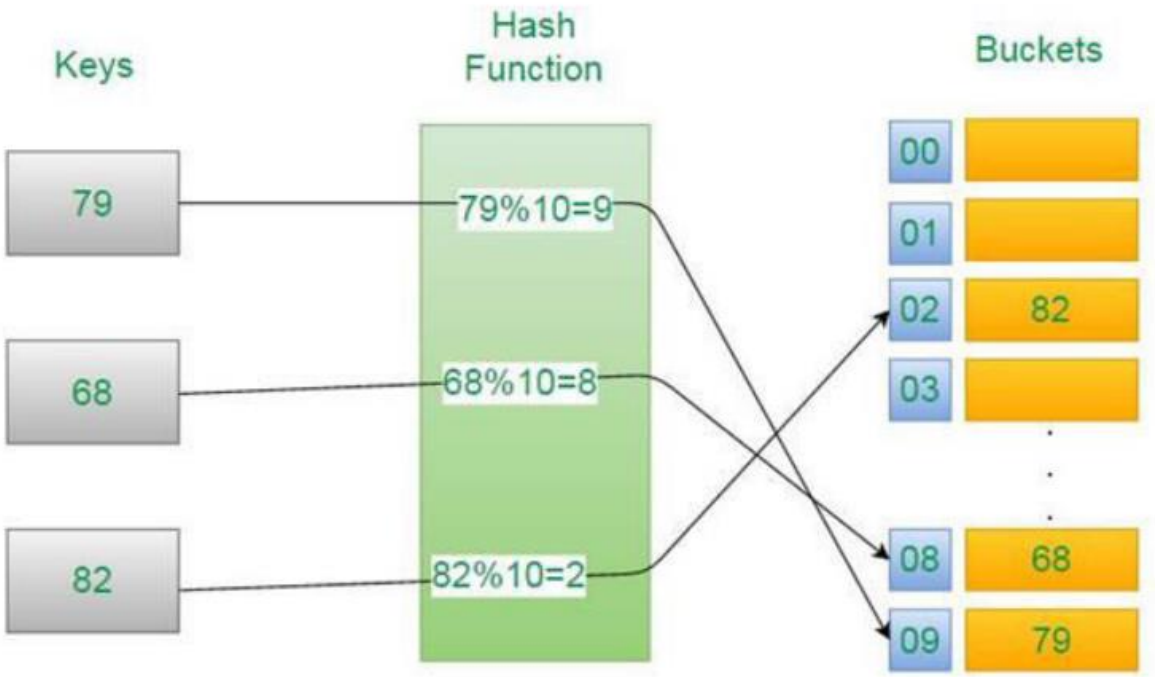
- In this method, the given key is partitioned into subparts k_1, k_2, k_3, k_4 ..... k_n each of which has the same length as the required address. Now add all these parts together and ignore the carry.
- For example:- if number of buckets be 100 and last address/index be 99, then the given key for which hashcode is calculated is divided into parts of two digits from beginning as shown below:
- h(95073) = h(95 + 07 + 3)

  = h(105) //ignoring the carry = 5
- Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.
- Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

Key	**5678**	**321**	**34567**
Parts	56 and 78	32 and 1	34,56 and 7
Sum	134	33	97
Hash Value	34(ignore the last carry)	33	97

2. Division Method :

- It is the most simple method of hashing an integer x. This method divides x by M(slots available) and then uses the remainder obtained.
- In this case, the hash function can be given as **h(x) = x mod M**
- For example:- Let us say apply division approach to find hash value for some values considering number of buckets be 10 as shown below.



3. Mid-Square Method :

- It is a good hash function which works in two steps:

  Step 1: Square the value of the key. That is, find k^2 .

  Step 2: Extract the middle r digits of the result obtained in Step 1.

- The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.
- In it, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as: **h(k) = s** where s is obtained by selecting r digits from k^2.
- Example : Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.
- Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2.
- When k = 1234, k^2 = 1522756, h (1234) = 27
- When k = 5642, k^2 = 31832164, h (5642) = 21
- Observe that the 3rd and 4th digits starting from the right are chosen.

Collisions :

- Collisions occur when the hash function maps two different keys to the same location.
- Two records cannot be stored in the same location of a hash table normally.

Collision resolution techniques:

1. Open Addressing (Closed Hashing):

a. Linear Probing

b. Quadratic Hashing

c. Double Hashing

2. Chaining(Open Hashing)

a. Linear Probing :

- Calculate the hash key. **h´(k) = k** $mod\ m$
- If hashTable[key] is empty, store the value directly. hashTable[key] = data.
- If the hash index already has some value, check for next index.
- h**(k,** i**) = (**h**´(k) +** i**)**$mod\ m$**;**
- If the next index is available hashTable[key], store the value. Otherwise try for next index.
- Do the above process till we find the space.

- Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81,92 and 101 into the table.

- Let h'(k) = k mod m, m = 10

k	**h(k,i)=(h'(k)+i) mod 10**
72	h(72,0)=(72 mod 10 + 0) mod 10=2 mod 10 =2
27	h(27,0)=(27 mod 10 + 0) mod 10=7 mod 10 =7
36	h(36,0)=(36 mod 10 + 0) mod 10=6 mod 10 =6
24	h(24,0)=(24 mod 10 + 0) mod 10=4 mod 10 =4
63	h(63,0)=(63 mod 10 + 0) mod 10=3 mod 10 =3
81	h(81,0)=(81 mod 10 + 0) mod 10=1 mod 10 =1
92	h(92,0)=(92 mod 10 + 0) mod 10=2 mod 10 =2(A[2] is occupied) Then i=1, h(92,1)=(92 mod 10 + 1) mod 10=3 mod 10 =3(A[3] is occupied) Then i=2, h(92,2)=(92 mod 10 + 2) mod 10=4 mod 10 =4(A[4] is occupied) Then i=3, h(92,3)=(92 mod 10 + 3) mod 10=5 mod 10 =5
101	h(101,0)=(101 mod 10 + 0) mod 10=1 mod 10 =1 (A[1] is occupied) Then i=1, h(101,1)=(101 mod 10 + 1) mod 10=2 mod 10 =2(A[2] is occupied) Repeat process until i=7.

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	92	36	27		

b. Quadratic Hashing:

- In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision**: $h(k, i) = [h'(k) + i^2] \bmod m$** where m is the size of the hash table, h'(k) = (k mod m), i is the probe number that varies from 0 to m–1.
- eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search.

- Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table.

- Let h'(k) = k mod m, m = 10

k	**$h(k,i)=(h'(k)+i^2) \bmod 10$**
72	$h(72,0)=(72 \bmod 10 + 0^2) \bmod 10=2 \bmod 10 =2$
27	$h(27,0)=(27 \bmod 10 + 0^2) \bmod 10=7 \bmod 10 =7$
36	$h(36,0)=(36 \bmod 10 + 0^2) \bmod 10=6 \bmod 10 =6$
24	$h(24,0)=(24 \bmod 10 + 0^2) \bmod 10=4 \bmod 10 =4$
63	$h(63,0)=(63 \bmod 10 + 0^2) \bmod 10=3 \bmod 10 =3$
81	$h(81,0)=(81 \bmod 10 + 0^2) \bmod 10=1 \bmod 10 =1$
101	$h(101,0)=(101 \bmod 10 + 0^2) \bmod 10=1 \bmod 10 =1$ $h(101,1)=(101 \bmod 10 + 1^2) \bmod 10=2 \bmod 10 =2$ $h(101,2)=(101 \bmod 10 + 2^2) \bmod 10=5 \bmod 10 =5$

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	101	36	27		

c. Double Hashing :

- It uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name double hashing.
- In double hashing, we use two hash functions rather than a single function. The hash function in the case of double hashing can be given as:
- **$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$**
- where m is the size of the hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as h1 (k) = k mod m, h2 (k) = k mod m', i is the probe number that varies from 0 to m–1, and m' is chosen to be less than m. We can choose m' = m–1 or m–2.

- Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table. Take h1 = (k mod 10) and h2 = (k mod 8). [Let m = 10]

k	**$h(k,i)=[h_1(k)+ih_2(k)]\bmod m$**
72	h(72,0)=[72 mod 10 + 0(72 mod 8)]mod 10=2
27	h(27,0)=[27 mod 10 + 0(27 mod 8)]mod 10=7
36	h(36,0)=[36 mod 10 + 0(36 mod 8)]mod 10=6
24	h(24,0)=[24 mod 10 + 0(24 mod 8)]mod 10=4
63	h(63,0)=[63 mod 10 + 0(63 mod 8)]mod 10=3
81	h(81,0)=[81 mod 10 + 0(81 mod 8)]mod 10=1
92	h(92,0)=[92 mod 10 + 0(92 mod 8)]mod 10=2 [Collision since A[2] is occupied.] h(92,1)=[92 mod 10 + 1(92 mod 8)]mod 10=(2+4) mod 10 = 6 [Collision since A[6] is occupied.] h(92,2)=[92 mod 10 + 2(92 mod 8)]mod 10=(2+2*4) mod 10= 0
101	h(101,0)=[101 mod 10 + 0(101 mod 8)]mod 10=1[Collision since A[1] is occupied.] h(101,1)=[101 mod 10 + 1(101 mod 8)]mod 10=6[Collision since A[6] is occupied.] h(101,2)=[101 mod 10 + 2(101 mod 8)]mod 10=1[Collision since A[1] is occupied.] Repeat the entire process until a vacant location is found. We will see that we have to probe many times to insert the key 101 in the hash table.

2. Chaining(Open Hashing)

- In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. As new collisions occur, the linked list grows to accommodate those collisions forming a chain.
- Chained hash tables with linked lists are widely used due to the simplicity of the algorithms to insert, delete, and search a key. The code for these algorithms is exactly the same as that for inserting, deleting, and searching a value in a single linked list

- Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use h(k) = k mod m. In this case, m=9. Initially, the hash table can be given as:

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

Step 1

Key = 7

h(k) = 7 mod 9

= 7

Create a linked list for location 7 and store the key value 7 in it as its only node.

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	→ 7 | X
8	NULL

Step 2

Key = 24

h(k) = 24 mod 9

= 6

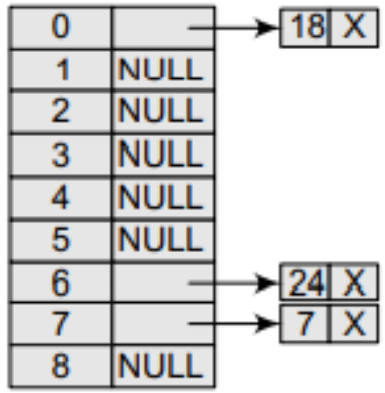
Create a linked list for location 6 and store the key value 24 in it as its only node.

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	→ 24 | X
7	→ 7 | X
8	NULL

Step 3 Key = 18

h(k)= 18 mod 9 = 0

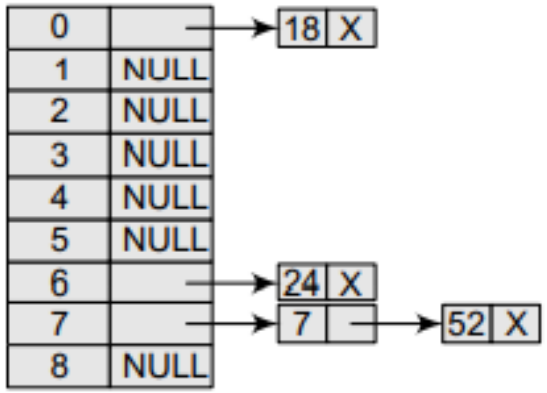
Create a linked list for location 0 and store the key value 18 in it as its only node.



Step 4 Key = 52

h(k) = 52 mod 9 = 7

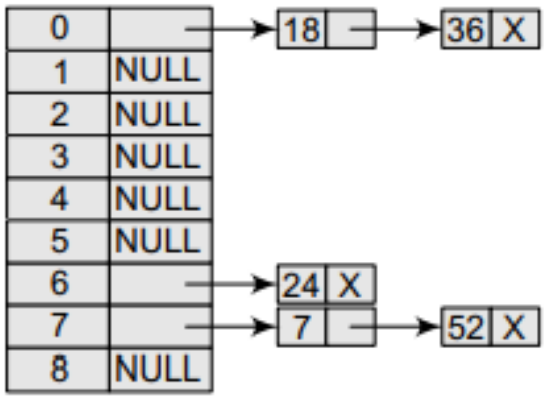
Insert 52 at the end of the linked list of location 7.



Step 5: Key = 36

h(k)= 36 mod 9 = 0

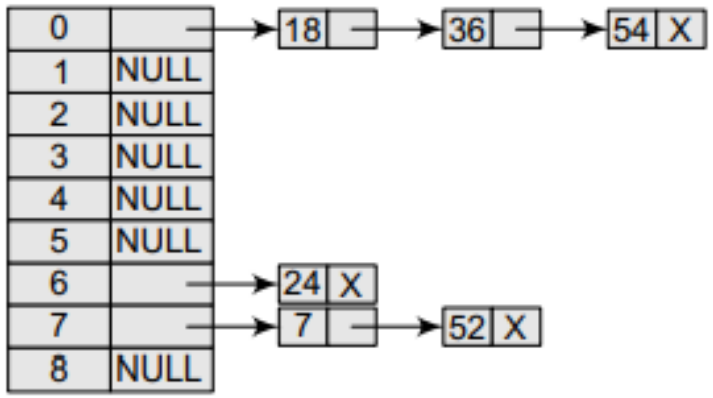
Insert 36 at the end of the linked list of location 0.



Step 6: Key = 54

h(k) = 54 mod 9 = 0

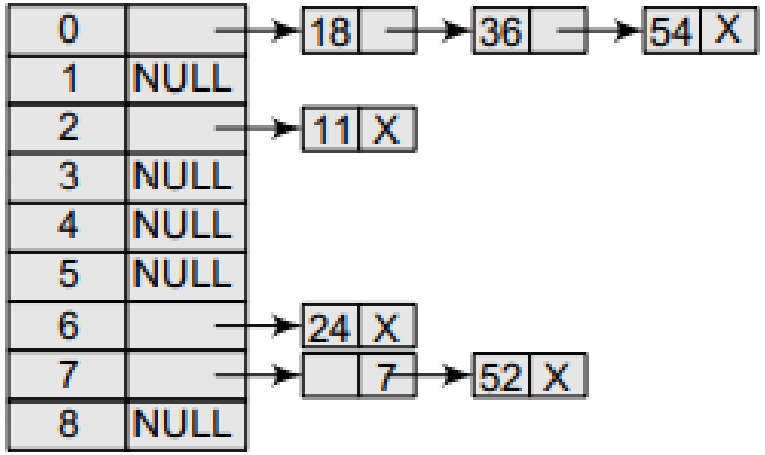
Insert 54 at the end of the linked list of location 0.



Step 7: `Key = 11`

`h(k)= 11 mod 9 = 2`

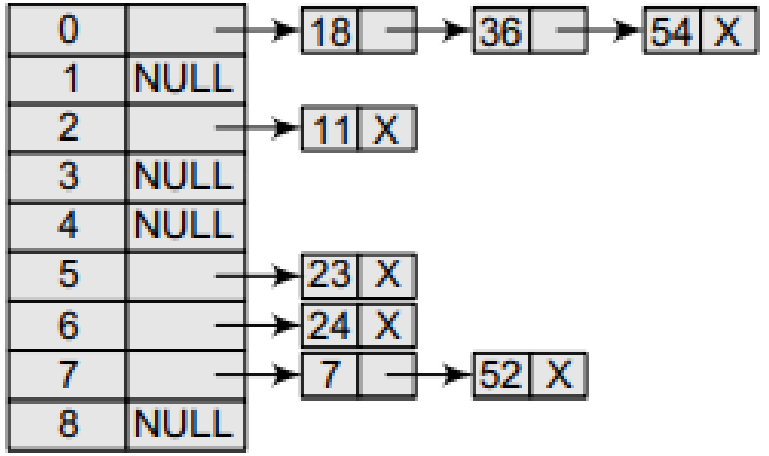
Create a linked list for location 2 and store the key value 11 in it as its only node.



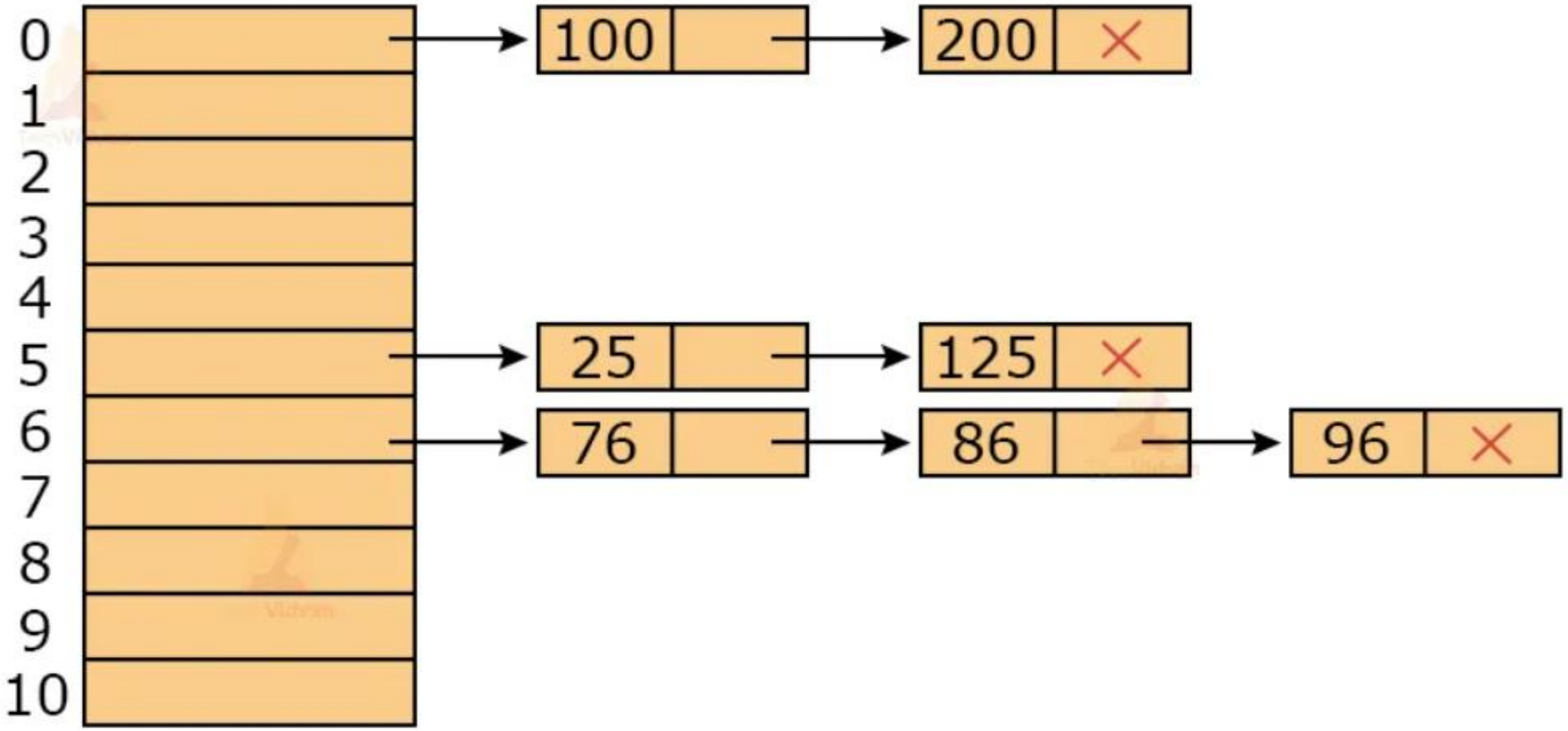
Step 8: `Key = 23`

`h(k) = 23 mod 9 = 5`

Create a linked list for location 5 and store the key value 23 in it as its only node.

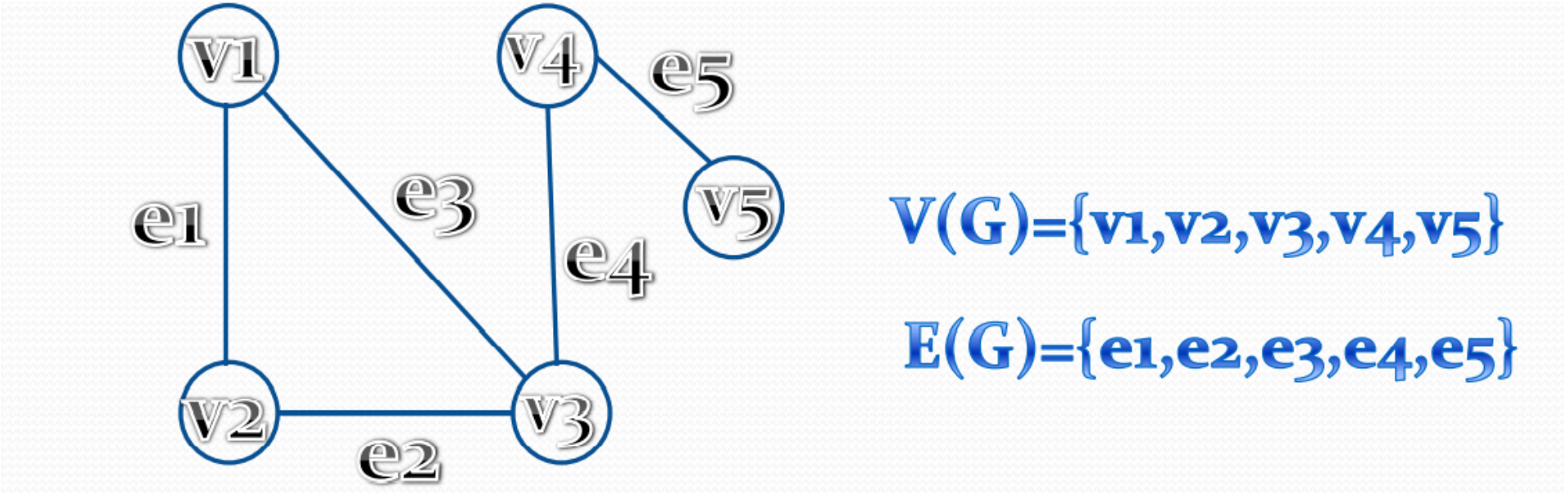


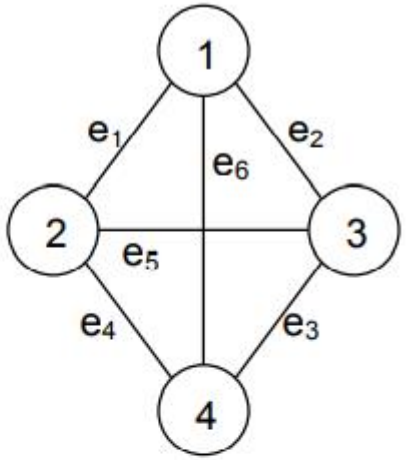
- let the keys be 100, 200, 25, 125, 76, 86, 96 and let m = 10. Given, h(k) = k mod 10
  Then, h(100) = 100 mod 10 = 0
  h(200) = 200 mod 10 = 0
  h(25) = 25 mod 10 = 5
  h(125) = 125 mod 10 = 5
  h(76) = 76 mod 10 = 6
  h(86) = 86 mod 10 = 6
  h(96) = 96 mod 10 = 6



Introduction to Graph

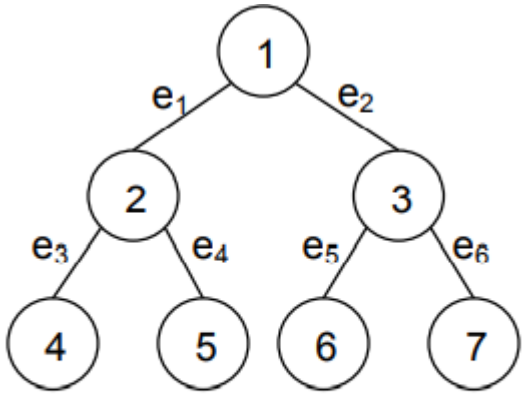
- A graph **G** consists of a set **V** of vertices (nodes) and a set **E** of edges (or arcs) which is a pair of vertices.
- We write **G=(V, E)**, **V** is a finite and non-empty set of vertices, and **E** is a set of pairs of vertices called edges.
- Thus, **V(G)**, read as **V** of **G**, is set of vertices and **E(G)**, read as **E** of **G**, is set of edges.
- An edge **$e_1=(v_1, v_2)$**, is a pair of vertices.





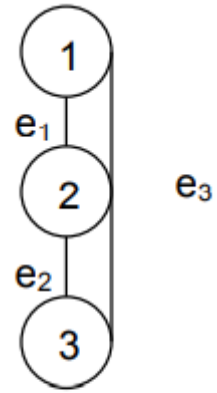
G_1

$V(G_1) = \{1, 2, 3, 4\}$
$E(G_1) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$
$e_1 = (1, 2)$
$e_2 = (1, 3)$
$e_3 = (3, 4)$
$e_4 = (2, 4)$
$e_5 = (2, 3)$
$e_6 = (1, 4)$



G_2

$V(G_2)=\{1,2,3,4,5,6,7\}$
$E(G_2)=\{e_1, e_2, e_3, e_4, e_5, e_6\}$
$e_1 =(1,2)$
$e_2 =(1,3)$
$e_3 =(2,4)$
$e_4 =(2,5)$
$e_5 =(3,6)$
$e_6 =(3,7)$



G_3

$V(G_3) = \{1, 2, 3\}$
$E(G_3) = \{e_1, e_2, e_3\}$
$e_1 = (1, 2)$
$e_2 = (2, 3)$
$e_3 = (1, 3)$

Applications of Graph

- Representing relationships between components in electronic circuits
- Transportation networks: Highway network, Flight network
- Computer networks: Local area network, Internet, Web
- Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases

Types of Edges :

1. **Undirected Edge** - An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

2. **Directed Edge** - A directed edge is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

3. **Weighted Edge** - A weighted edge is a edge with value (cost) on it.

Directed Graph (or) Digraph

- It is a graph in which each edge has a direction to its successor.
- It is a graph with only directed edges.

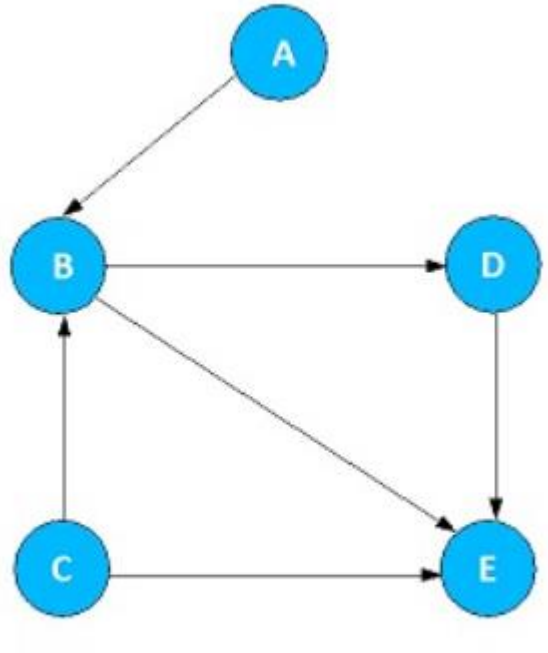


Figure 1. Directed Graph

Undirected Graph

- It is a graph in which there is no direction on the edges. The flow between two vertices can go in either direction.

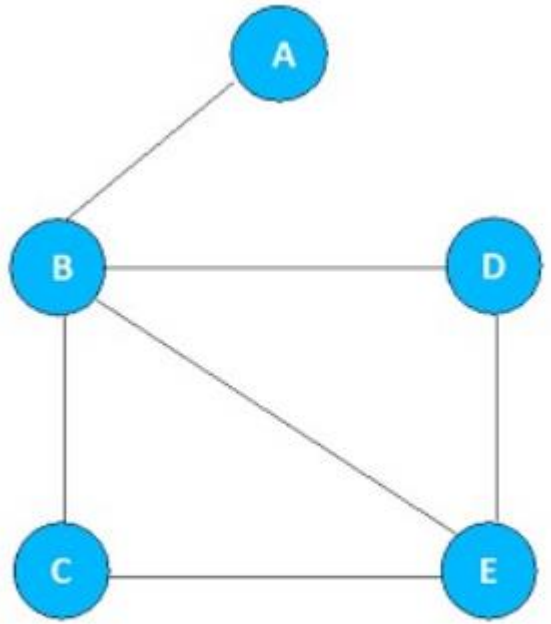


Figure 2. Undirected Graph

- **Connected Graph**: An undirected graph is called connected if there is a path between every pair of distinct vertices of the graph.
- **Not-Connected Graph**: An undirected graph that is not connected is called disconnected

Example: G1 is the connected graph because for every pair of distinct vertices there is a path between them and G2 is the not-connected graph because there is no path between vertices a and d.

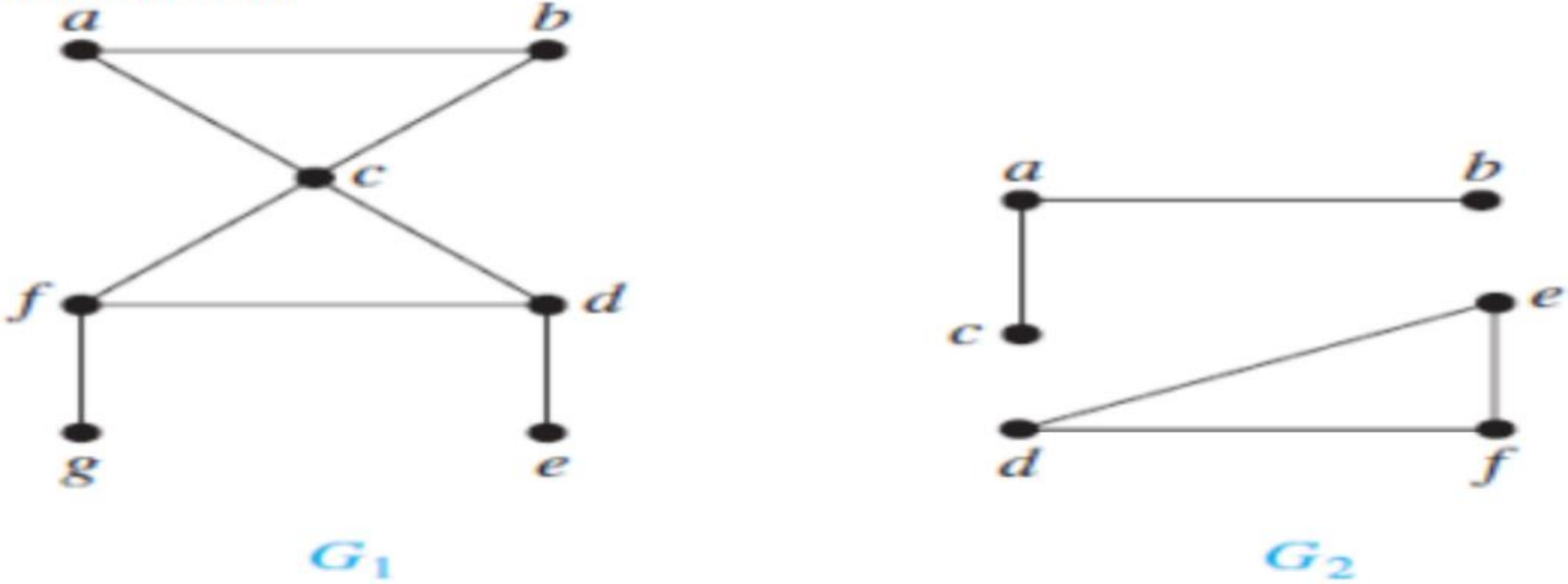
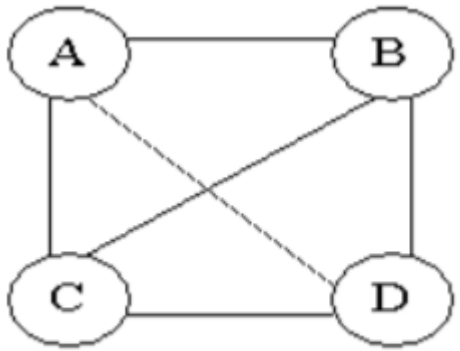


Fig: G1: Connected Graph and G2: Not-connected Graph

- **Complete Graph** :

- A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



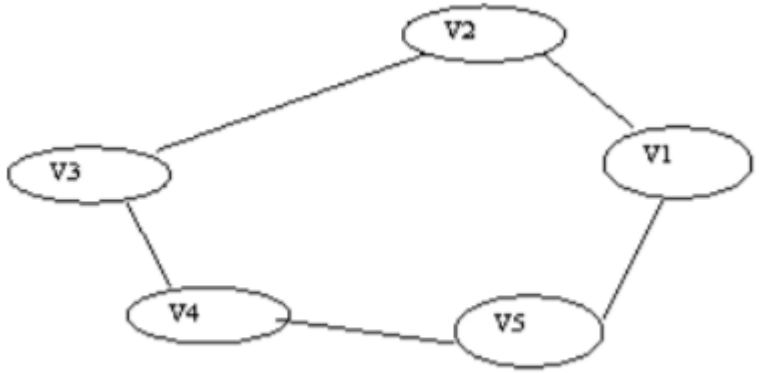
A complete graph.

- **Regular Graph :**

- It is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.

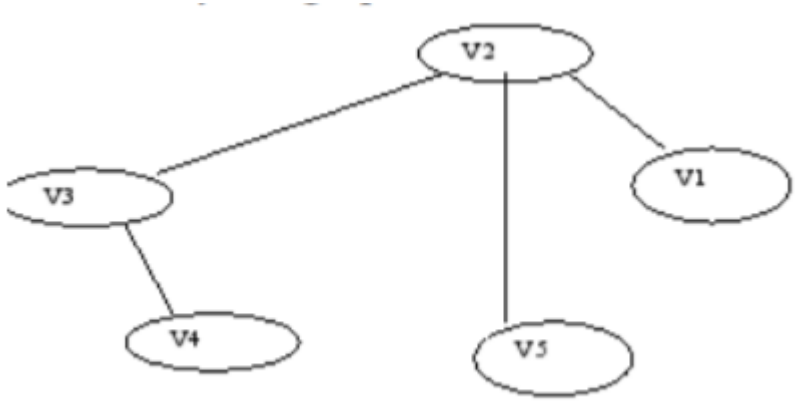
- **Cycle Graph :**

- A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle



A cycle graph

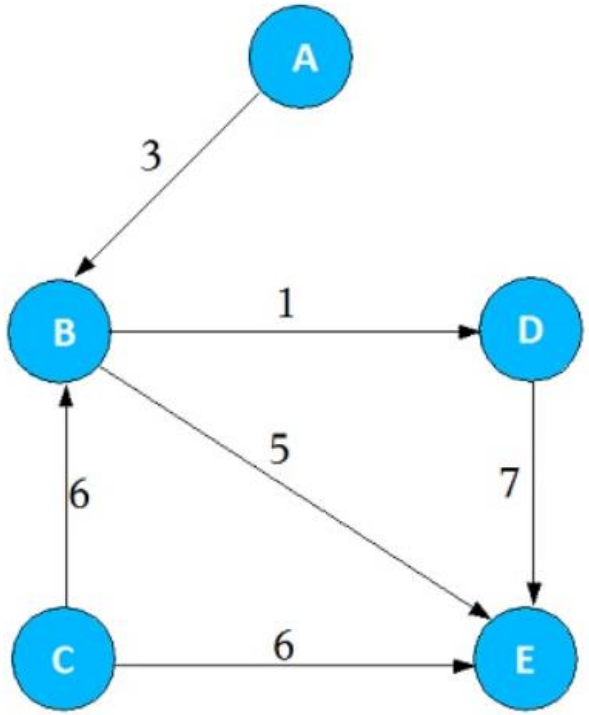
- **Acyclic Graph** : A graph without cycle is called acyclic graphs.



A acyclic graph

- **Weighted Graph**:

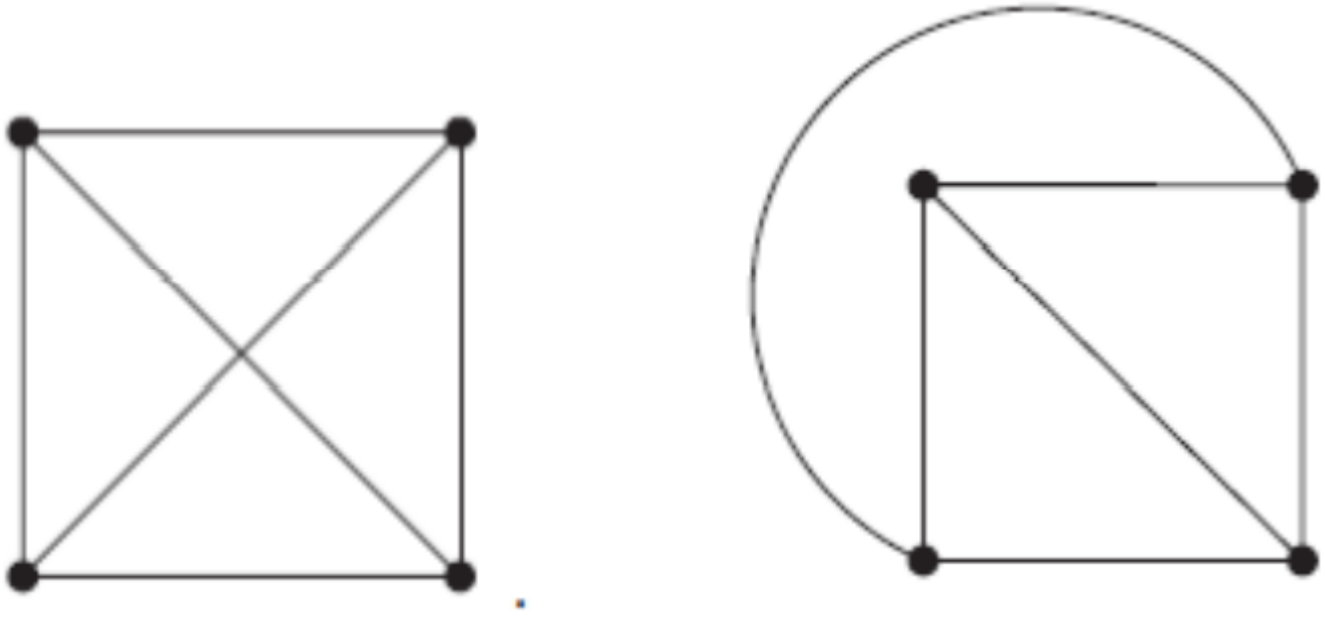
- Graphs that have a number assigned to each edge are called weighted graphs.
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge. Edge weights may represent distances, costs, etc.
- Example: In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Weighted Directed Graph

- **Planar Graph** :

- A graph is called planar if it can be drawn in the plane without any edges crossing, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other .



Graph as an ADT

- Following are basic primary operations of a Graph −
- **Add Vertex** − Adds a vertex to the graph.
- **Add Edge** − Adds an edge between the two vertices of the graph.
- **add Vertex** − adds a vertex of the graph.
- Delete edge
- Delete vertex

Graph Traversal

- Graph traversal is a technique used for searching a vertex in a graph.
- The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.
- There are two graph traversal techniques and they are as follows:

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

DFS(Depth First Search)

• DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal. We use the following steps to implement DFS traversal...

• Step 1 - Define a Stack of size total number of vertices in the graph.

• Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

• Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

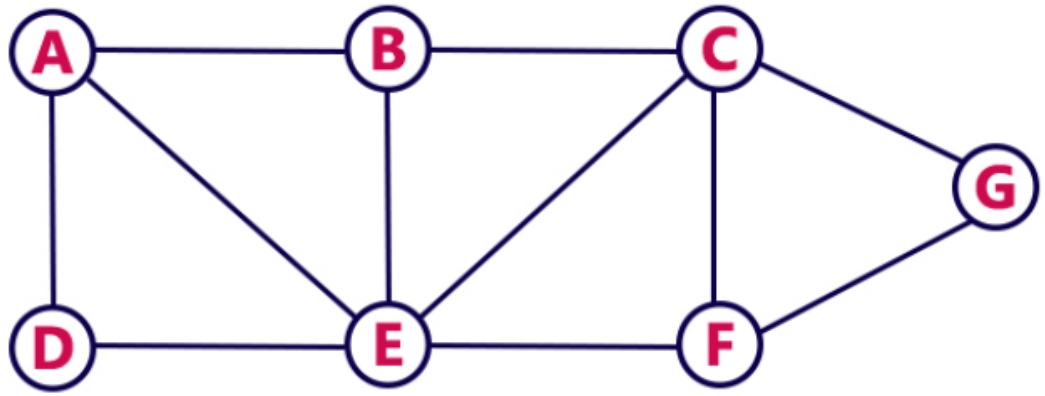
• Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

• Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

• Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

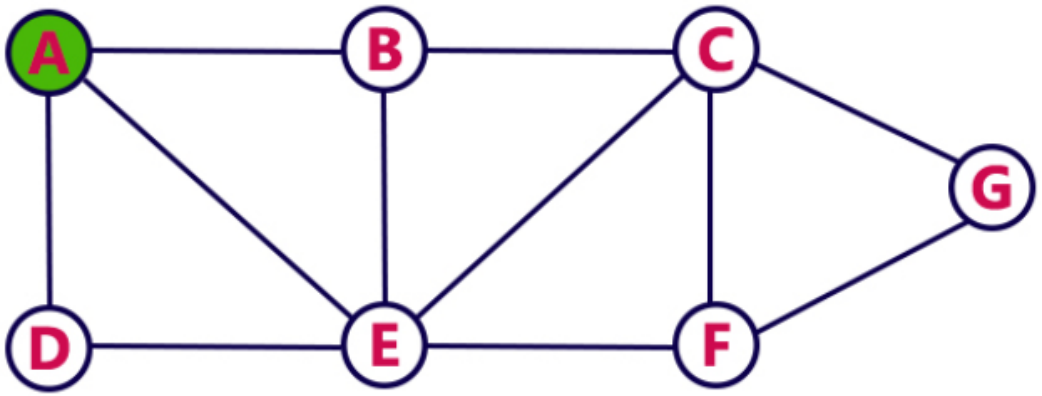
• Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

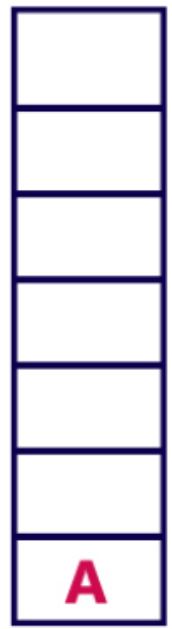
Consider the following example graph to perform DFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

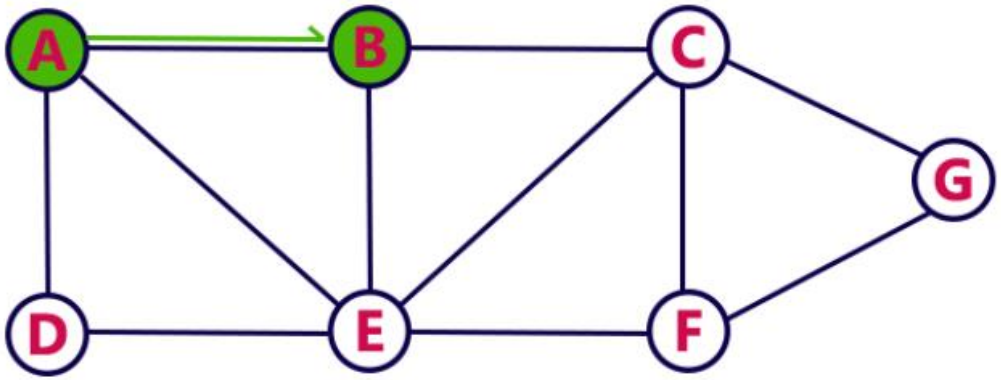


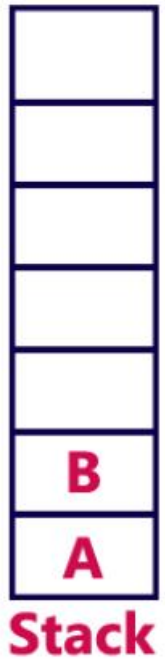


Stack

Step 2:

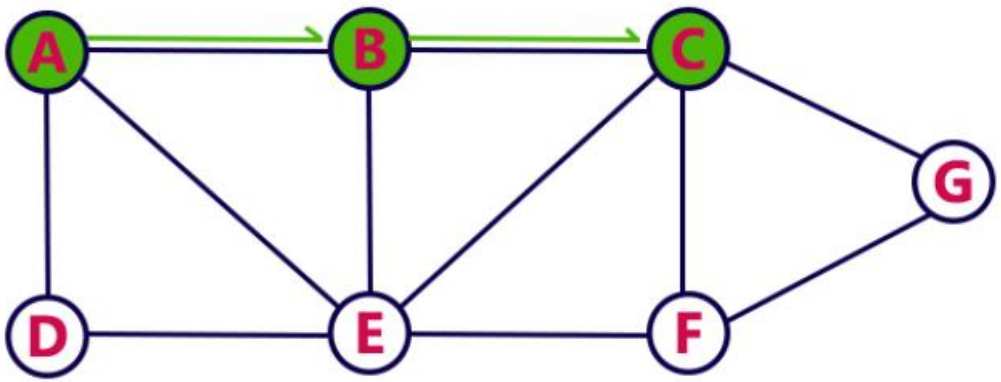
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

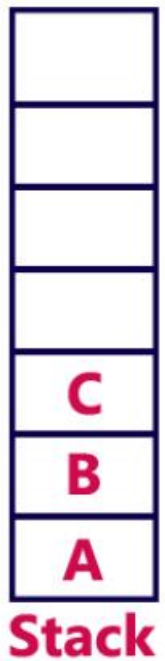




Step 3:

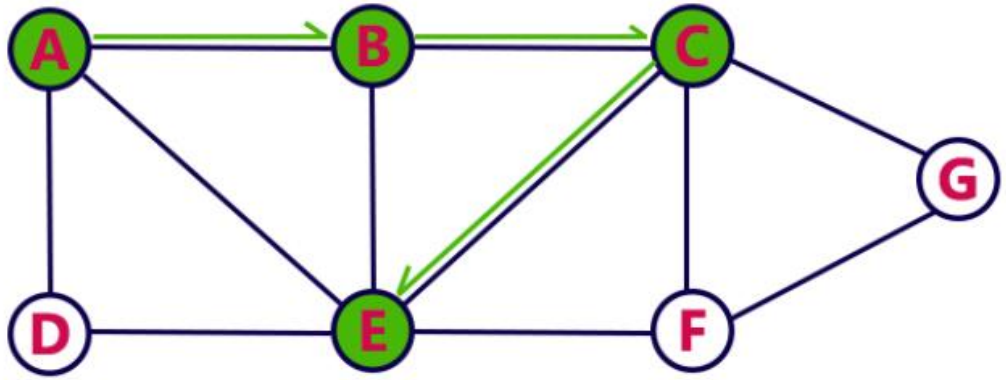
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

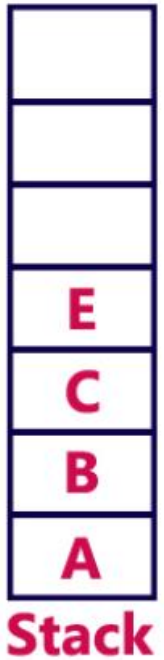




Step 4:

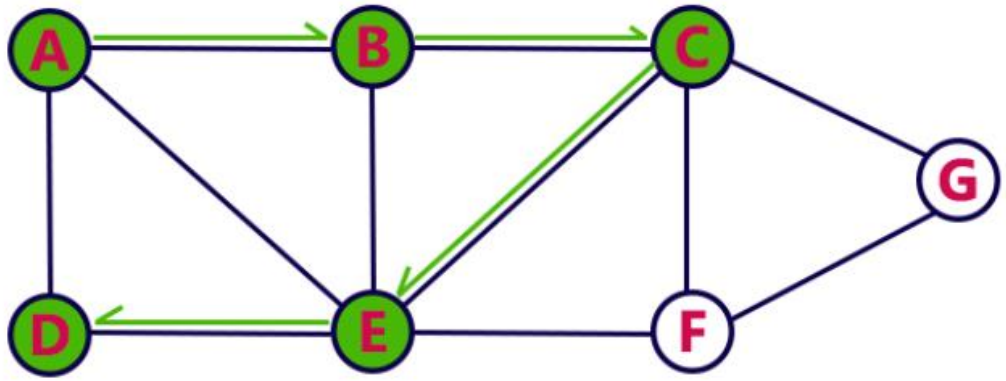
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

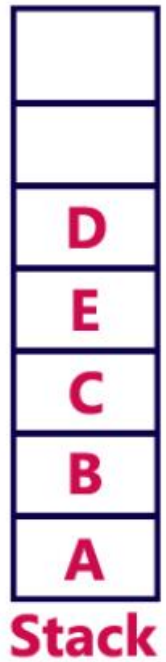




Step 5:

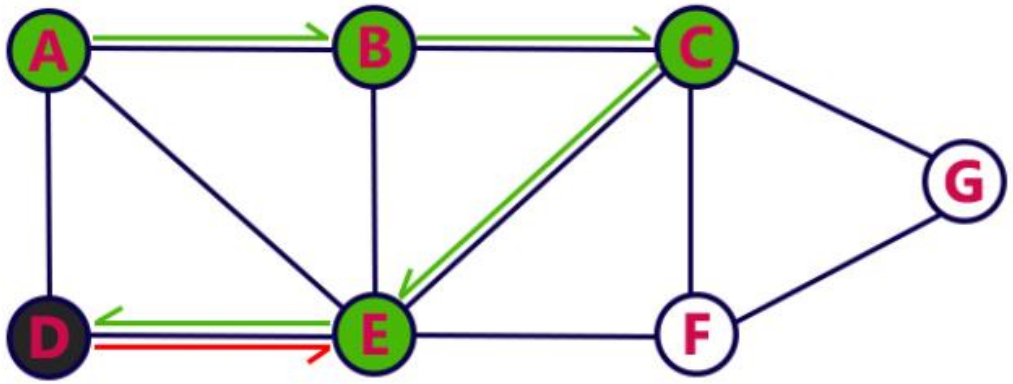
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

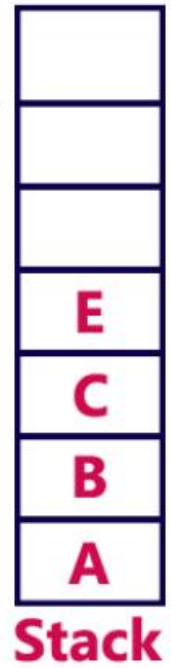




Step 6:

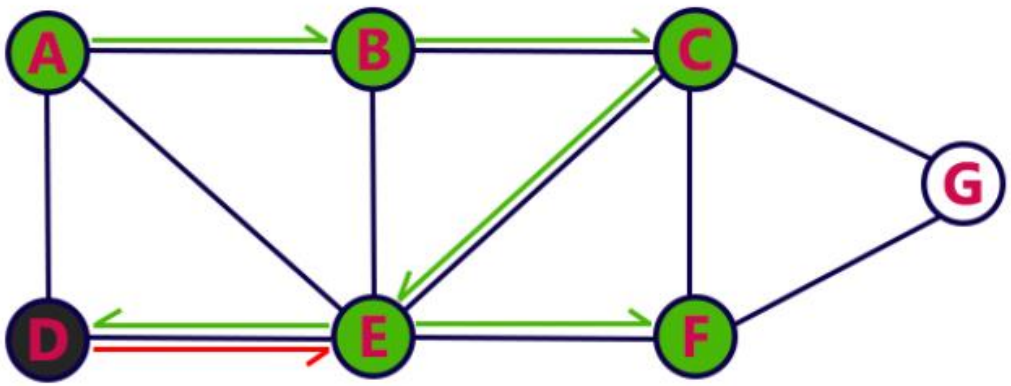
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

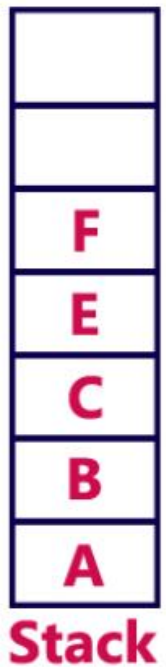




Step 7:

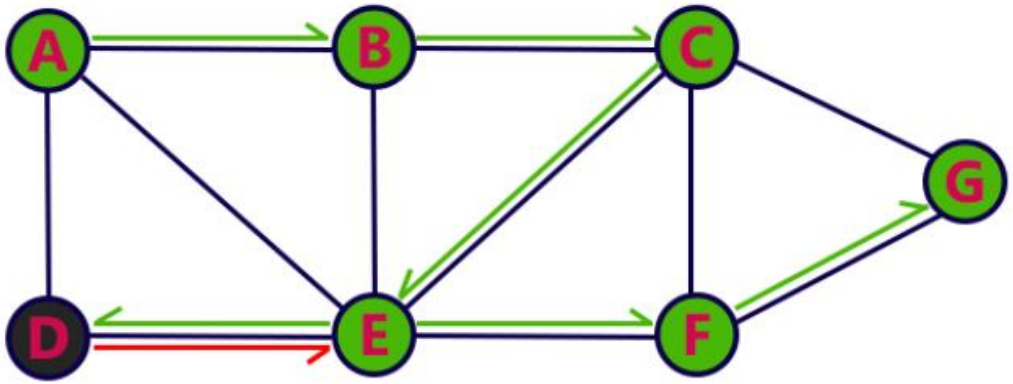
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

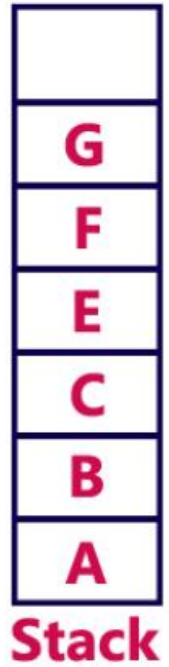




Step 8:

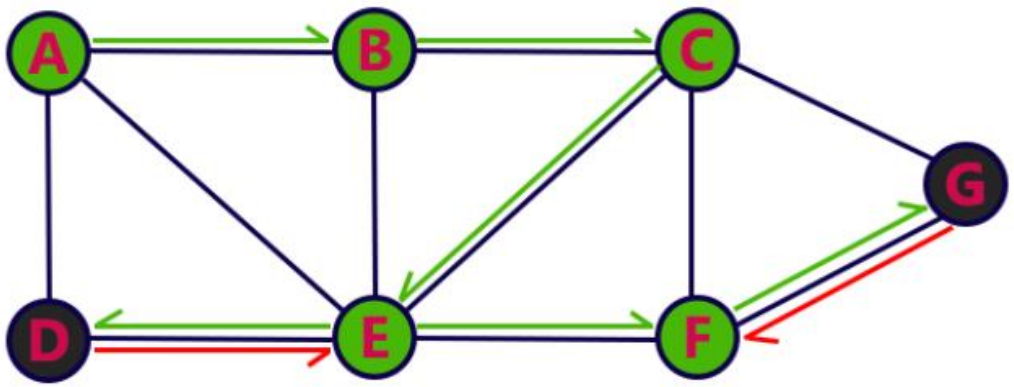
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

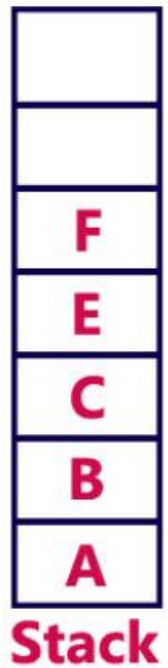




Step 9:

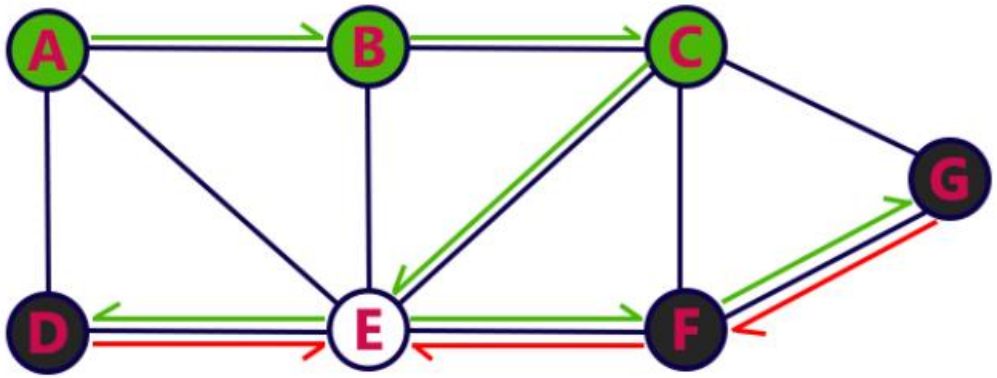
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

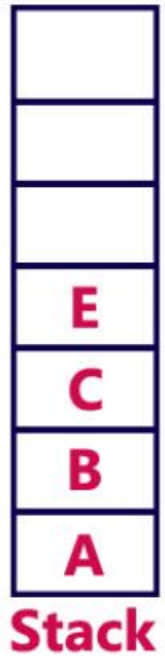




Step 10:

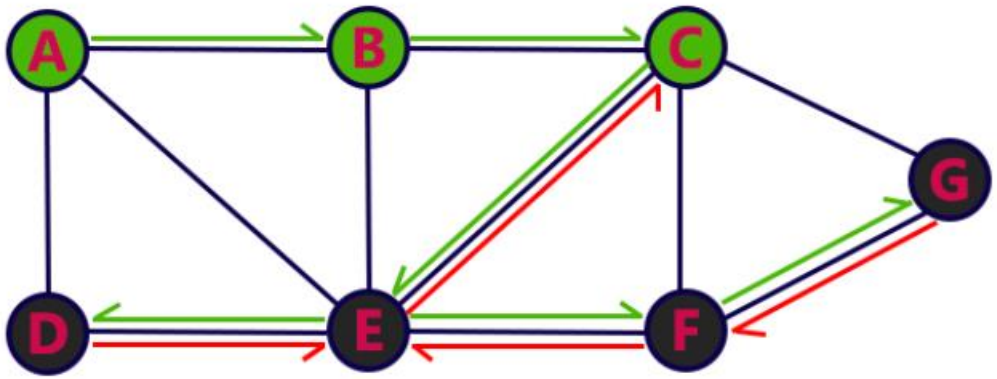
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

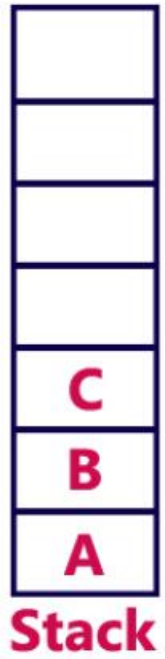




Step 11:

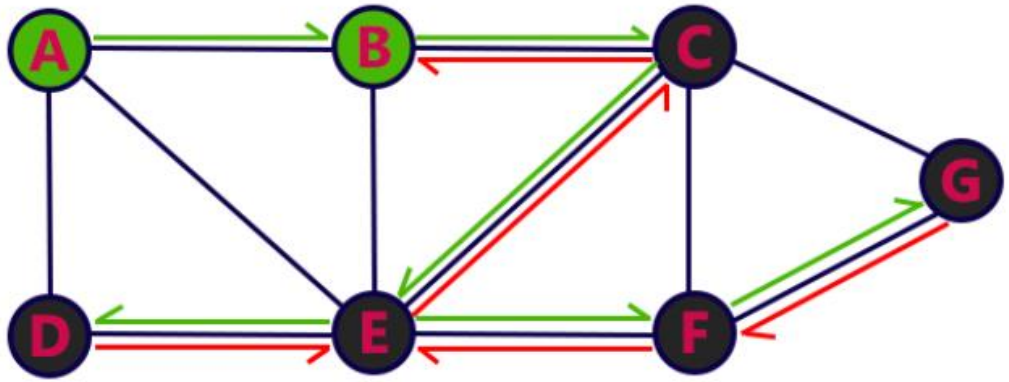
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

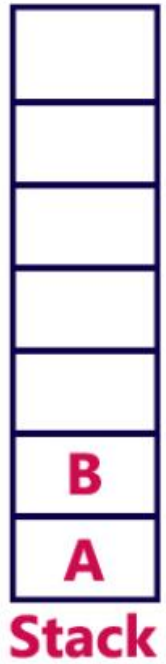




Step 12:

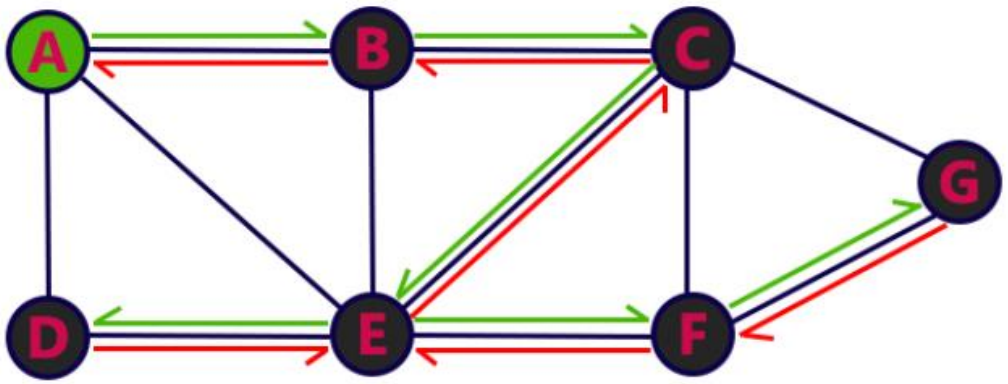
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

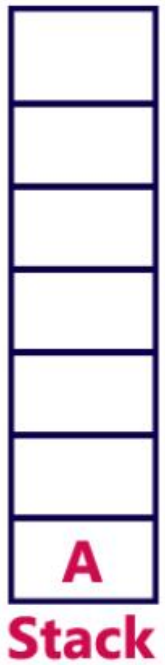




Step 13:

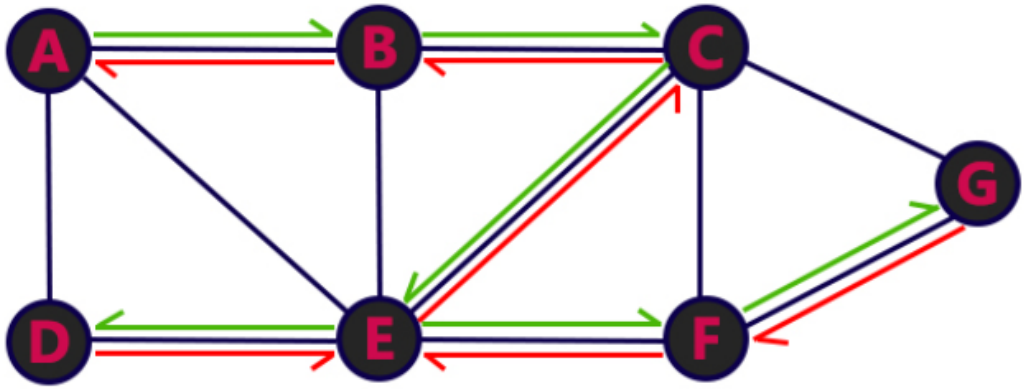
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



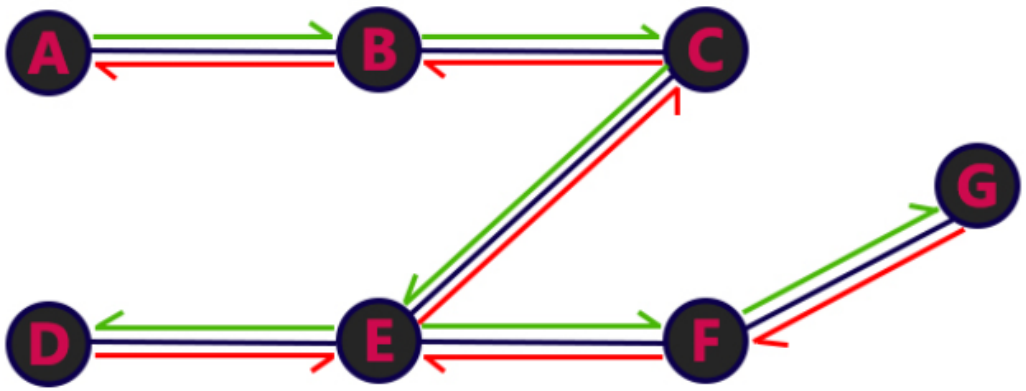


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



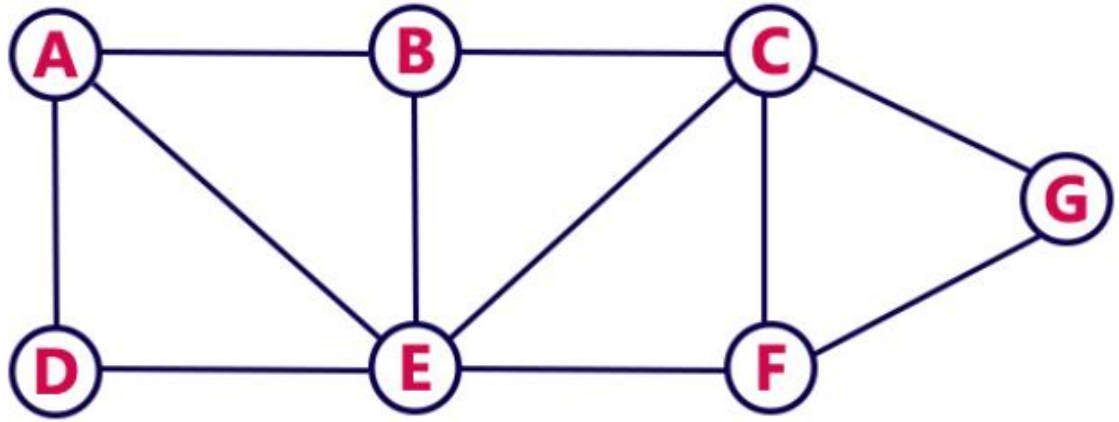
BFS (Breadth First Search)

- BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

  We use the following steps to implement BFS traversal...

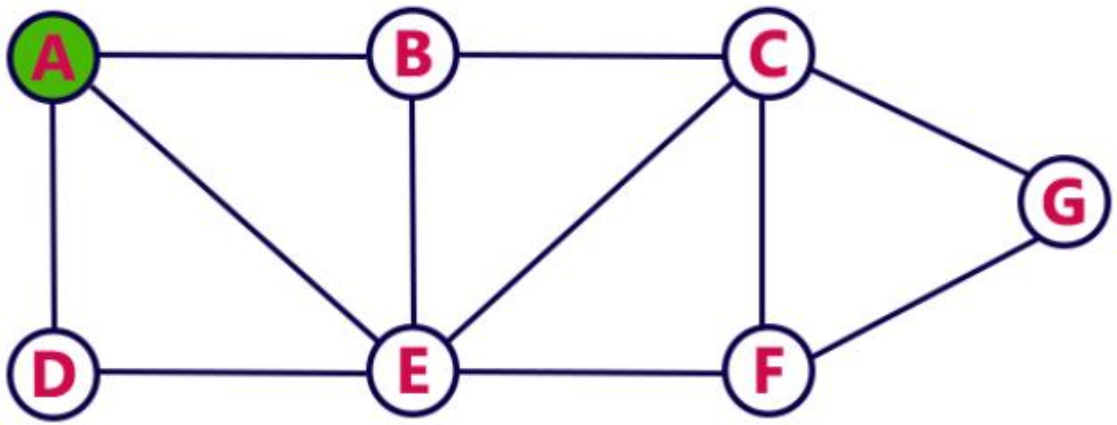
- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform BFS traversal



Step 1:

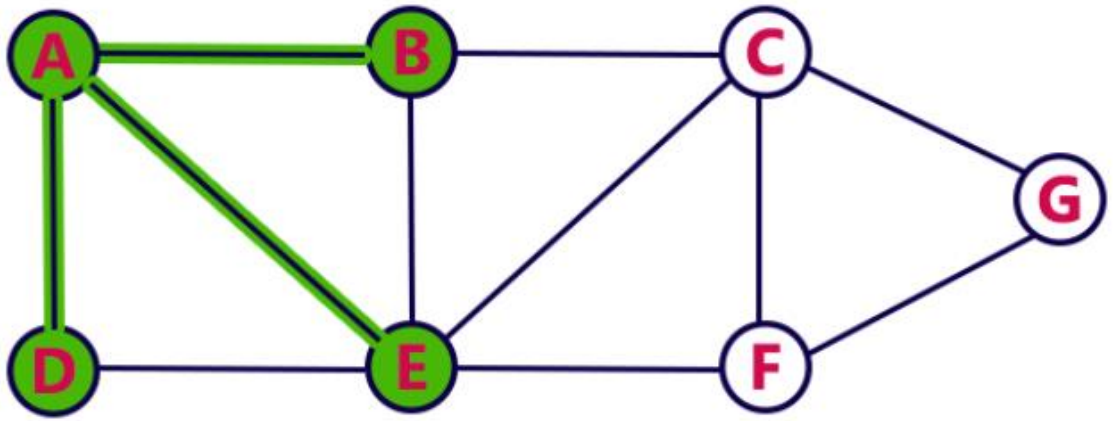
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



Queue

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

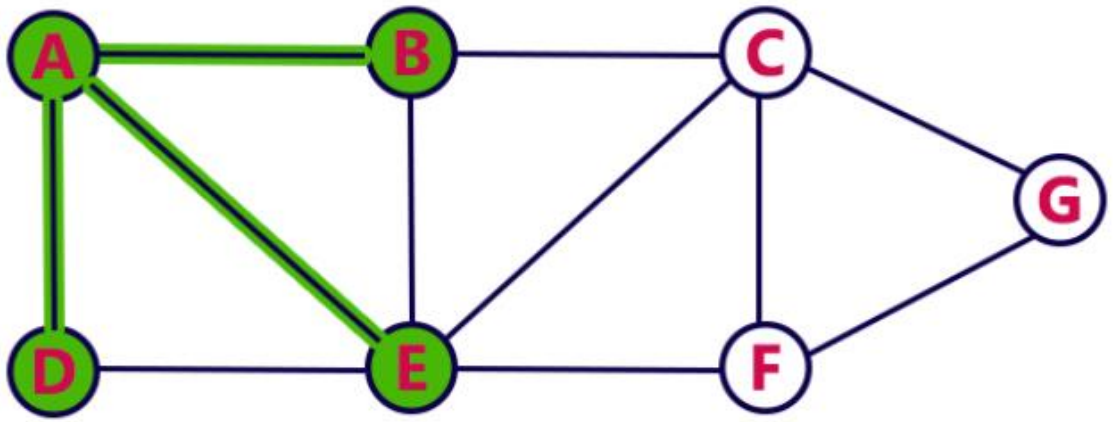


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

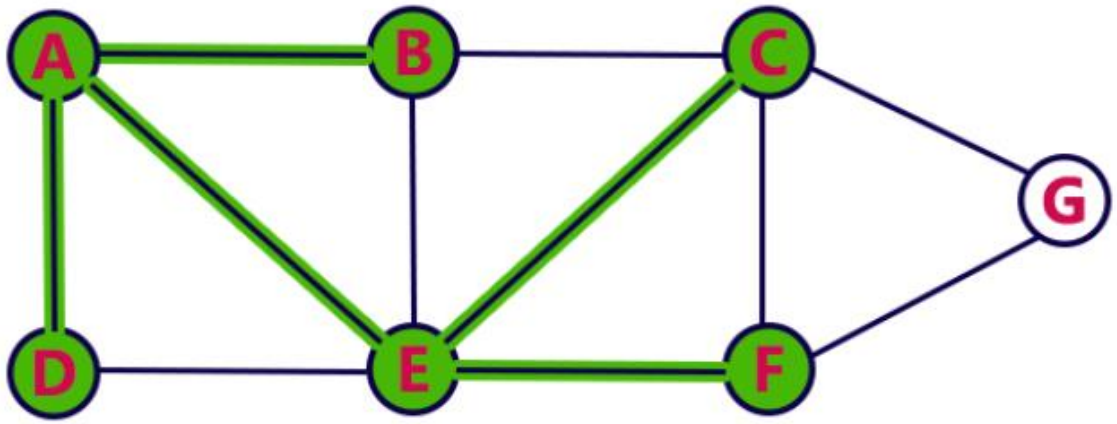


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

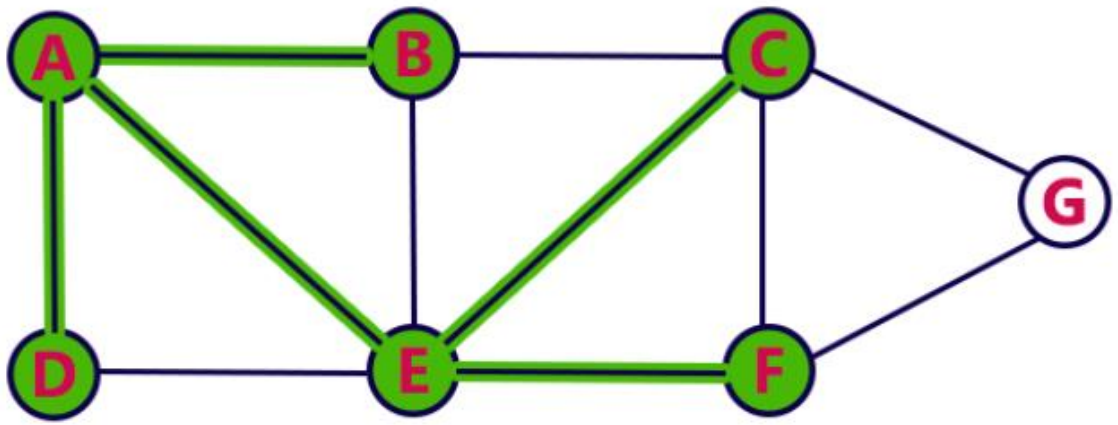


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

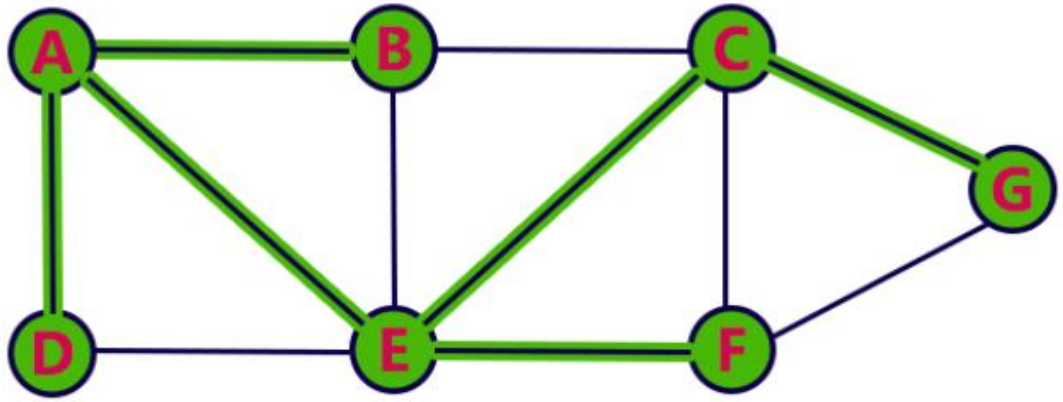


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

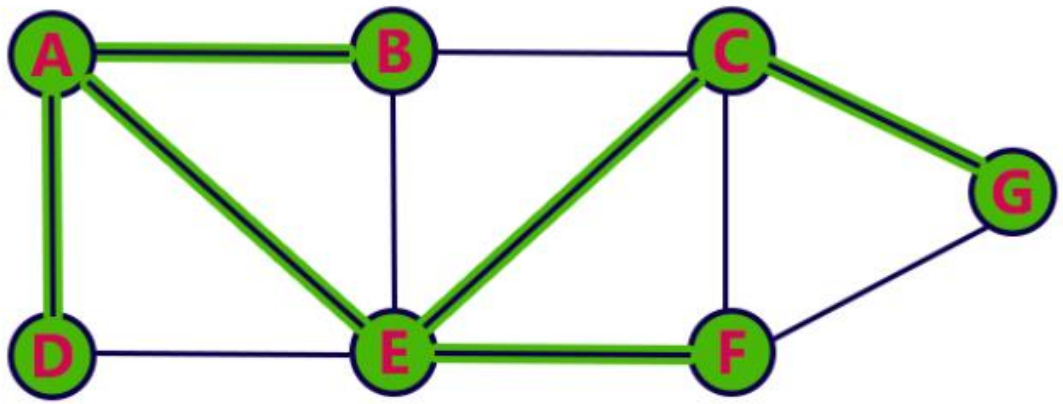


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

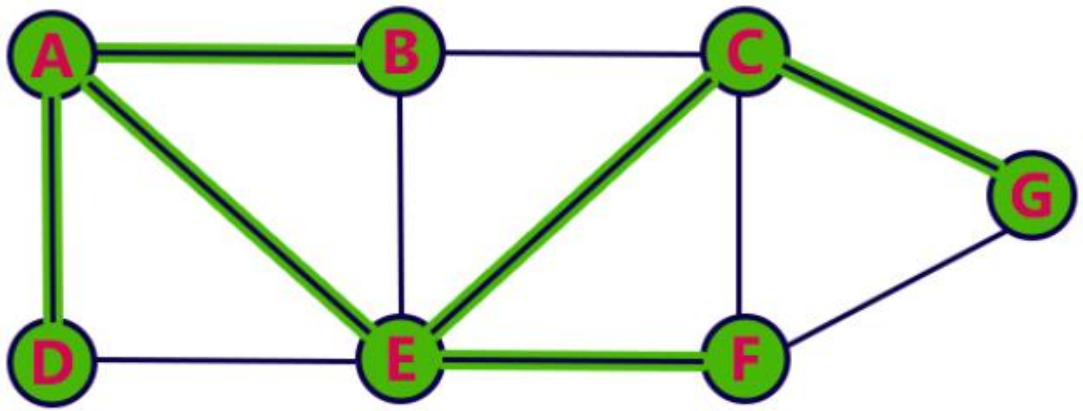


Queue

						G

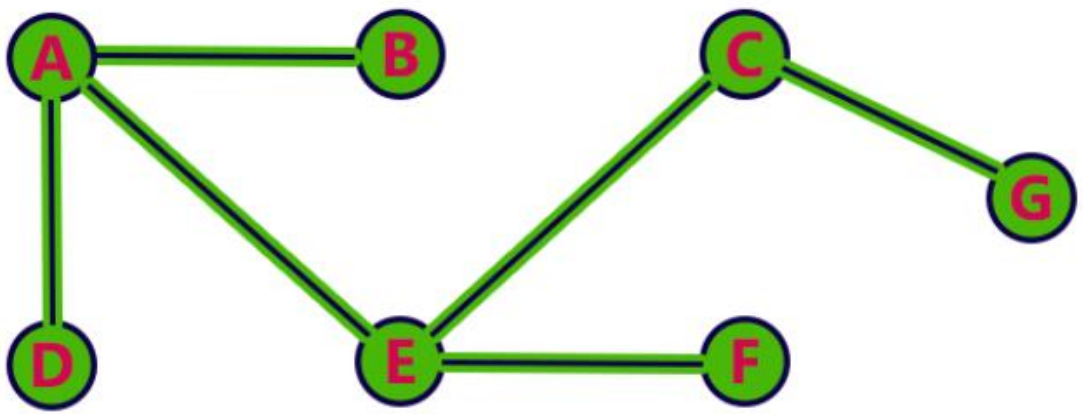
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Spanning Tree

- A spanning tree of an undirected graph G is tree that includes every vertex of G and is a subgraph of G.
- It covers all the vertices of G with minimum possible number of edges. Therefore, a spanning tree doesn't have cycles and it can’t be disconnected.

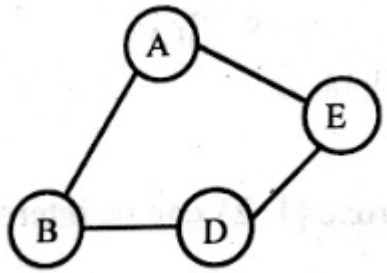


Fig: Graph **G**

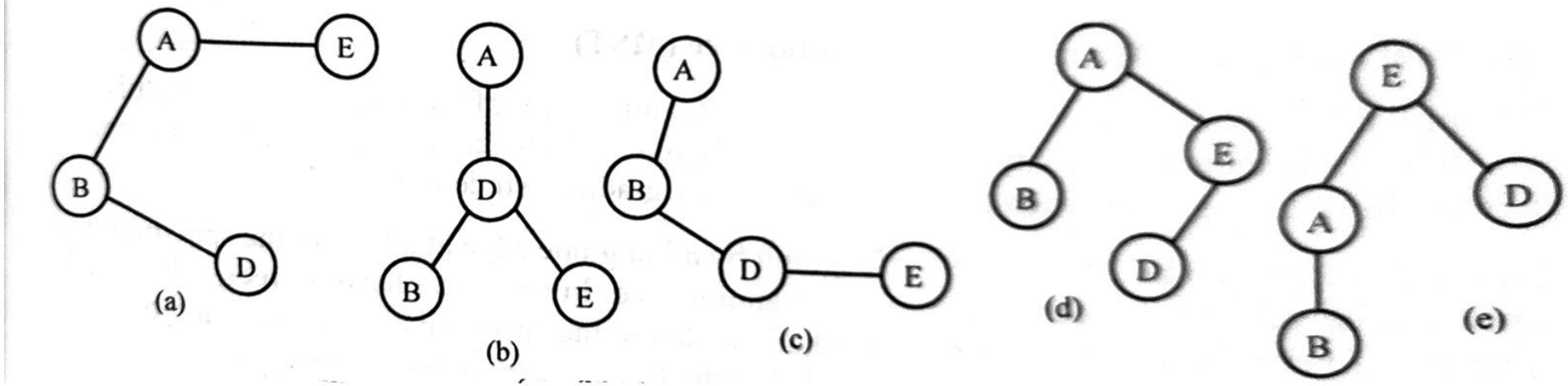


Fig: a, b, c, d, e spanning tree of graph G

Properties of Spanning tree

1. Connected Graph G can have more than one spanning tree.
2. All possible spanning trees of graph G have the same number of edges and vertices.
3. A spanning tree doesn't have any cycle.
4. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In above graph G, 4^{4-2} = 16 spanning trees are possible.
5. Spanning tree must include every vertex of graph G.
6. A spanning tree can’t be disconnected. That means it is minimally connected.
7. A spanning tree has n vertices and n-1 edges.

Minimum Spanning Tree

- **A minimum spanning tree** is a tree constructed from a weighted, undirected graph, so it:
  - ✓Connects all nodes (also referred to as vertices)
  - ✓Has no cycles
  - ✓Has the smallest possible sum of edge weights

Minimum Spanning Tree

- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

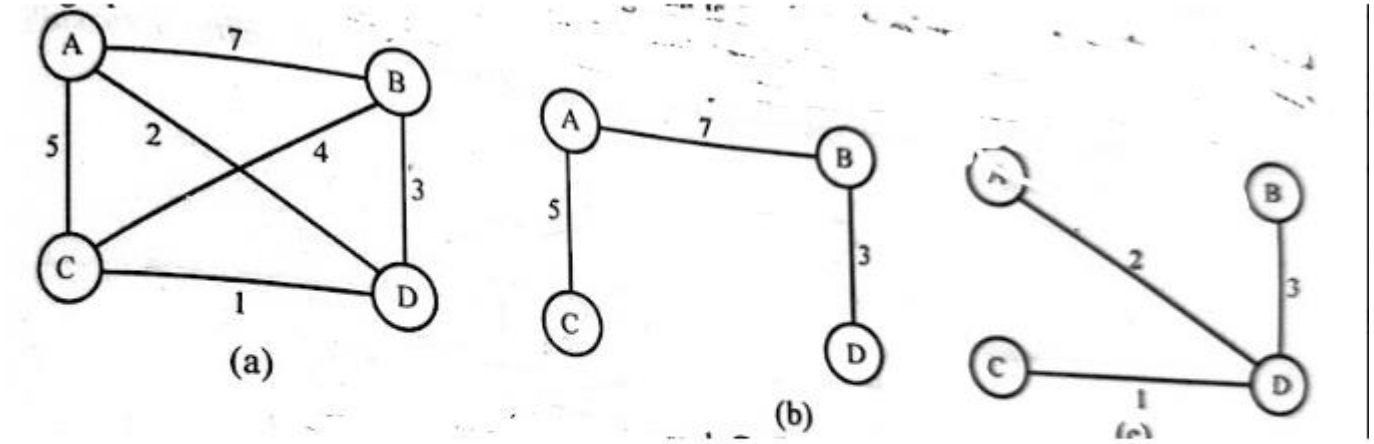
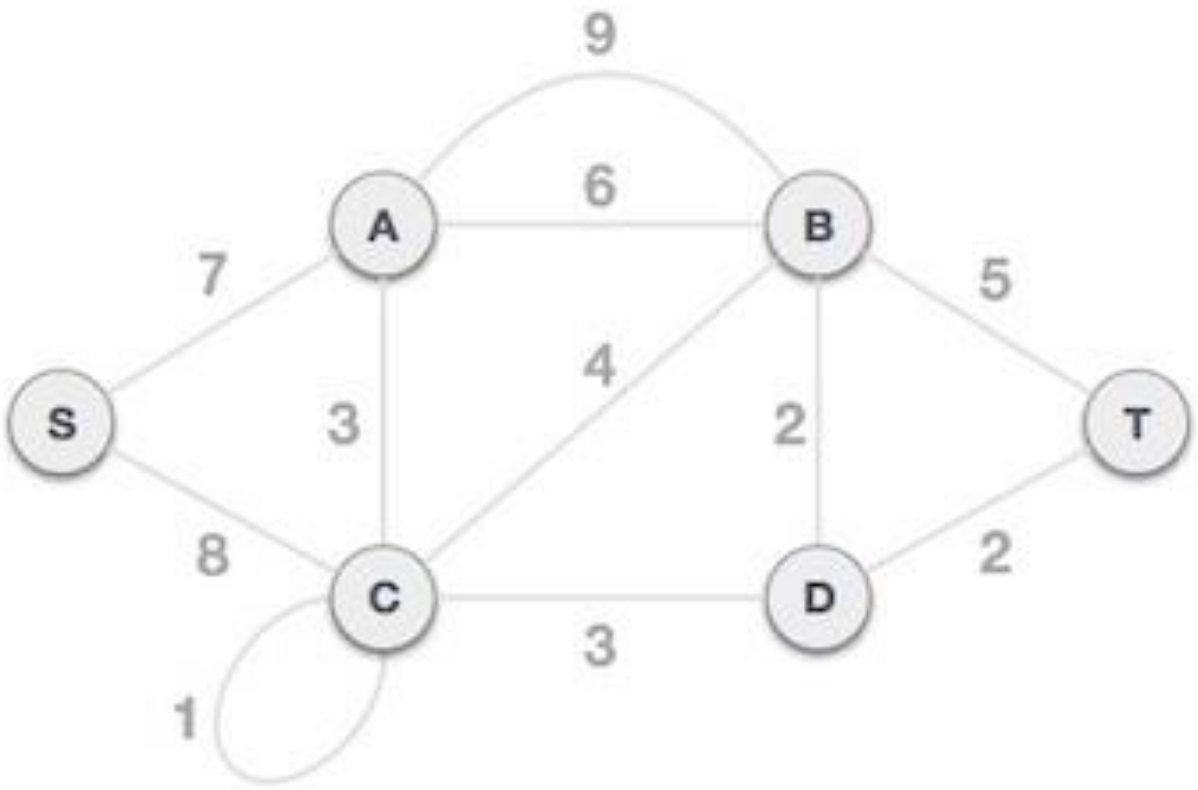


Fig: (a) Simple graph G (b) Spanning tree of G and (c) Minimum Spanning tree

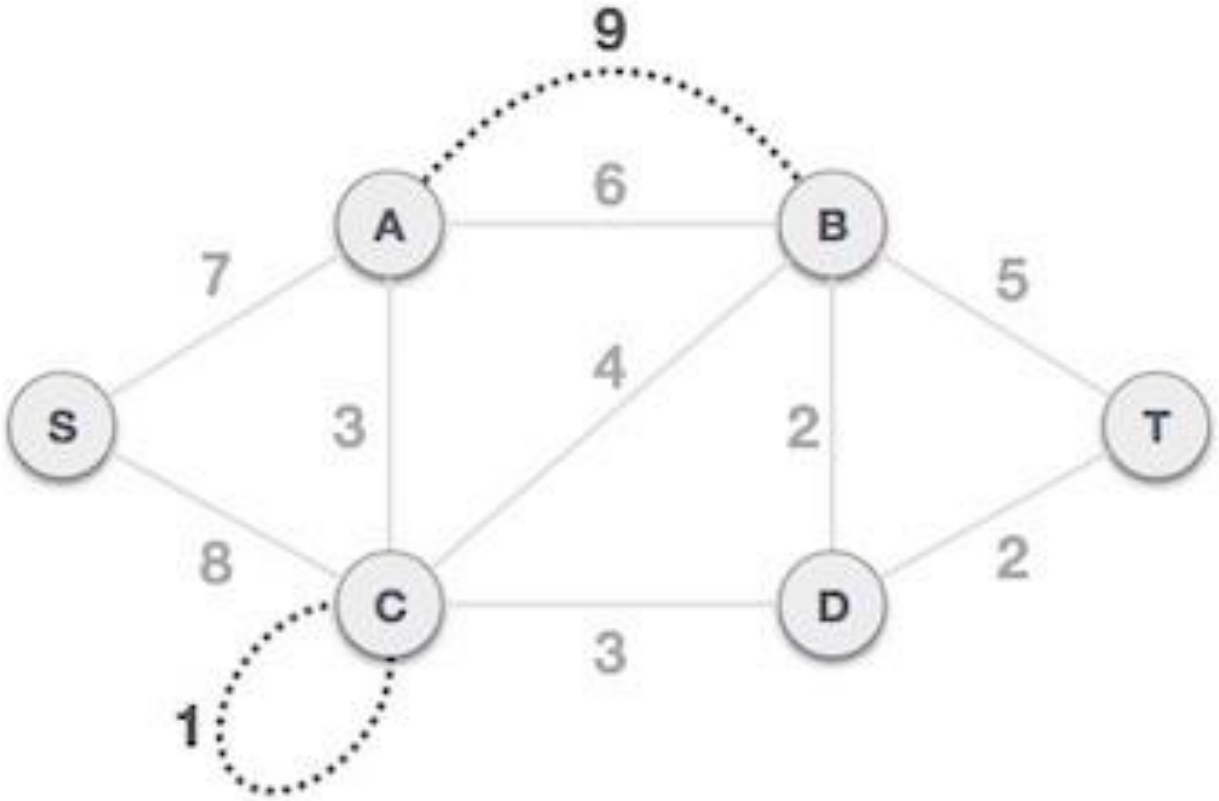
Kruskal's Algorithm

- Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree.
- It follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.
- **Algorithm**:
  - sort all edges by their weight in the ascending order
  - pick the edge with the smallest weight and try to add it to the tree
    - if it forms a cycle, skip that edge
  - repeat these steps until you have a connected tree that covers all nodes

Kruskal's Algorithm

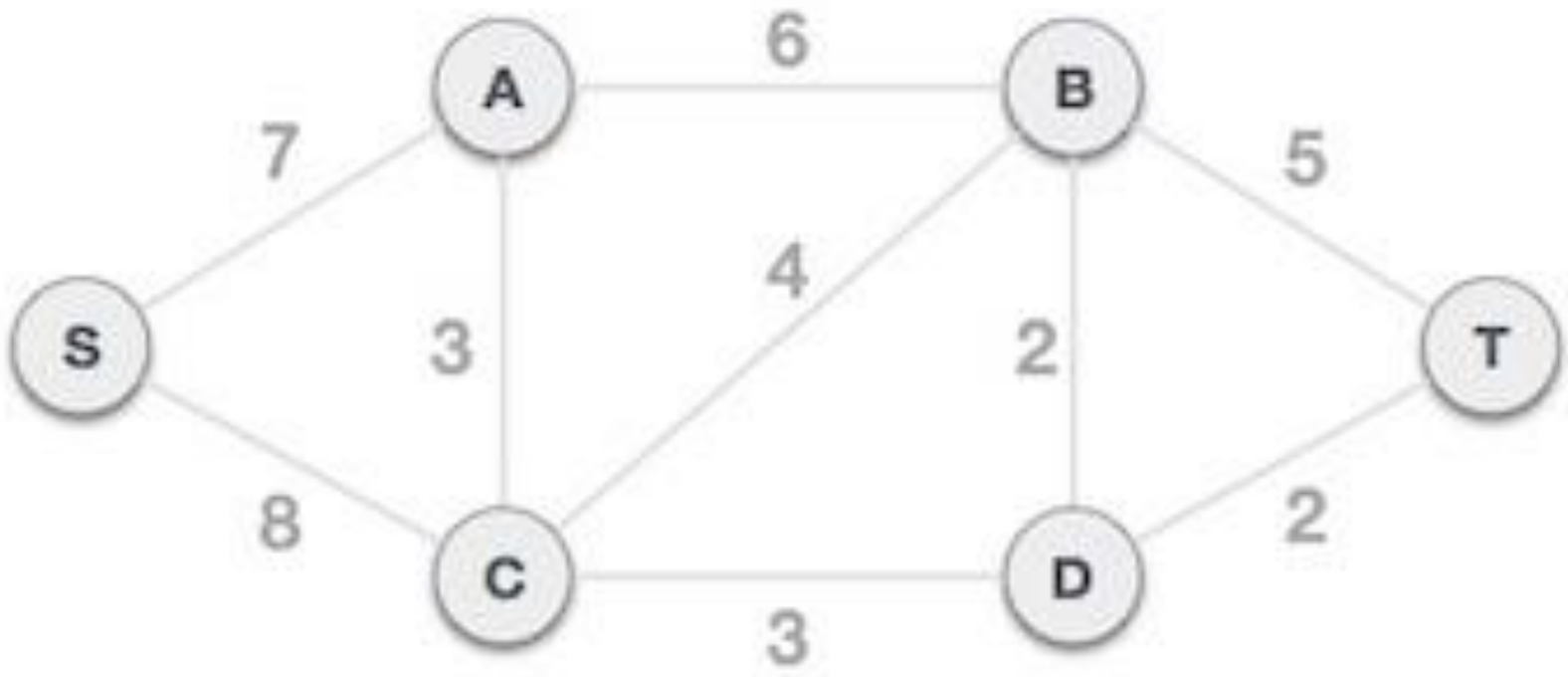


Step 1: Remove all loops and parallel edges



- In case of parallel edges, keep the one which has the least cost associated and remove all others.

Step 1: Remove all loops and parallel edges



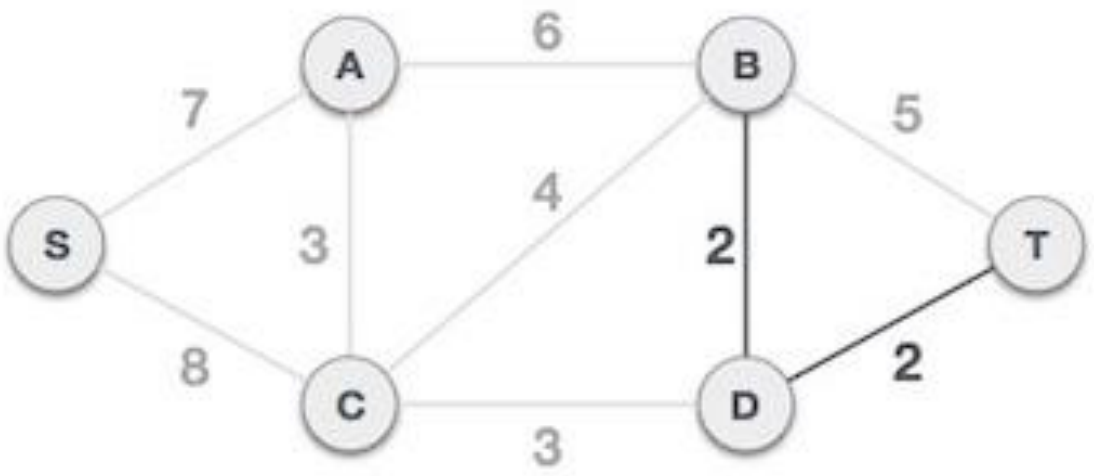
Step 2: Arrange all edges in their increasing order of weight

- The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

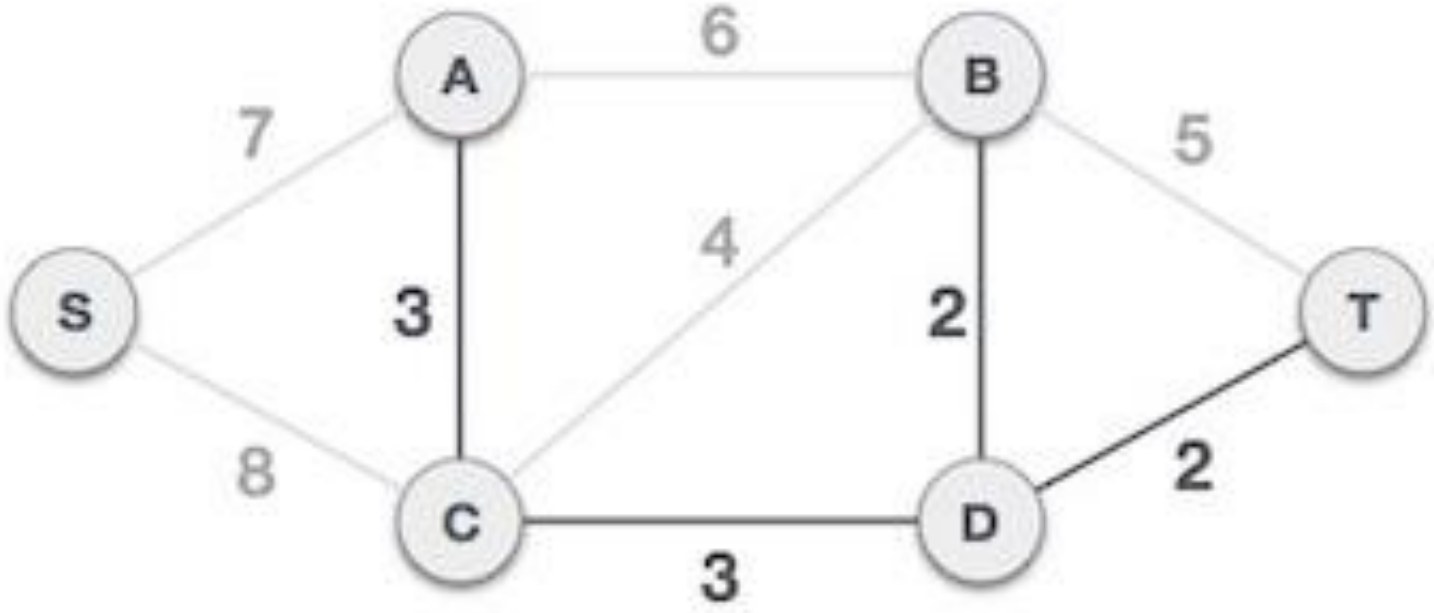
Step 3 - Add the edge which has the least weightage

- Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



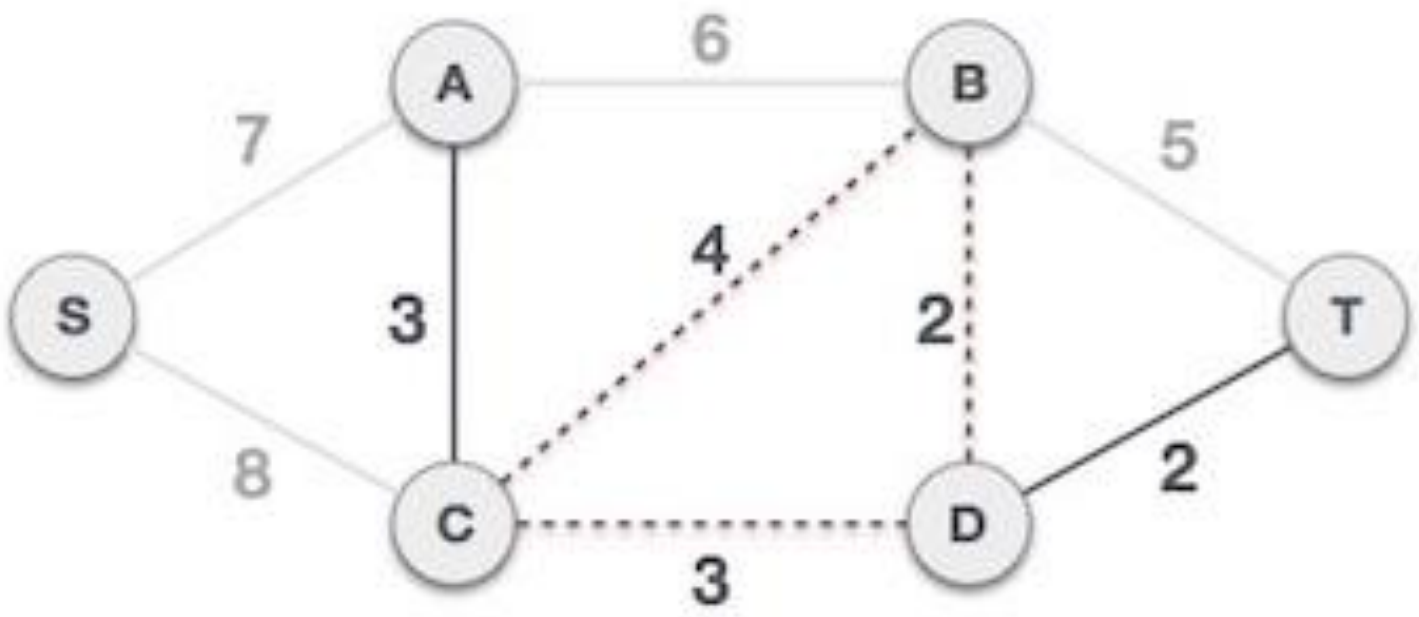
Step 3 - Add the edge which has the least weightage

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8



Step 3 - Add the edge which has the least weightage

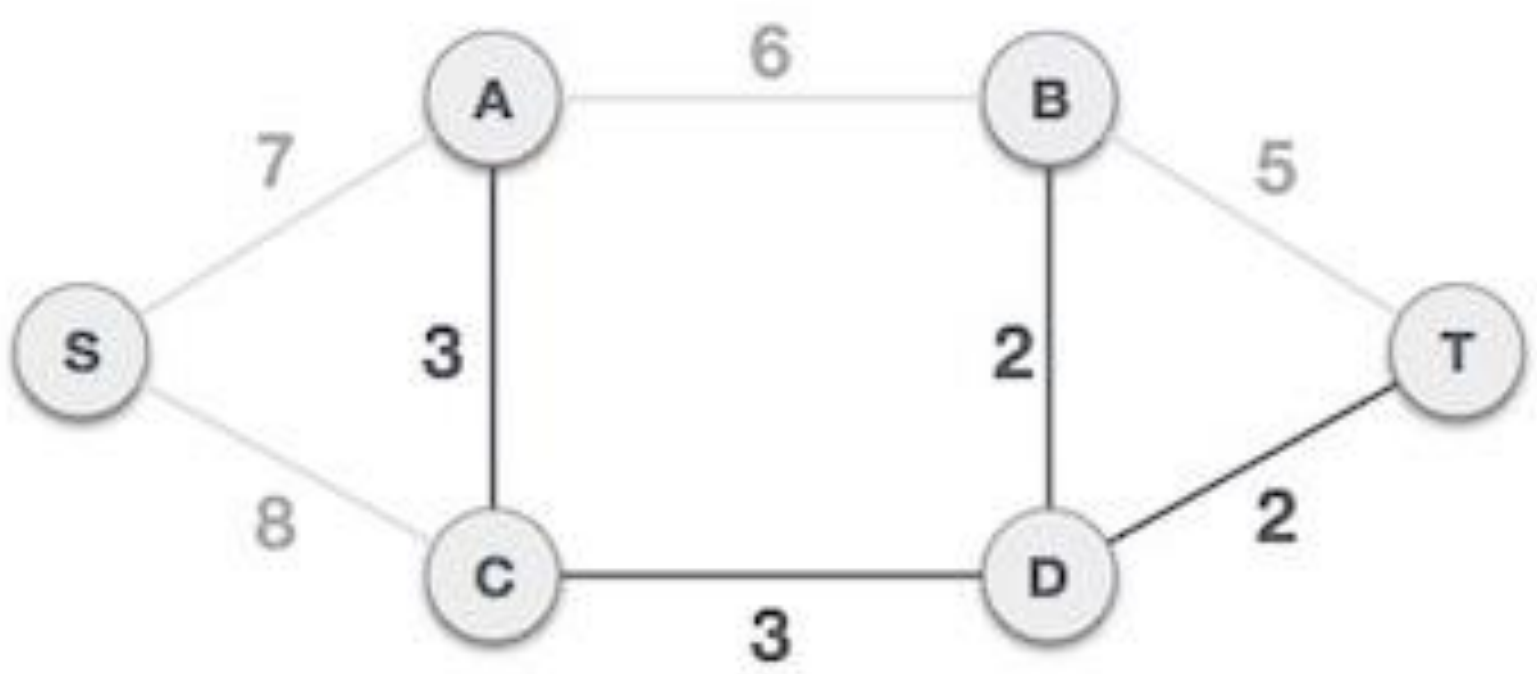
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8



Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

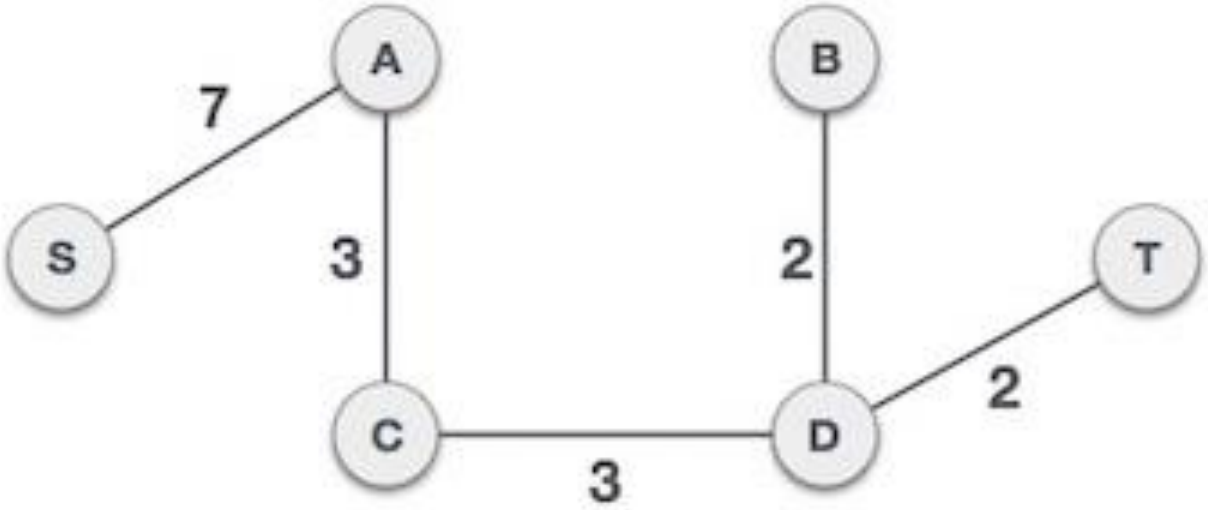
Step 3 - Add the edge which has the least weightage

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

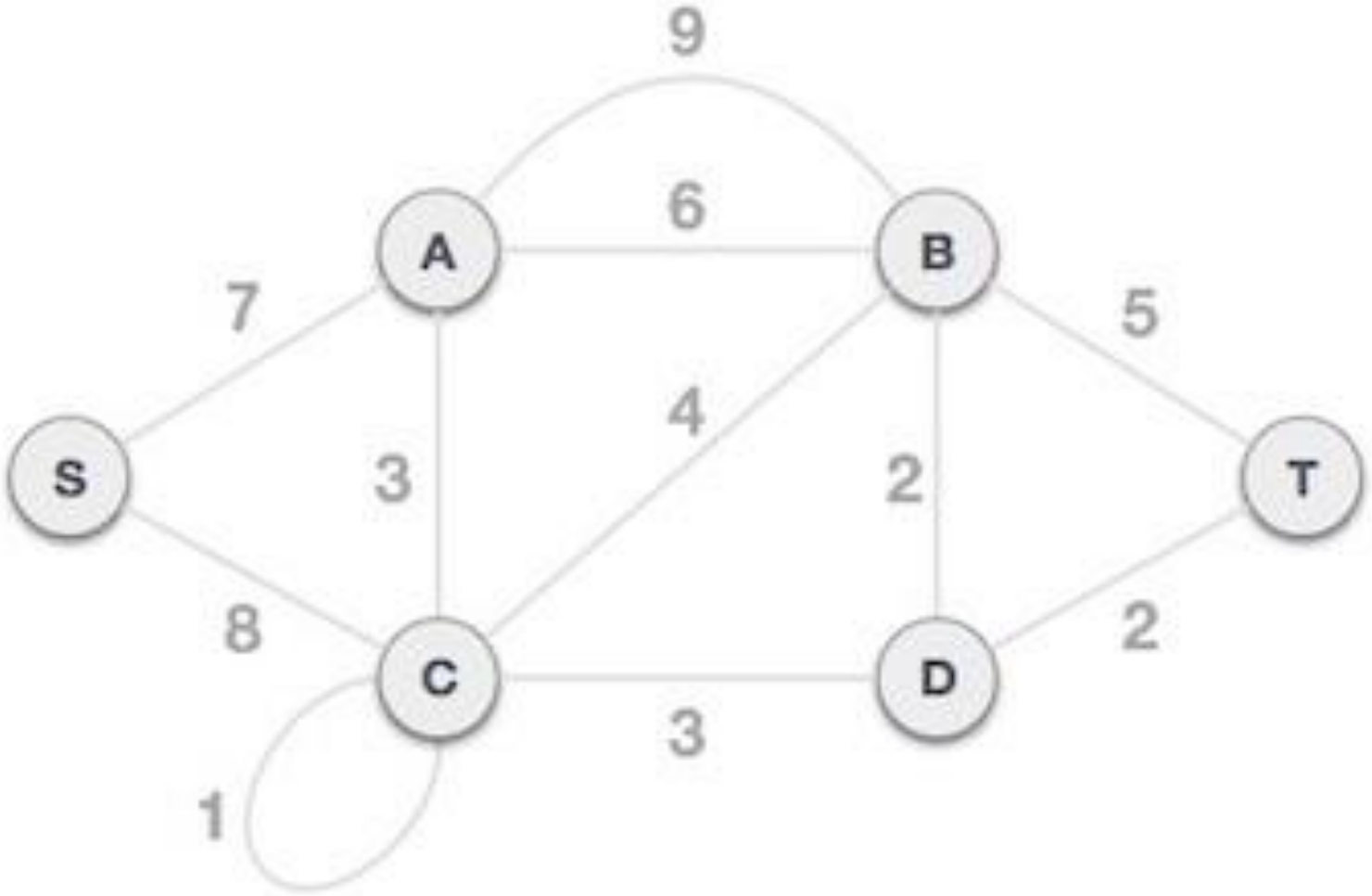


By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

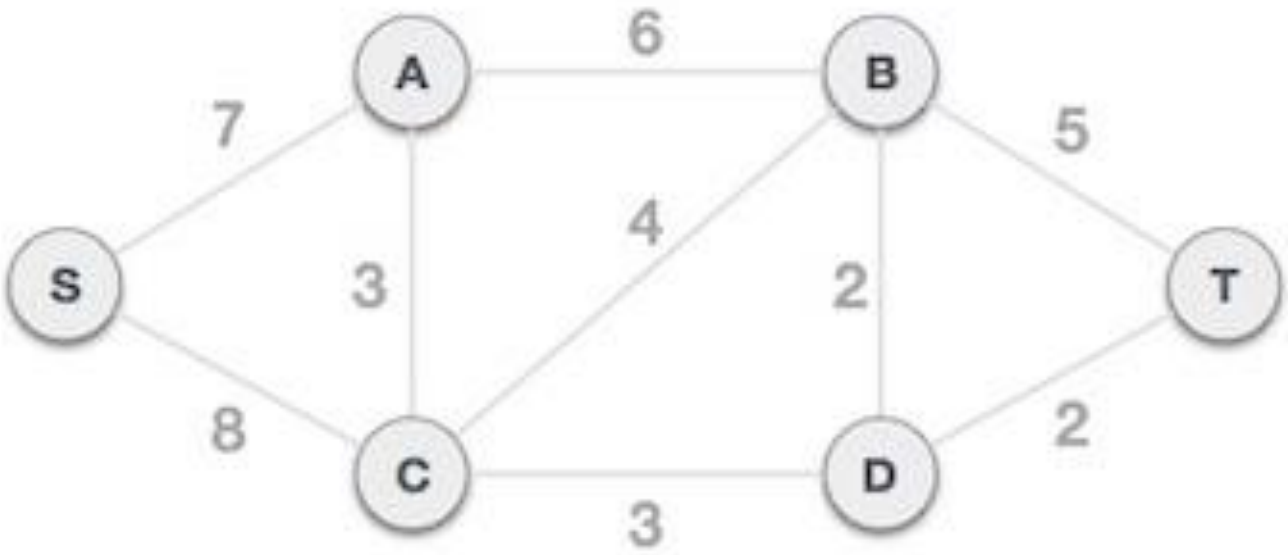
Prim's Spanning Tree Algorithm

- Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.
- Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.
- To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −

Prim's Spanning Tree Algorithm

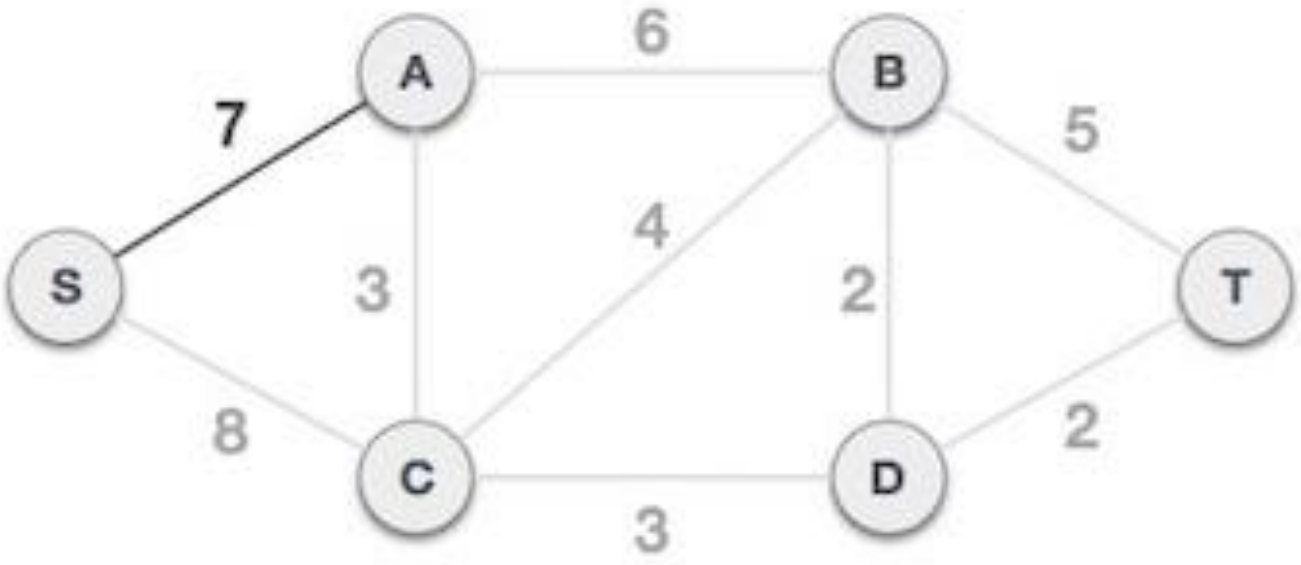


Prim's Spanning Tree Algorithm



Step1: Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all other

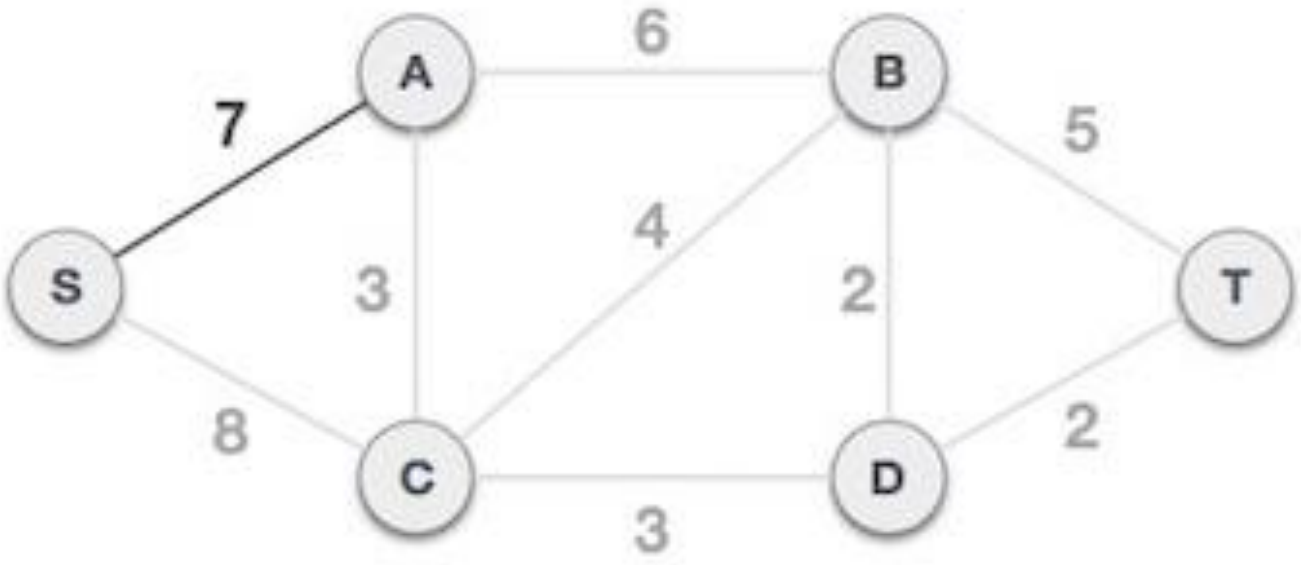
Prim's Spanning Tree Algorithm



Step 2 - Choose any arbitrary node as root node

In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

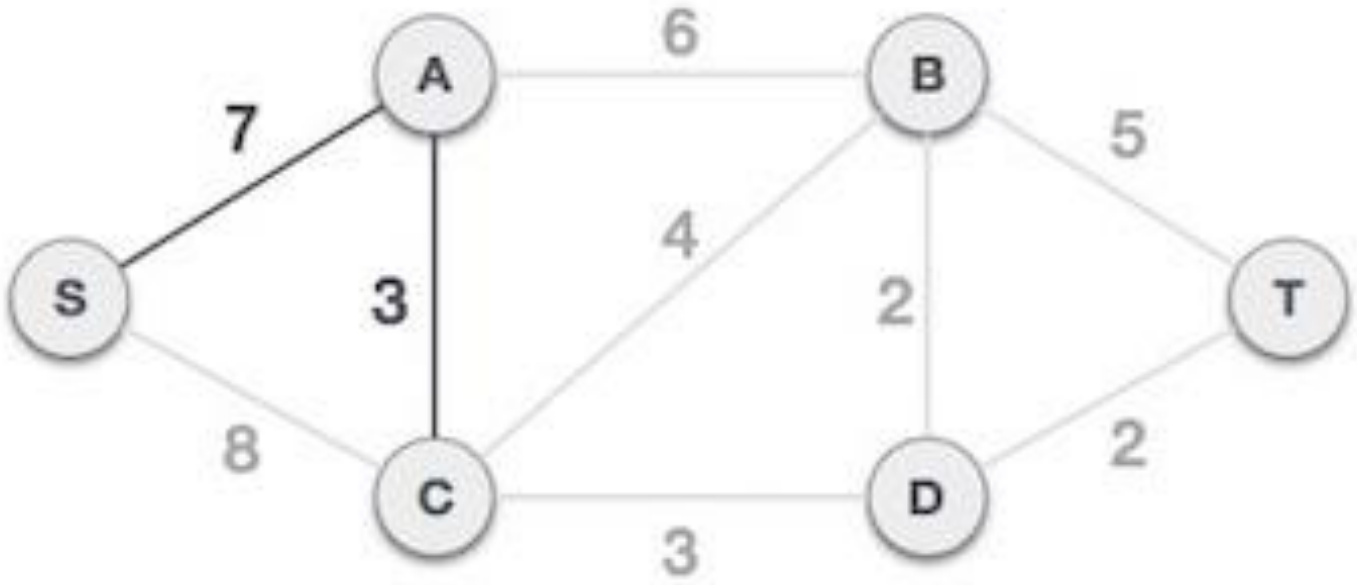
Prim's Spanning Tree Algorithm



Step 2 - Choose any arbitrary node as root node

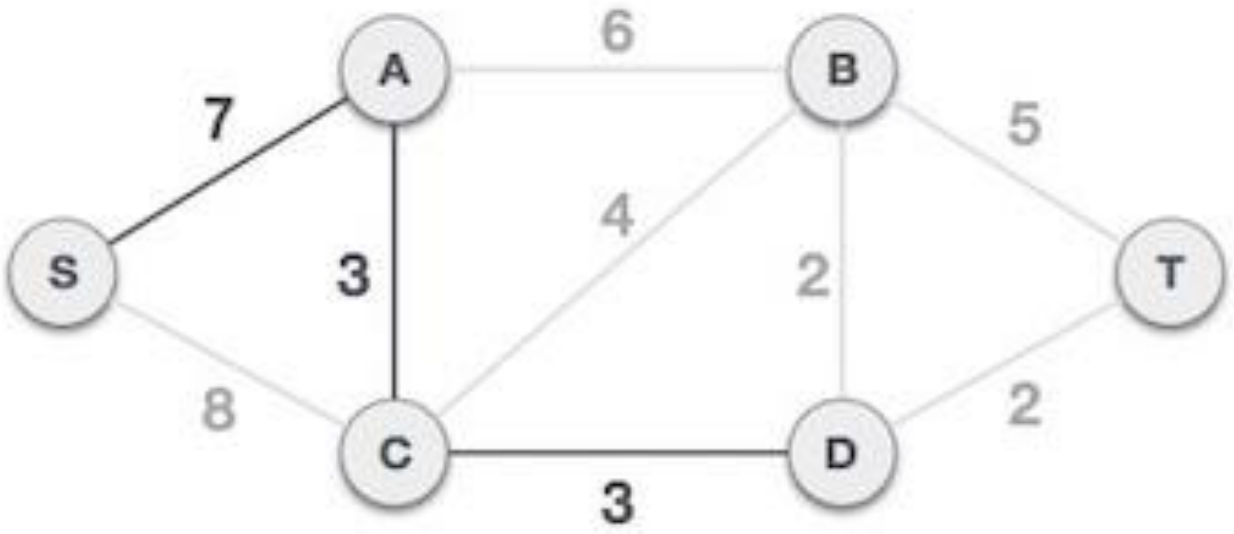
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Prim's Spanning Tree Algorithm

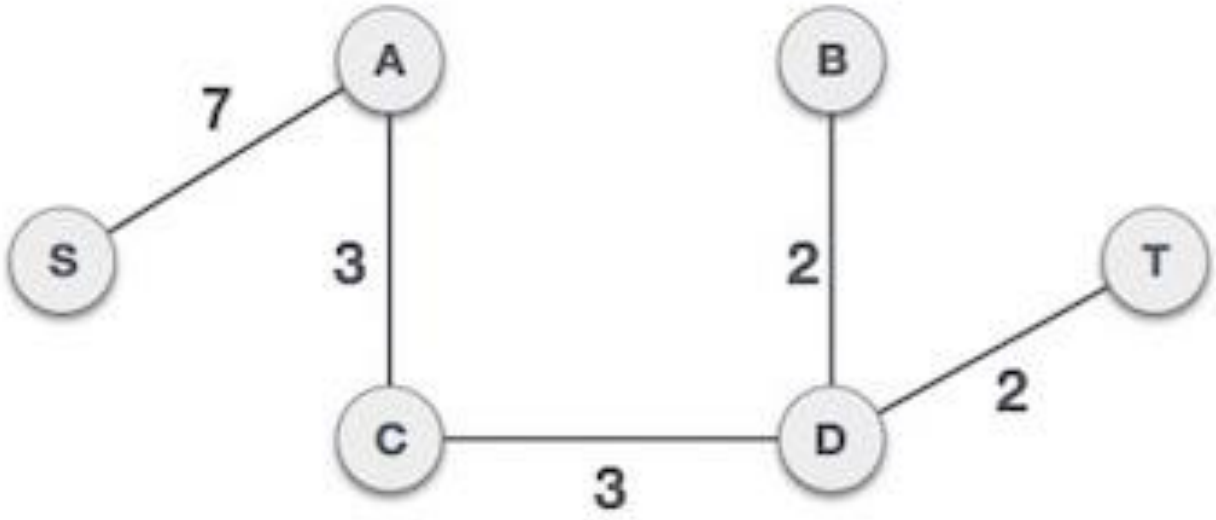


Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree i.e, S-7-A-3-C.

Prim's Spanning Tree Algorithm



Prim's Spanning Tree Algorithm



Dijkstra’s Algorithm

- **find the shortest path from a node (called the "source node") to all other nodes in the graph**, producing a shortest-path tree.
- This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, specially in domains that require modeling networks.

Implementation of Dijkstra Algorithm

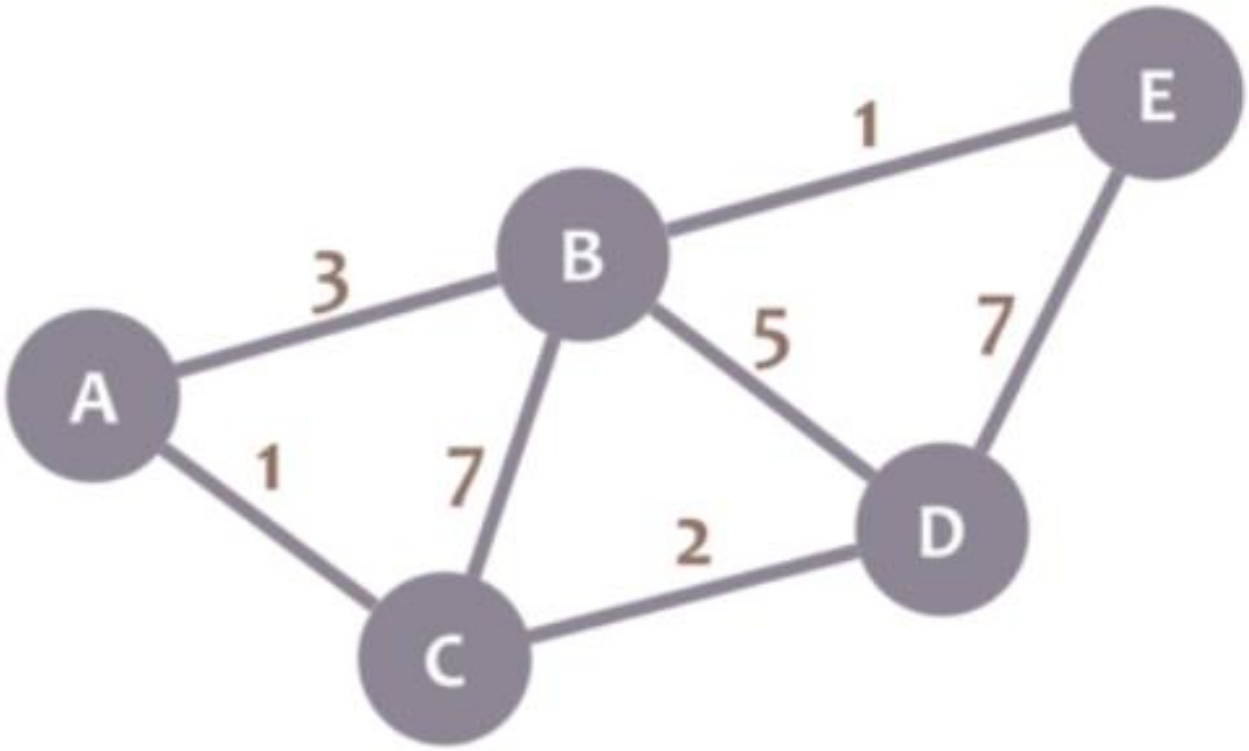
Before proceeding the step by step process for implementing the algorithm, let us consider some essential characteristics of Dijkstra's algorithm;

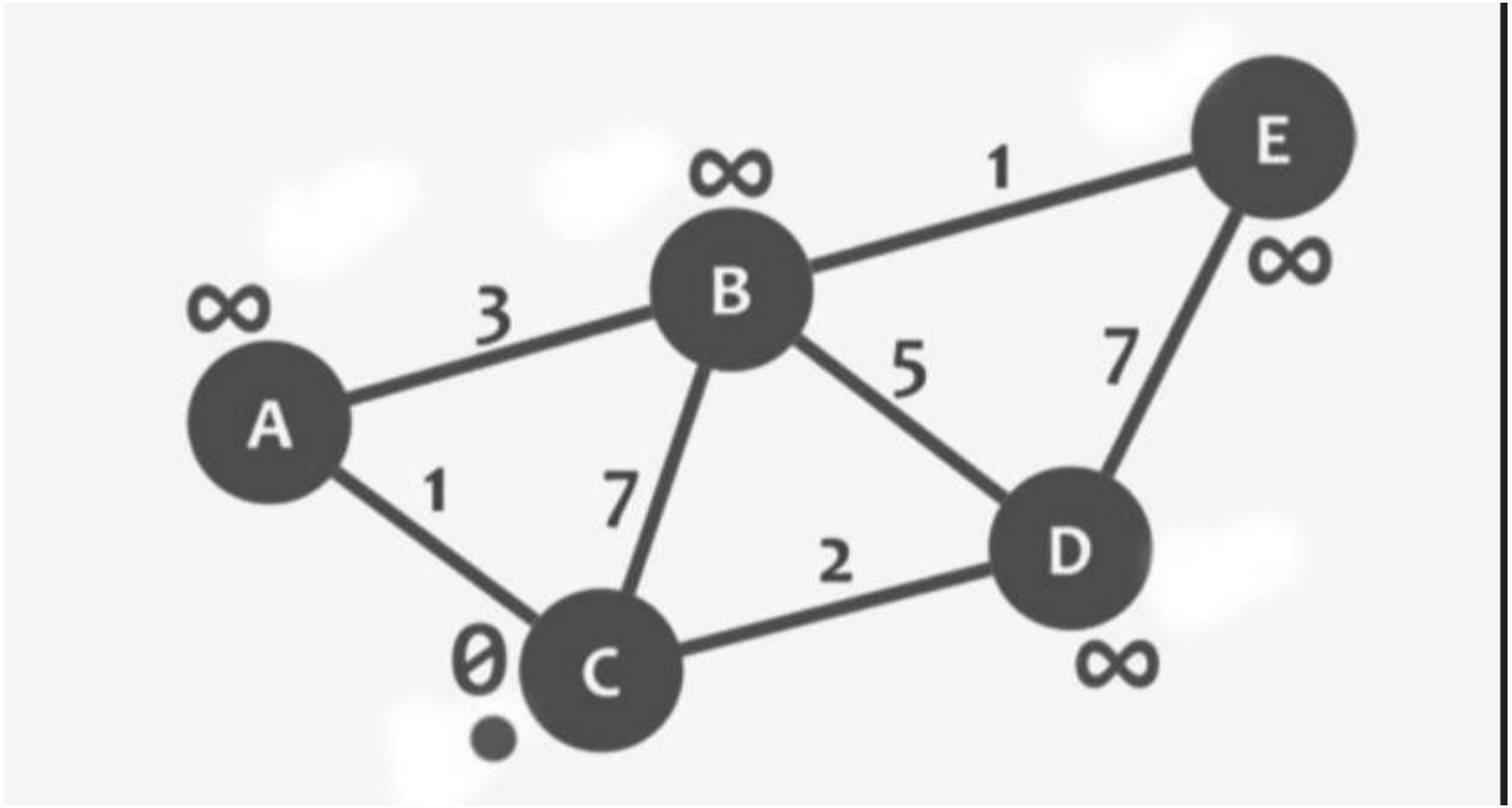
- Basically, the Dijkstra's algorithm begins from the node to be selected, the source node, and it examines the entire graph to determine the shortest path among that node and all the other nodes in the graph.
- The algorithm maintains the track of the currently recognized shortest distance from each node to the source code and updates these values if it identifies another shortest path.
- Once the algorithm has determined the shortest path amid the source code to another node, the node is marked as "visited" and can be added to the path.
- This process is being continued till all the nodes in the graph have been added to the path, as this way, a path gets created that connects the source node to all the other nodes following the plausible shortest path to reach each node.

Steps of Dijkstra Algorithm

- The very first step is to mark all nodes as unvisited,
- Mark the picked starting node with a current distance of 0 and the rest nodes with infinity,
- Now, fix the starting node as the current node,
- For the current node, analyze all of its unvisited neighbours and measure their distances by adding the current distance of the current node to the weight of the edge that connects the neighbour node and current node,
- Compare the recently measured distance with the current distance assigned to the neighbouring node and make it as the new current distance of the neighbouring node,
- After that, consider all of the unvisited neighbours of the current node, mark the current node as visited,
- If the destination node has been marked visited then stop, an algorithm has ended, and
- Else, choose the unvisited node that is marked with the least distance, fix it as the new current node, and repeat the process again from step 4.

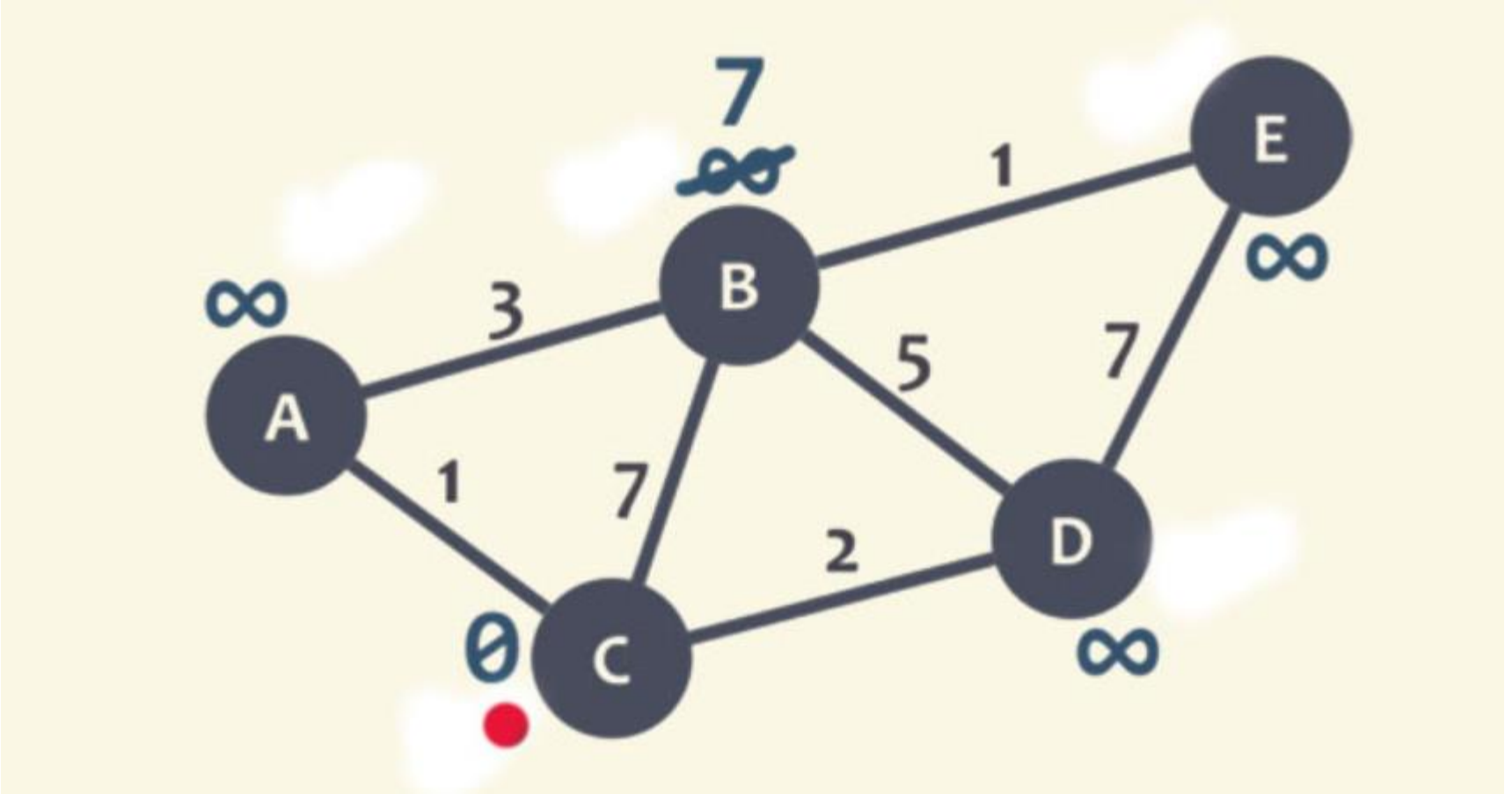
Working Example of Dijkstra's Algorithm





1.During the execution of the algorithm, each node will be marked with its minimum distance to node C as we have selected node C.

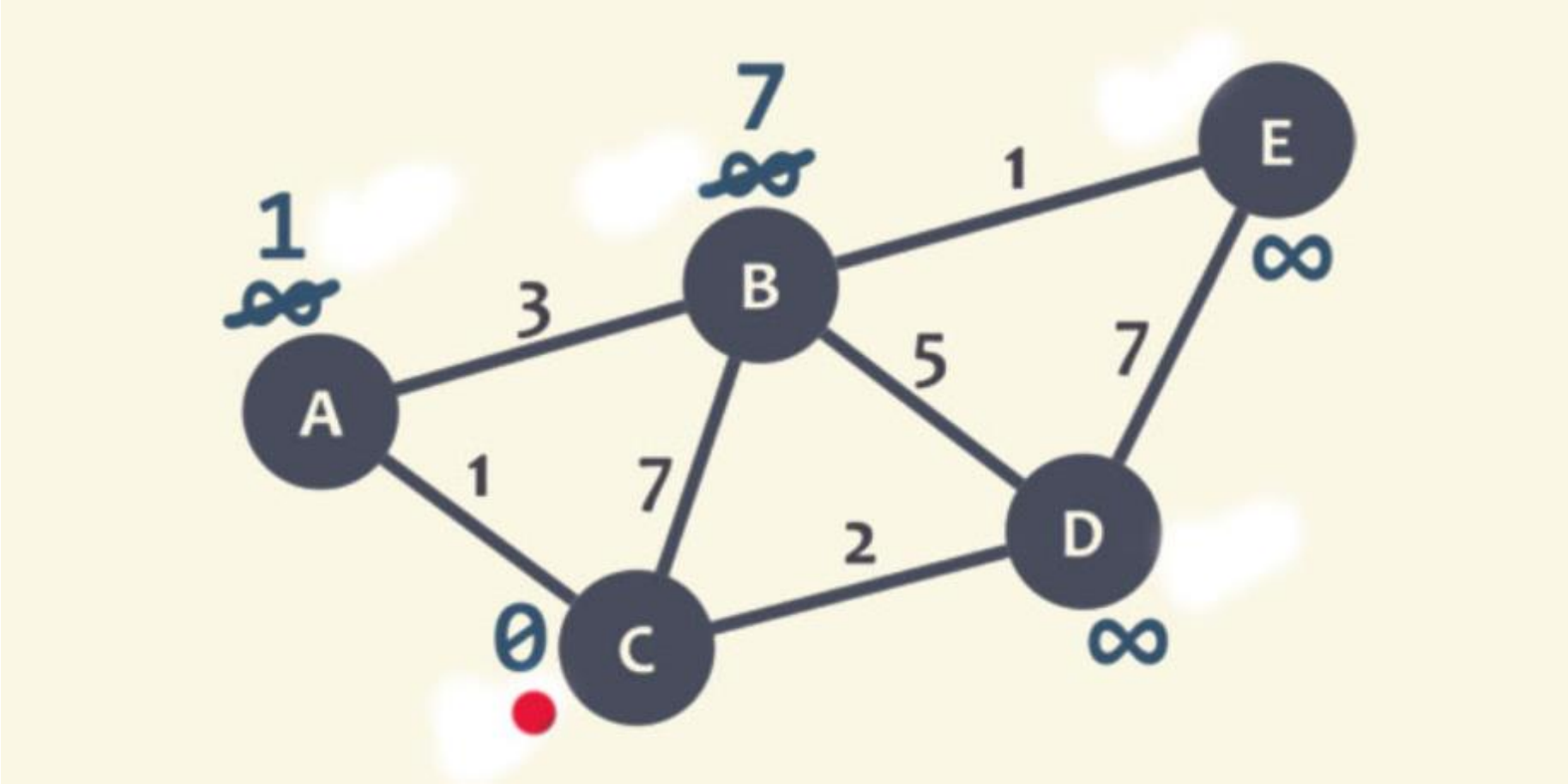
In this case, the minimum distance is 0 for node C. Also, for the rest of the nodes, as we don’t know this distance, they will be marked as infinity (∞), except node C (currently marked as red dot).



Assign Node B a minimum distance value

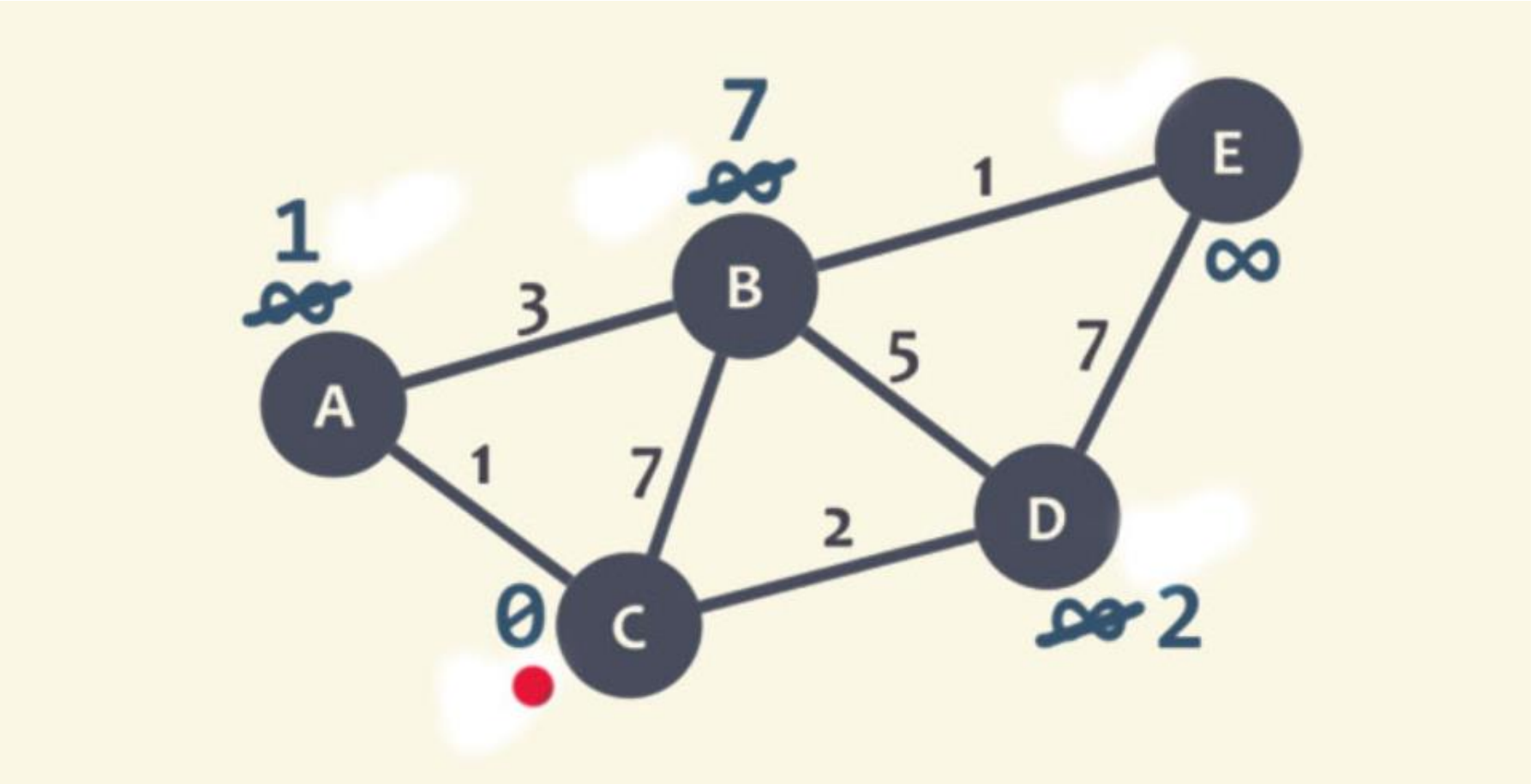
2.Now the neighbours of node C will be checked, i.e, node A, B, and D. We start with B, here we will add the minimum distance of current node (0) with the weight of the edge (7) that linked the node C to node B and get 0+ 7= 7.

Now, this value will be compared with the minimum distance of B (infinity), the least value is the one that remains the minimum distance of B, like in this case, 7 is less than infinity, and marks the least value to node B.



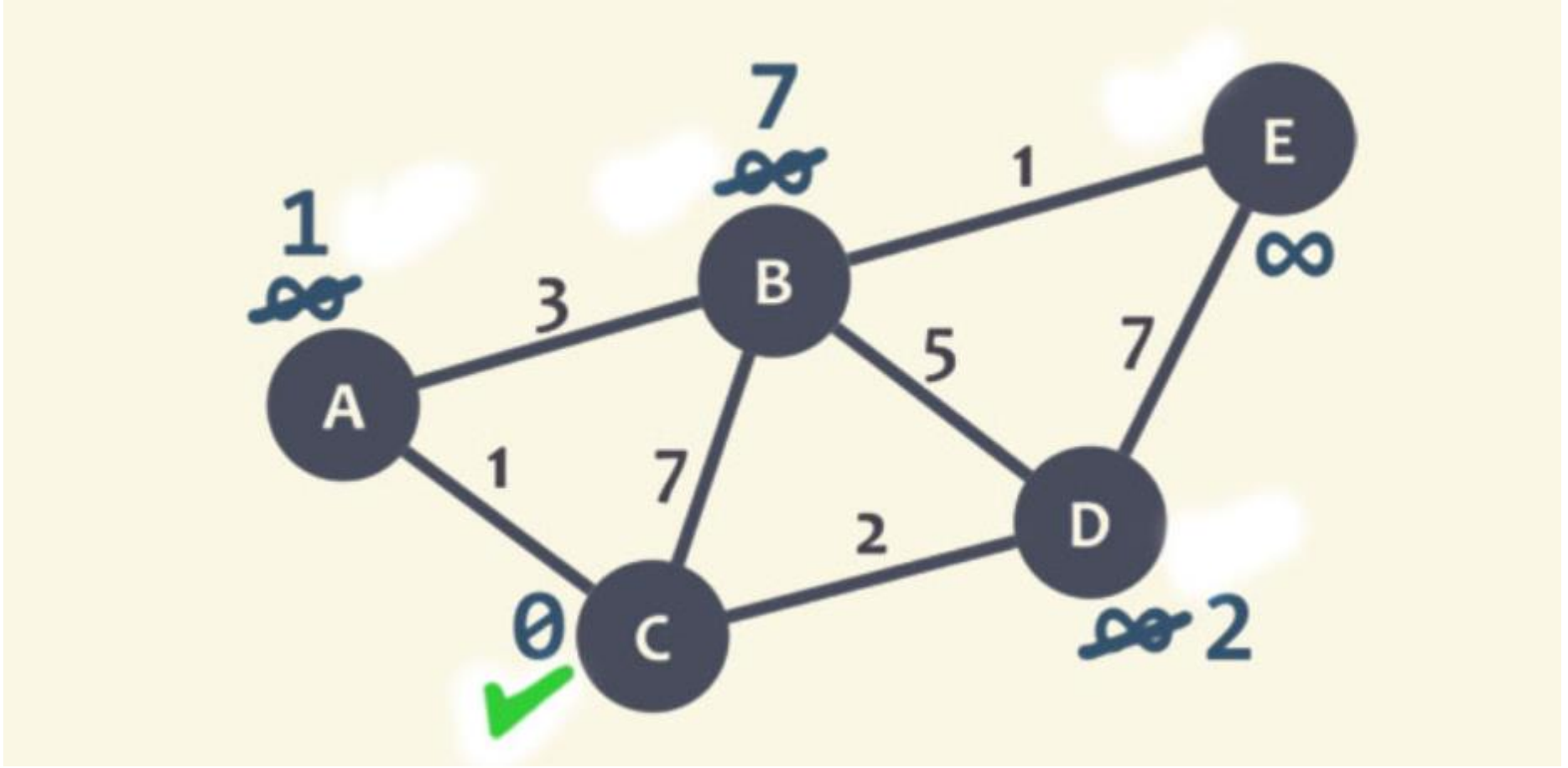
Assign Node A a minimum distance value

Now, the same process is checked with neighbour A. We add 0 with 1 (weight of edge that connects node C to A), and get 1. Again, 1 is compared with the minimum distance of A (infinity), and marks the lowest value.

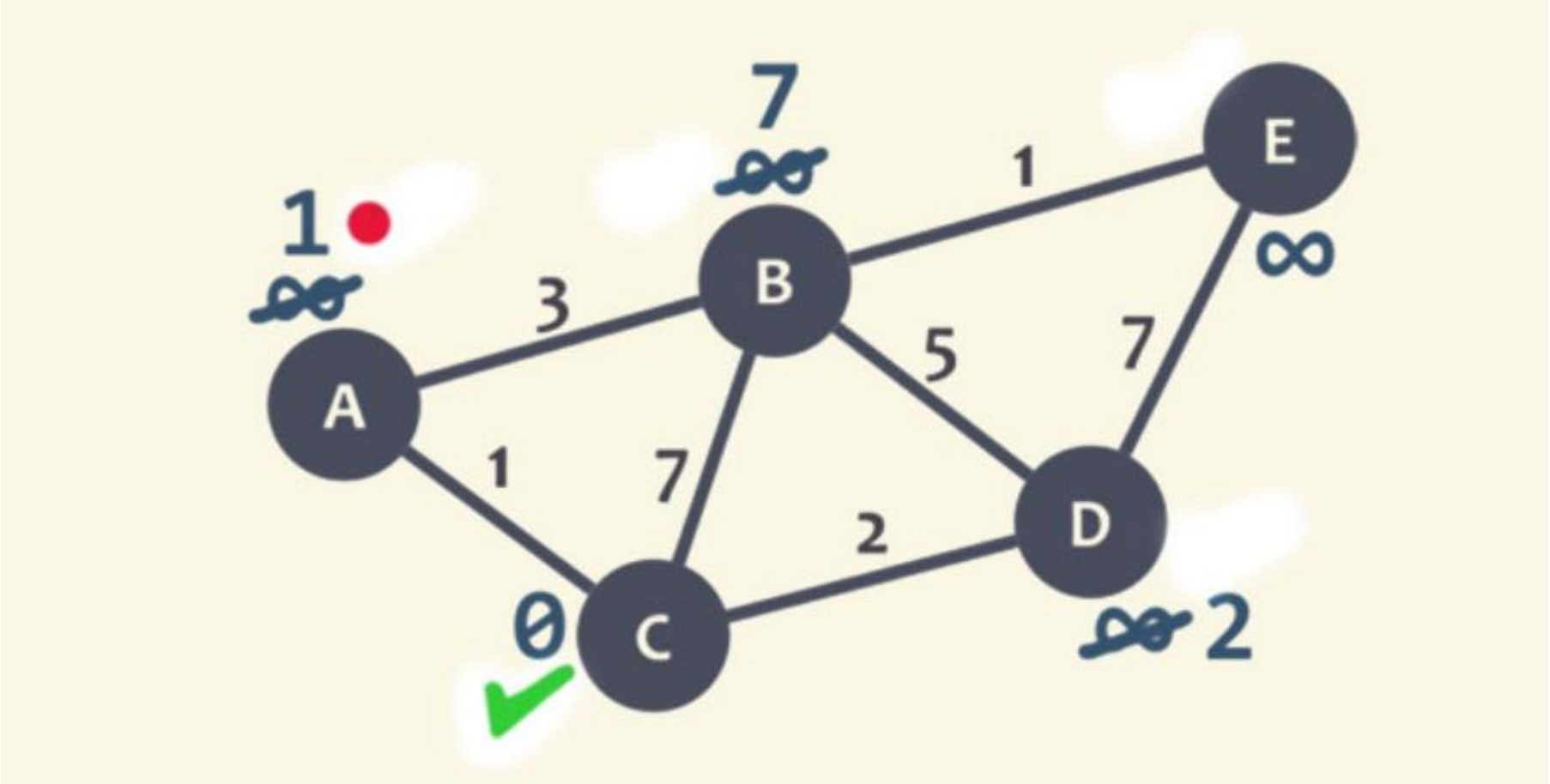


Assign Node D a minimum distance value

Since, all the neighbours of node C have checked, so node C is marked as visited with a green check mark.



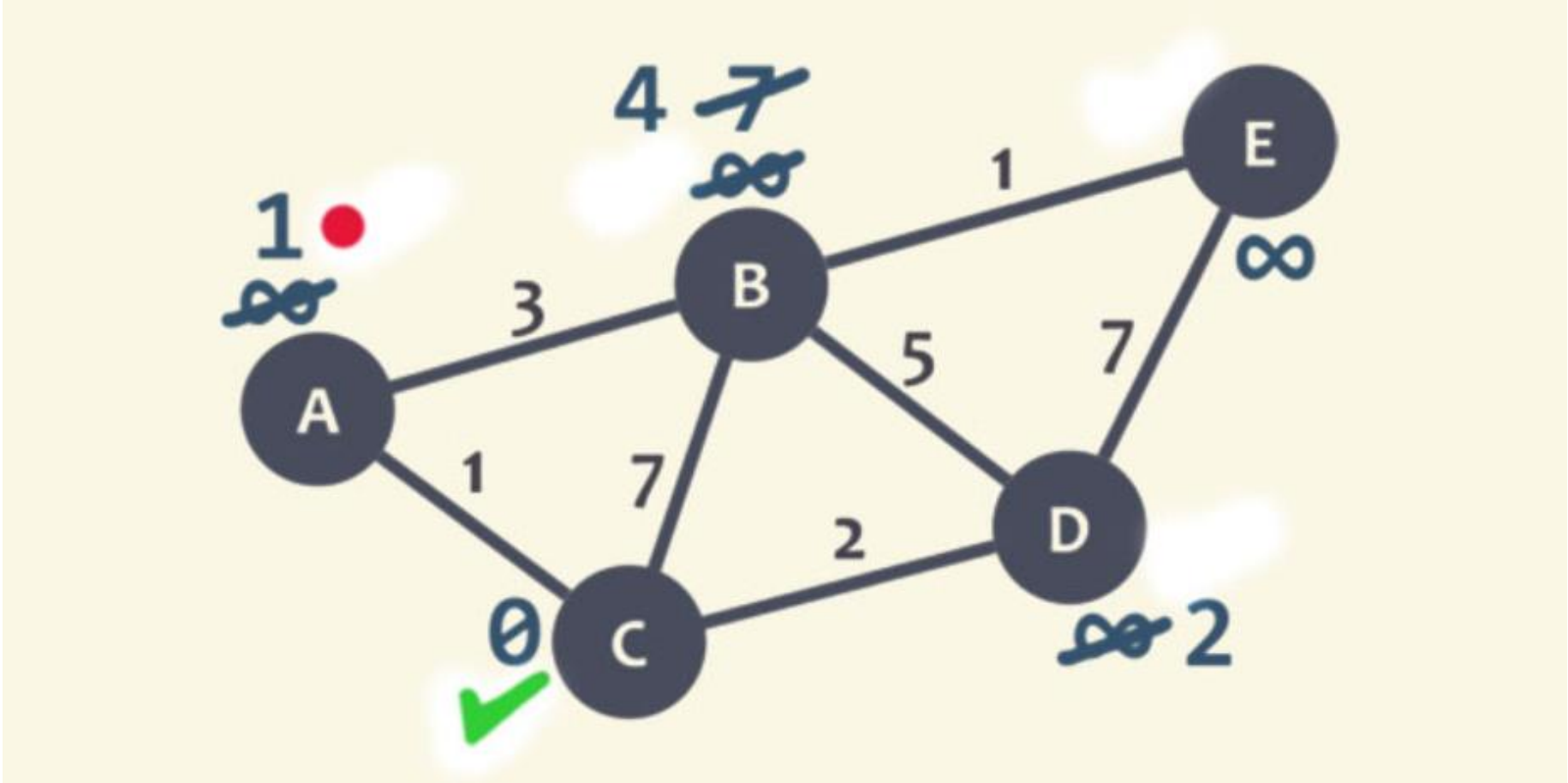
Marked Node C as visited



Graphical Representation of Node A as Current Node

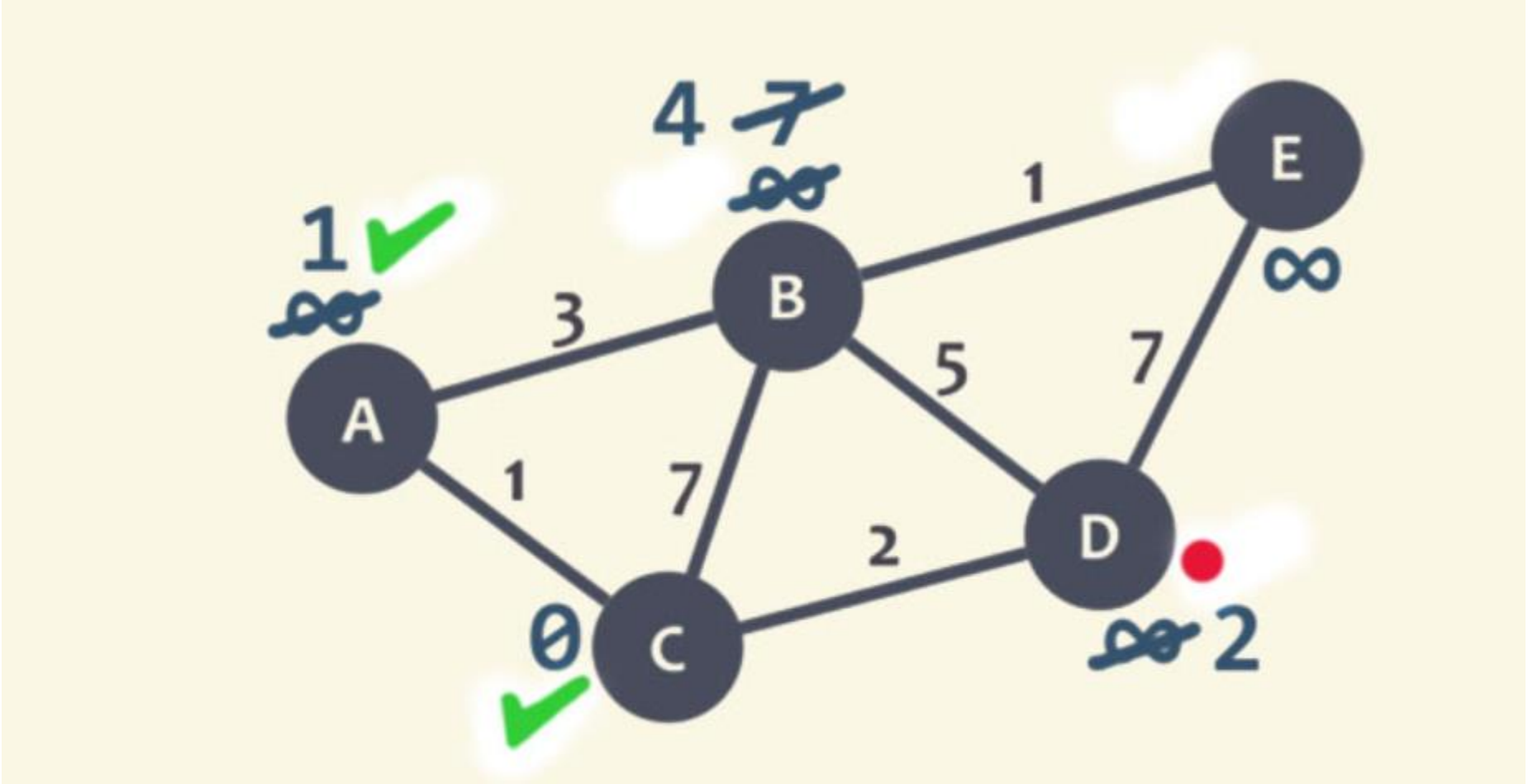
Now, we will select the new current node such that the node must be unvisited with the lowest minimum distance, or the node with the least number and no check mark. Here, node A is the unvisited with minimum distance 1, marked as current node with red dot.

We repeat the algorithm, checking the neighbour of the current node while ignoring the visited node, so only node B will be checked.



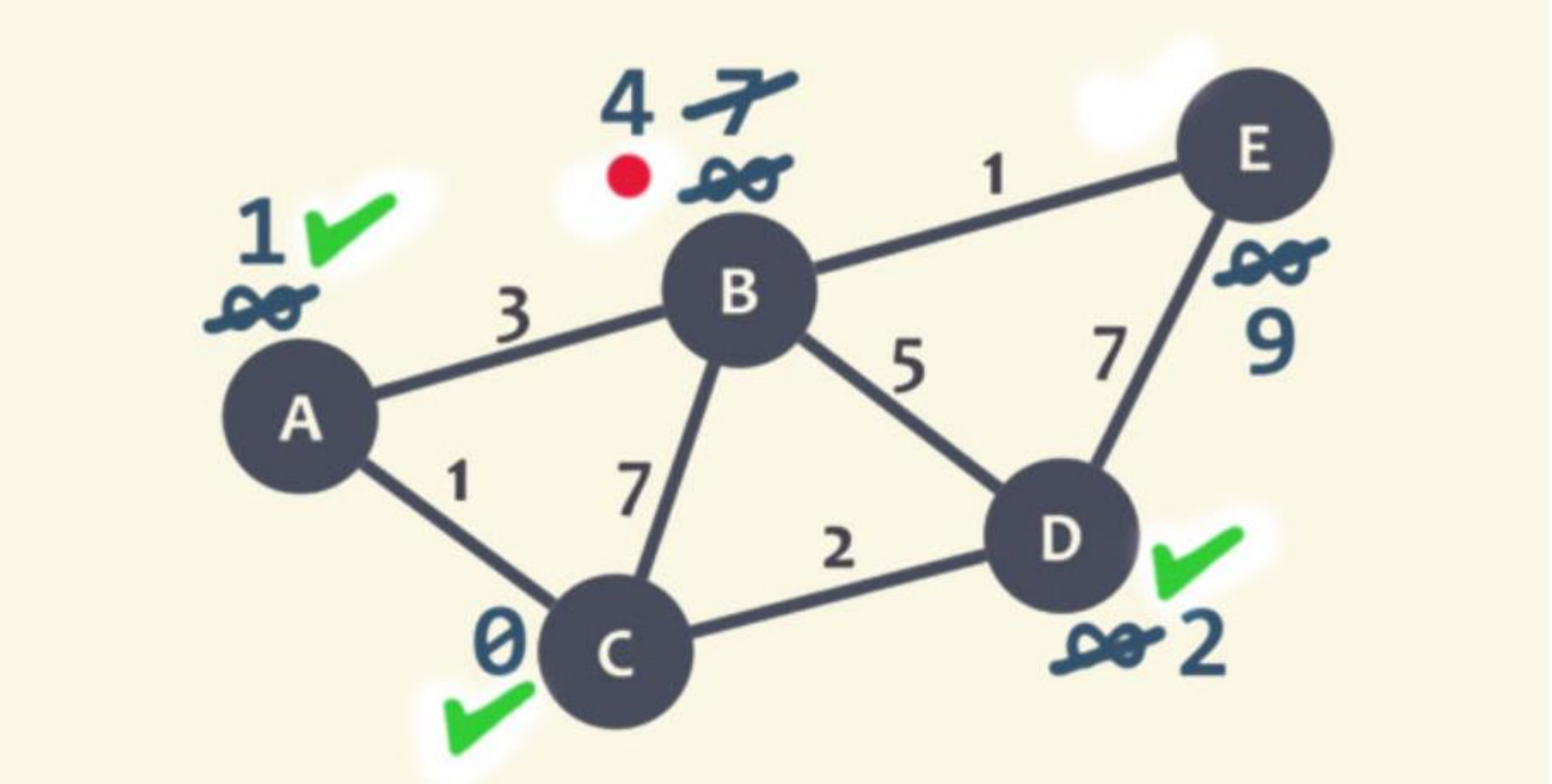
Assign Node B a minimum distance value

For node B, we add 1 with 3 (weight of the edge connecting node A to B) and obtain 4. This value, 4, will be compared with the minimum distance of B, 7, and mark the lowest value at B as 4.



Graphical Representation of Node D as Current Node

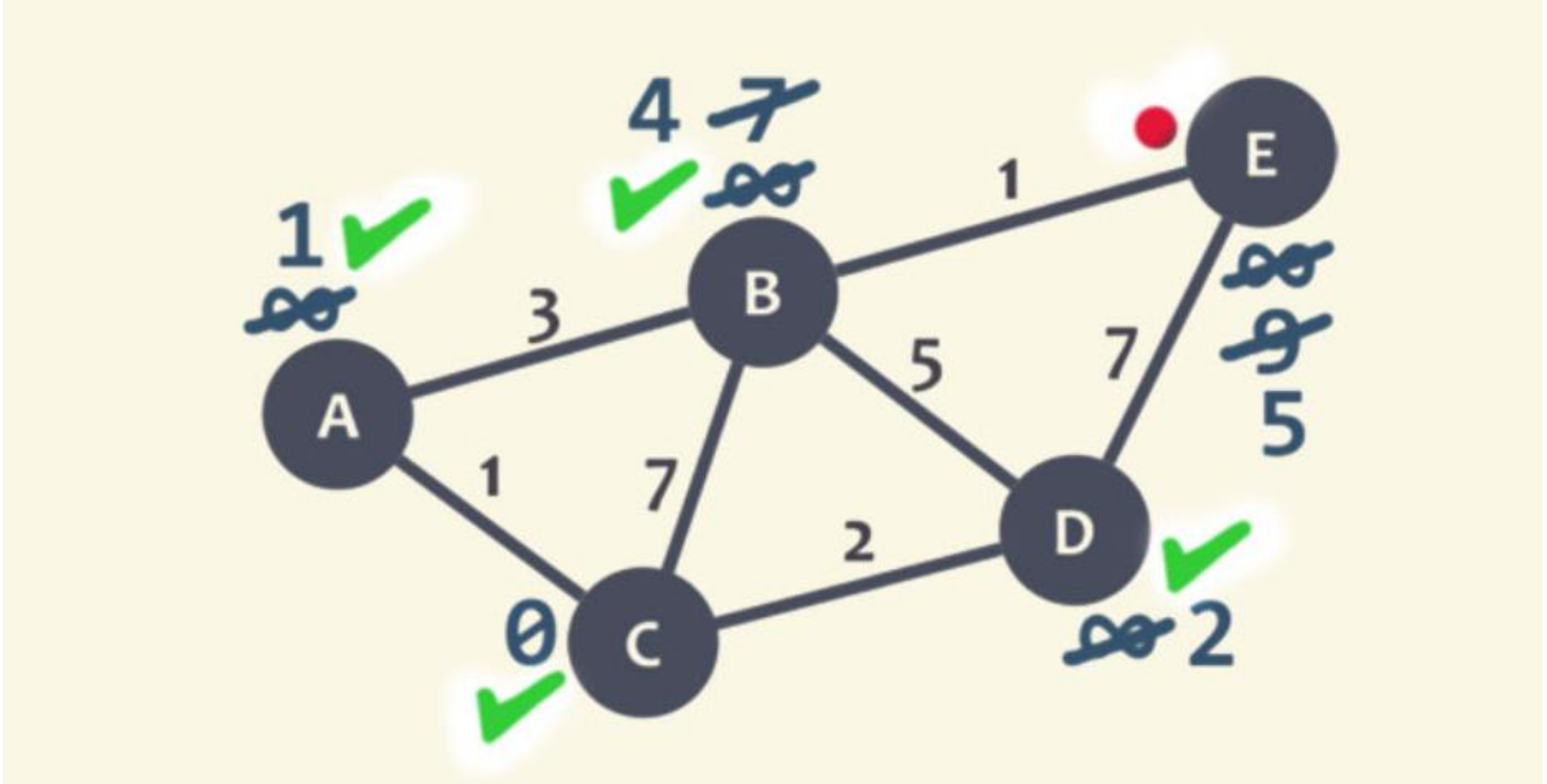
5.After this, node A marked as visited with a green check mark. The current node is selected as node D, it is unvisited and has a smallest recent distance. We repeat the algorithm and check for node B and E.



Marked Node D as visited

For node B, we add 2 to 5, get 7 and compare it with the minimum distance value of B, since 7>4, so leave the smallest distance value at node B as 4.

For node E, we obtain 2+ 7= 9, and compare it with the minimum distance of E which is infinity, and mark the smallest value as node E as 9. The node D is marked as visited with a green check mark.

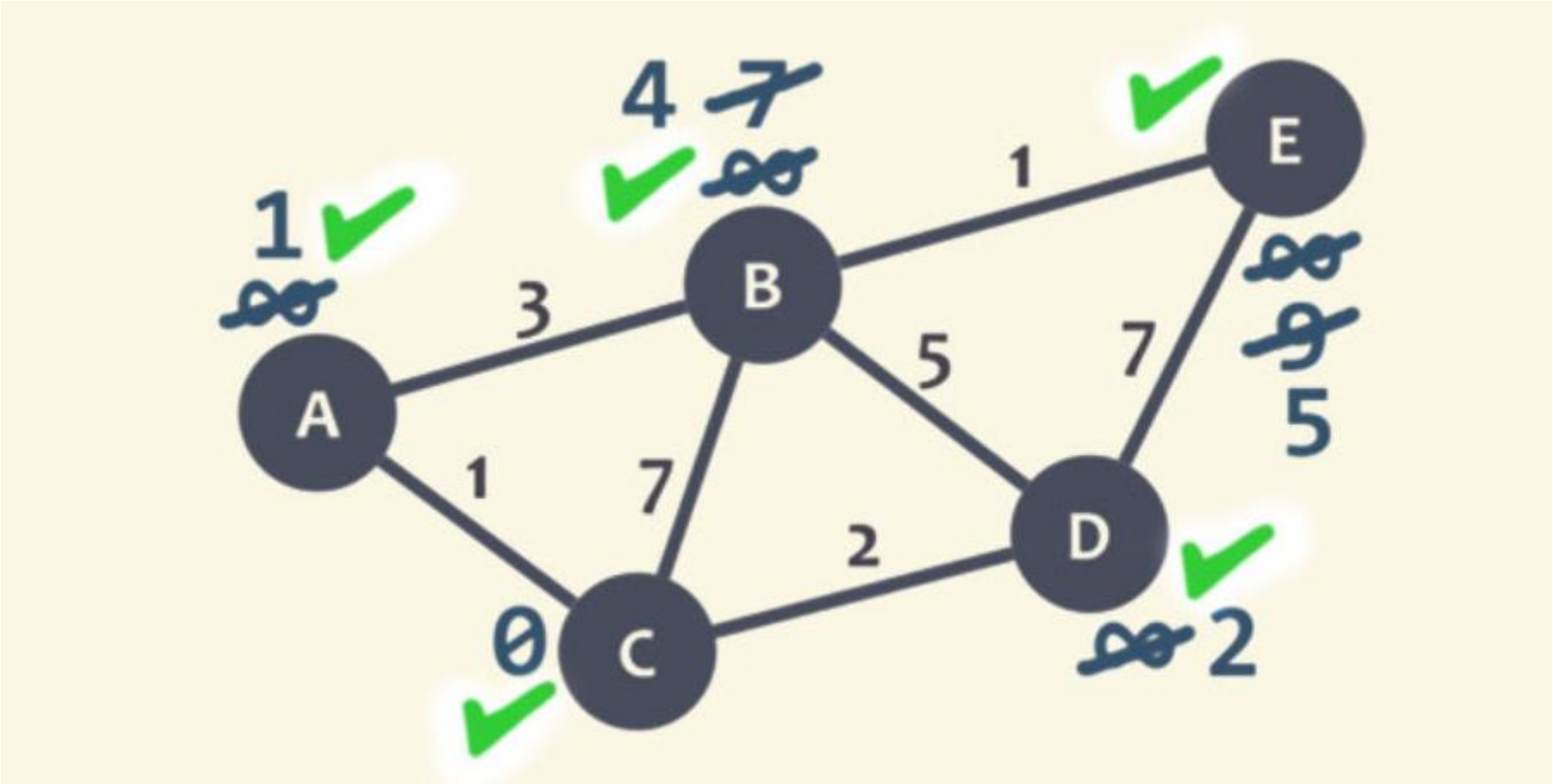


Marked Node B as visited

6.The current node is set as node B, here we need to check only node E as it is unvisited and the node D is visited. We obtain 4+ 1=5, compare it with the minimum distance of the node.

As 9 > 5, leave the smallest value at node node E as 5.

We mark D as visited node with a green check mark, and node E is set as the current node.



Marked Node E as visited

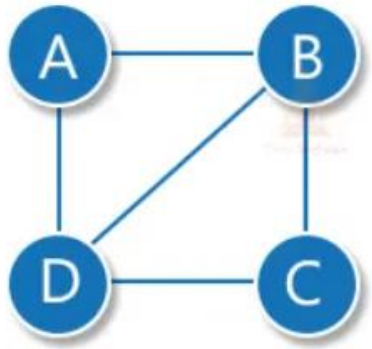
7.Since it doesn't have any unvisited neighbours, so there is not any requirement to check anything. Node E is marked as a visited node with a green mark.

So, we are done as no unvisited node is left. The minimum distance of each node is now representing the minimum distance of that node from node C.

Graphs Representation in data structure

1. Adjacency matrix :

- In it, we have a matrix of order n*n where n is the number of nodes in the graph. The matrix represents the mapping between various edges and vertices.
- In the matrix, each row and column represents a vertex. The values determine the presence of edges.
- Let Aij represents each element of the adjacency matrix. Then,
- For an undirected graph, the value of Aij is 1 if there exists an edge between i and j. Otherwise, the value of Aij is 0.

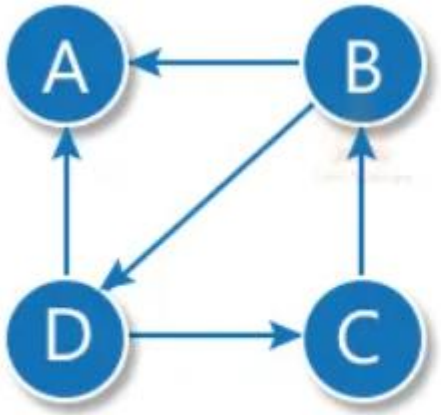


Undirected graph

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	1	0

Adjacency matrix

- For a directed graph, the value of Aij is 1 only if there is an edge from i to j i.e. **i** is the initial node and **j** is the terminal node.



Directed graph

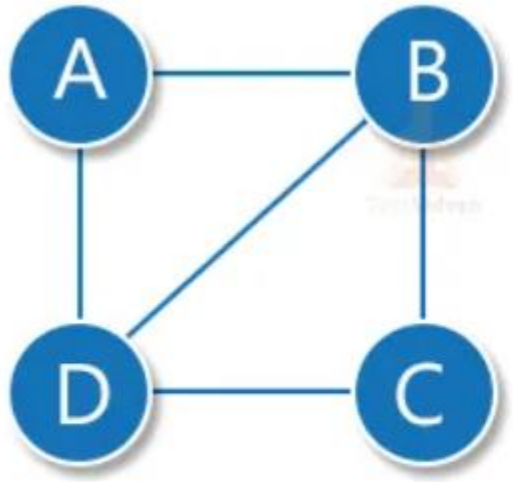
	A	B	C	D
A	0	0	0	0
B	1	0	0	1
C	0	1	0	0
D	1	0	1	0

Adjacency matrix

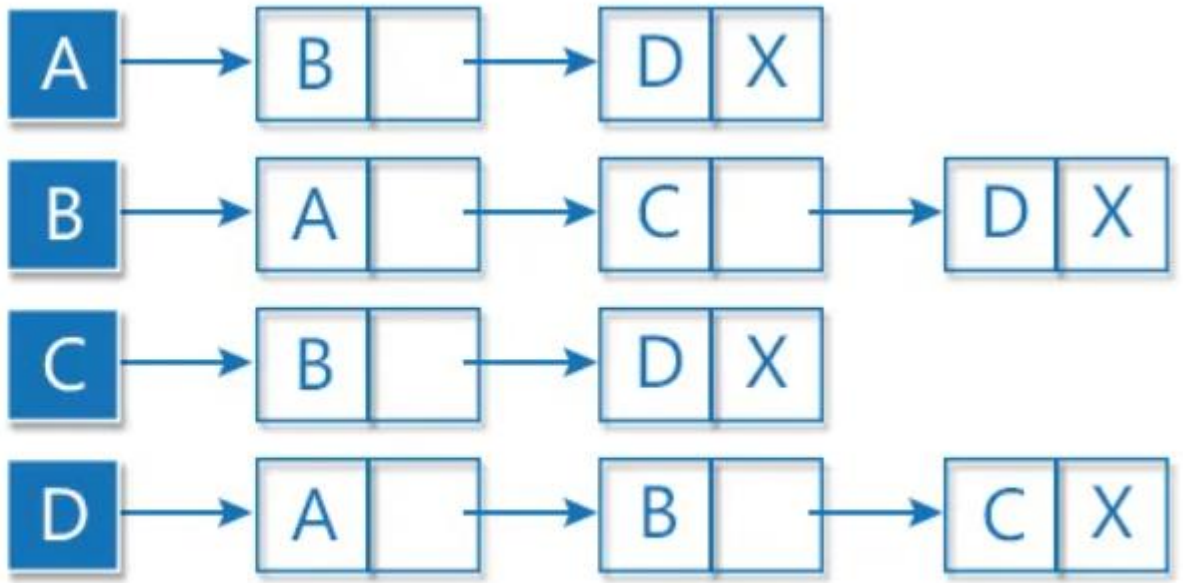
- The time complexity of the adjacency matrix is $O(n^2)$.

2. Adjacency list

- The adjacency list is an array of linked lists where the array denotes the total vertices and each linked list denotes the vertices connected to a particular node.
- In a linked list, the most important component is the pointer named ‘Head’ because this single pointer maintains the whole linked list. For linked list representation, we will have total pointers equal to the number of nodes in the graph.
- For an undirected graph, we will link all the edges in the list that are connected to a node as shown:

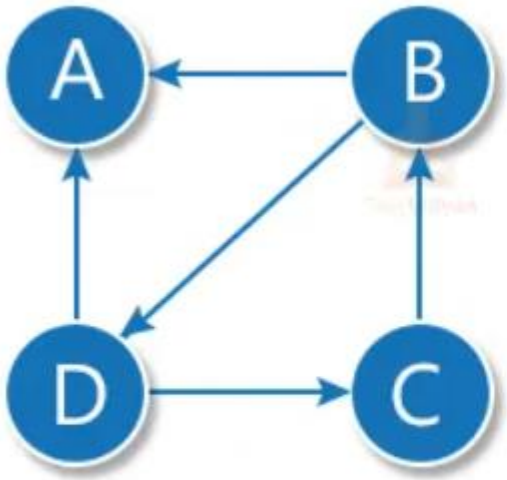


Undirected graph

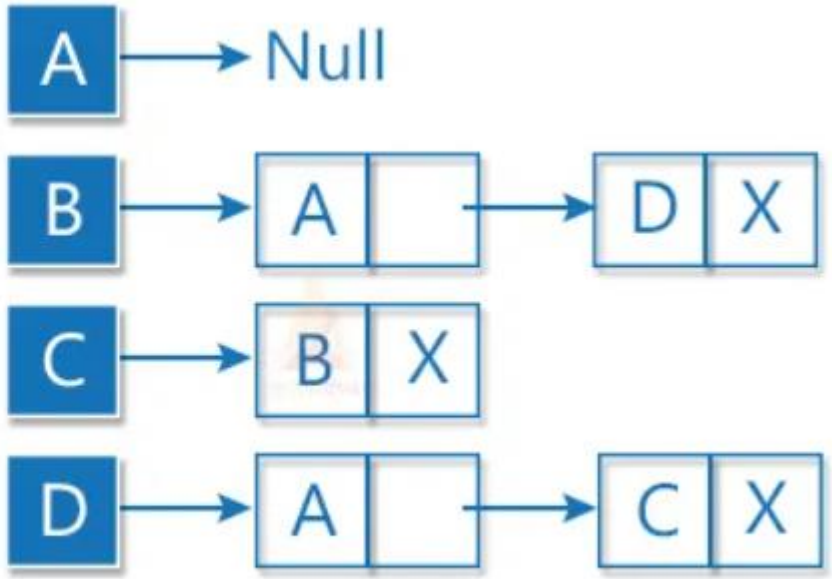


Adjacency List

- In a directed graph, we will link only the initial nodes in the list as shown:



Directed graph



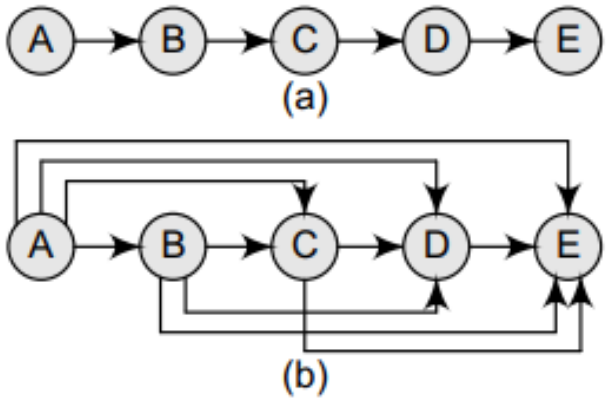
Adjacency List

Transitive Closure

- Transitive Closure it the reachability matrix to reach from vertex u to vertex v of a graph. One graph is given, we have to find a vertex v which is reachable from another vertex u, for all vertex pairs (u, v).
- It states if there is a path from vertex a to b then there should be an edge from a to b.
- For finding transitive closure of a graph
  - Add an edge from a to c if there exists a path from a to b and b to c
  - Repeat this process of adding edge until no new edges are added.
  - Hence, it can be defined as, If G=(V,E) in a graph then its transitive closure can be defined as G*=(V,E*) where E*={(V_i,V_j): there exists a path from V_i to V_j in G}

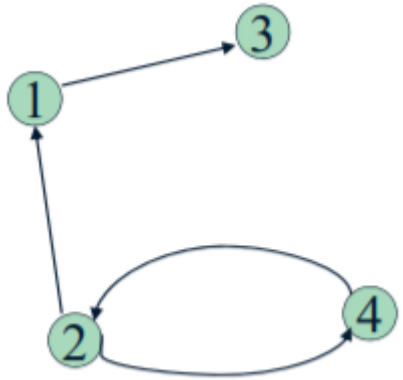
Transitive Closure of a Directed Graph

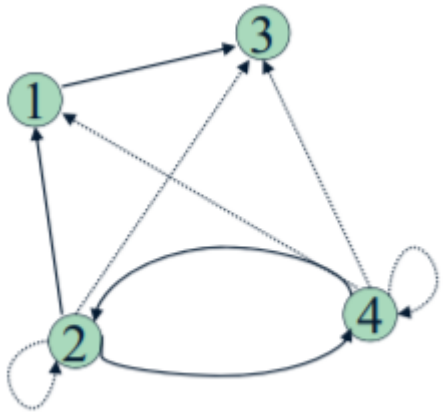
- A transitive closure of a graph is constructed to answer reachability questions. That is, is there a path from a node A to node E in one or more hops?
- A binary relation indicates only whether the node A is connected to node B, whether node B is connected to node C, etc.
- But once the transitive closure is constructed as shown in Figure, we can easily determine in O(1) time whether node E is reachable from node A or not.
- transitive closure is also stored as a matrix T, so if T[1][5] = 1, then node 5 can be reached from node 1 in one or more hops.



(a) A graph G and its (b) transitive closure G*

Example of transitive closure:





V	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0

V	1	2	3	4
1	0	0	1	0
2	1	1	1	1
3	0	0	0	0
4	1	1	1	1

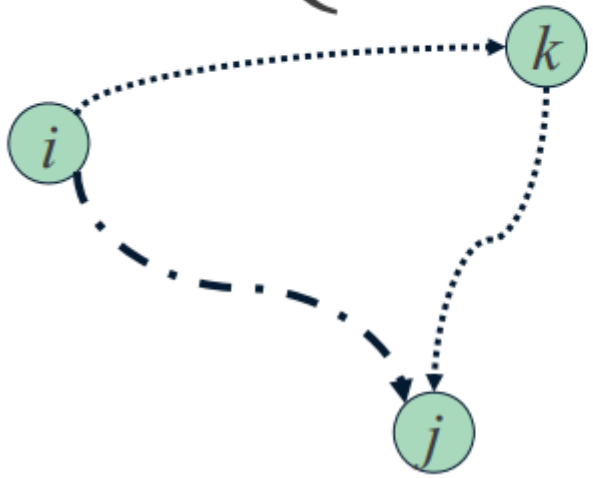
Warshall's Algorithm

- Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix.
- For this, it generates a sequence of n matrices where, n is used to describe the number of vertices.
- $R^{(0)}$, ..., $R^{(k-1)}$, $R^{(k)}$, ... , $R^{(n)}$

Warshall's Algorithm

- Diagram – Adjacency Matrix- transitive closure through iteration

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just 1 ,…,}k\text{-1)} \\ \text{or} & \\ (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just 1 ,…,}k\text{-1)} \end{cases}$$



k^{th} iteration

Warshall's Algorithm

- Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:
- $R^{(k)}[i,j] = R^{(k-1)}[i,j]$ or ($R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j]$)
- It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:
  Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$
  Rule 2 If an element in row i and column j is 0 in $R^{(k-1)}$, it has to be changed to 1 in R(k) it has to be changed to 1 in R if and only if (k) if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.

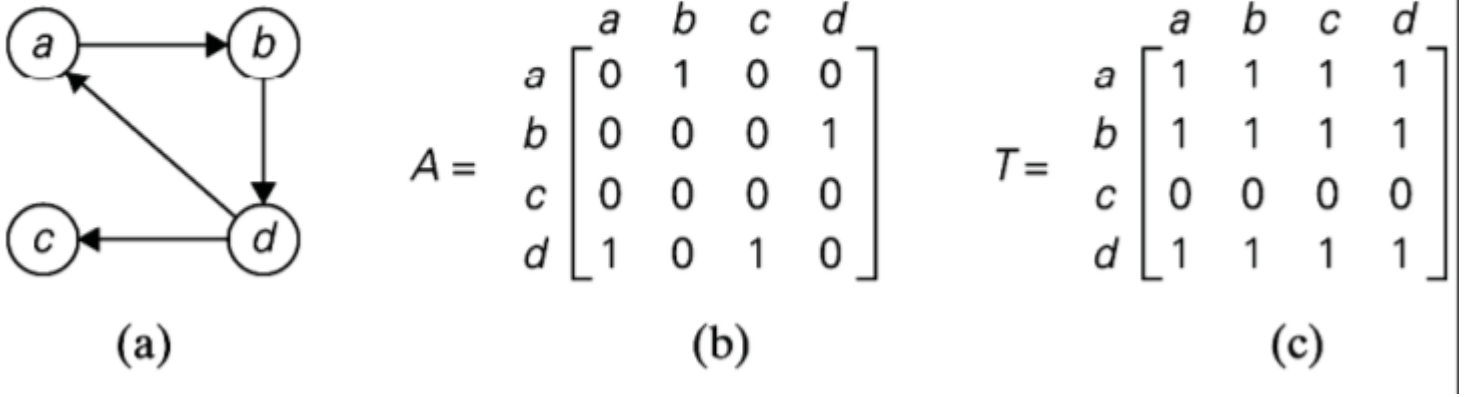


Fig: (a)Diagraph (b) its adjacency matrix (c) its transitive closure

Warshall's Algorithm: Transitive Closure

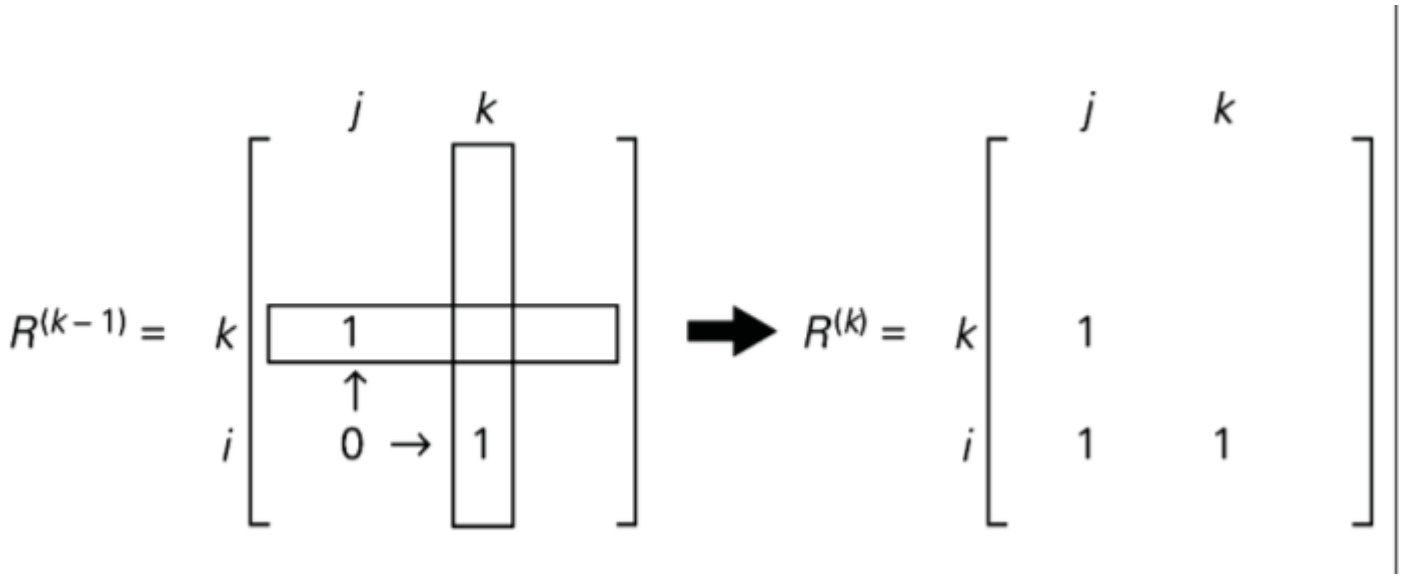


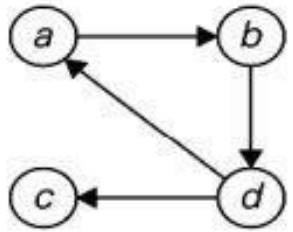
Fig: Rule for changing zero's in Warshall's Algorithm

Warshall's Algorithm

ALGORITHM *Warshall*$(A[1..n, 1..n])$

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Warshall's Algorithm Example



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a, b,* and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)}$ =

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a, b, c,* and *d* (note five new paths).

MCQs Practice

of

Introduction to data structures, list, linked list and trees

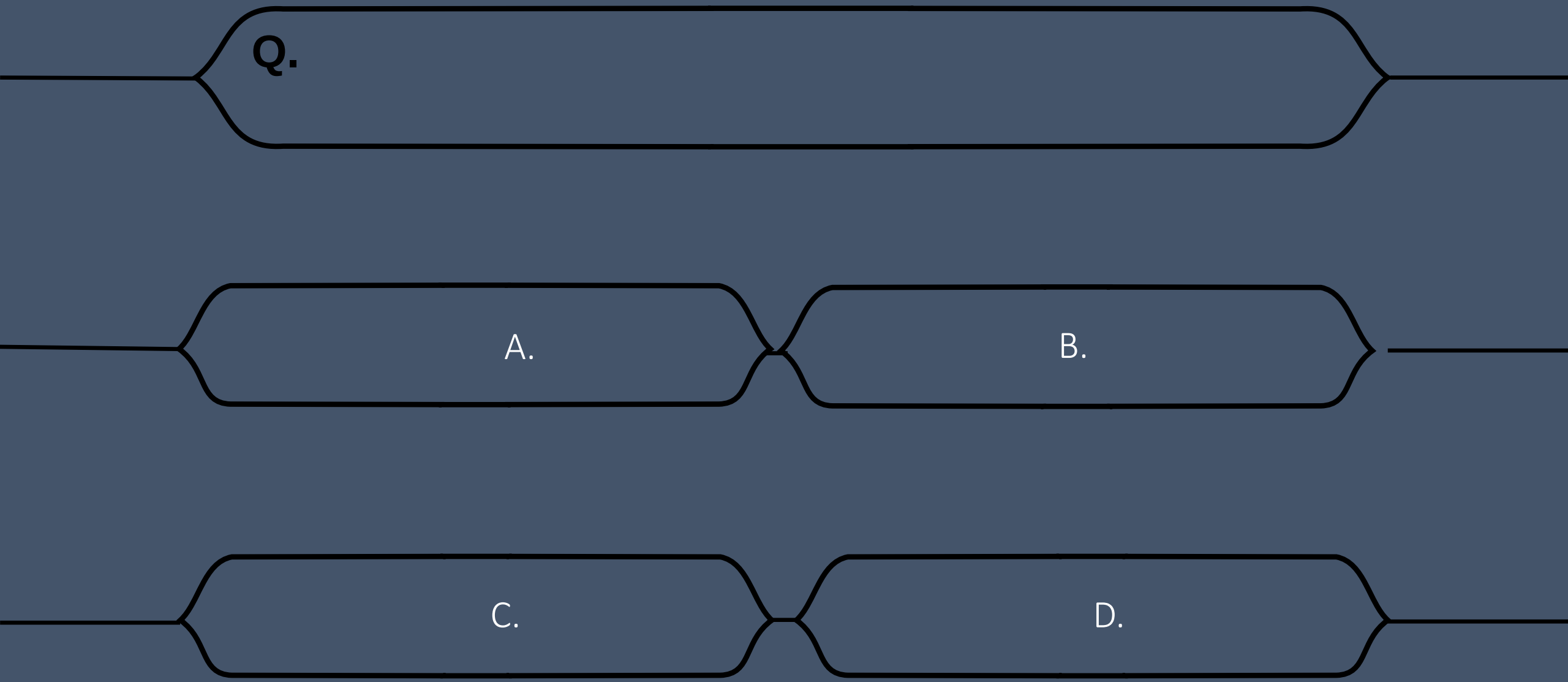
Q. What is an external sorting algorithm?

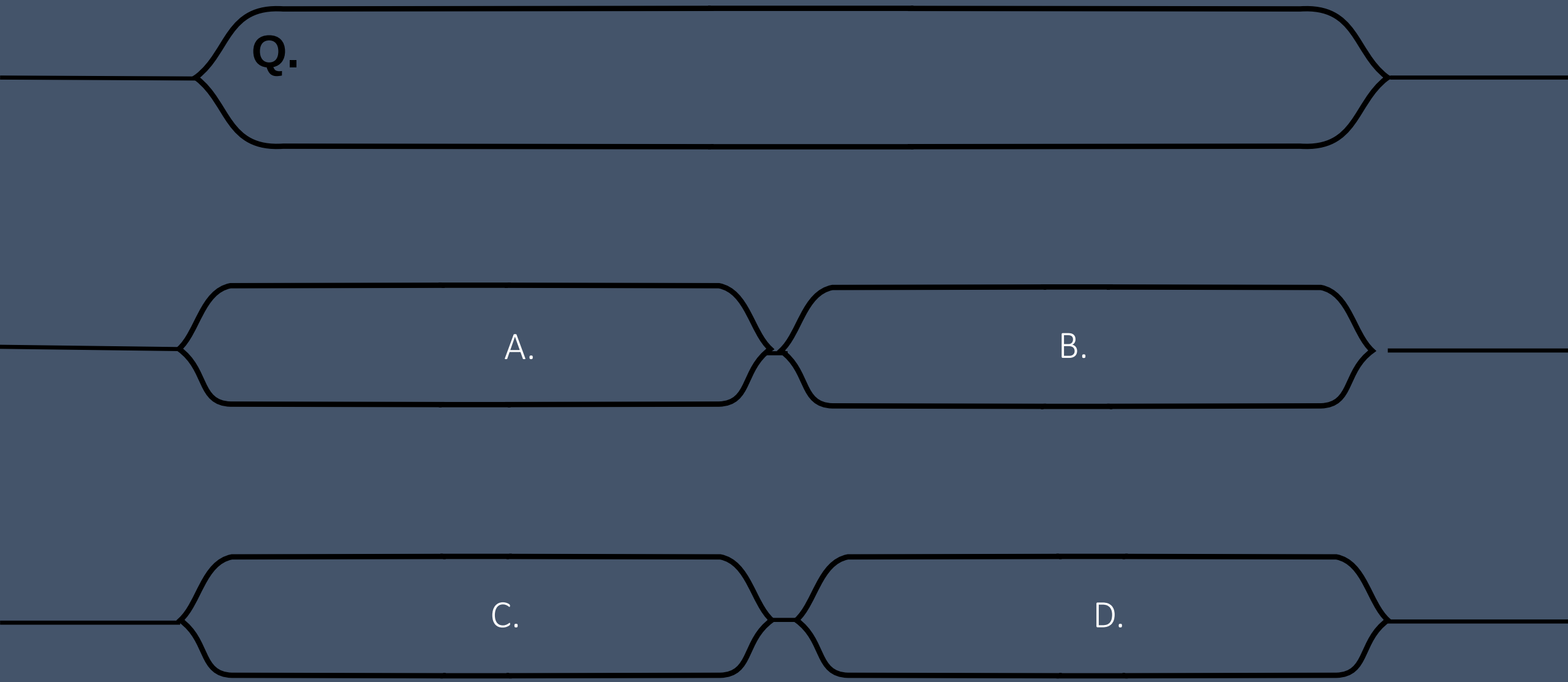
A. Algorithm that uses tape or disk during the sort

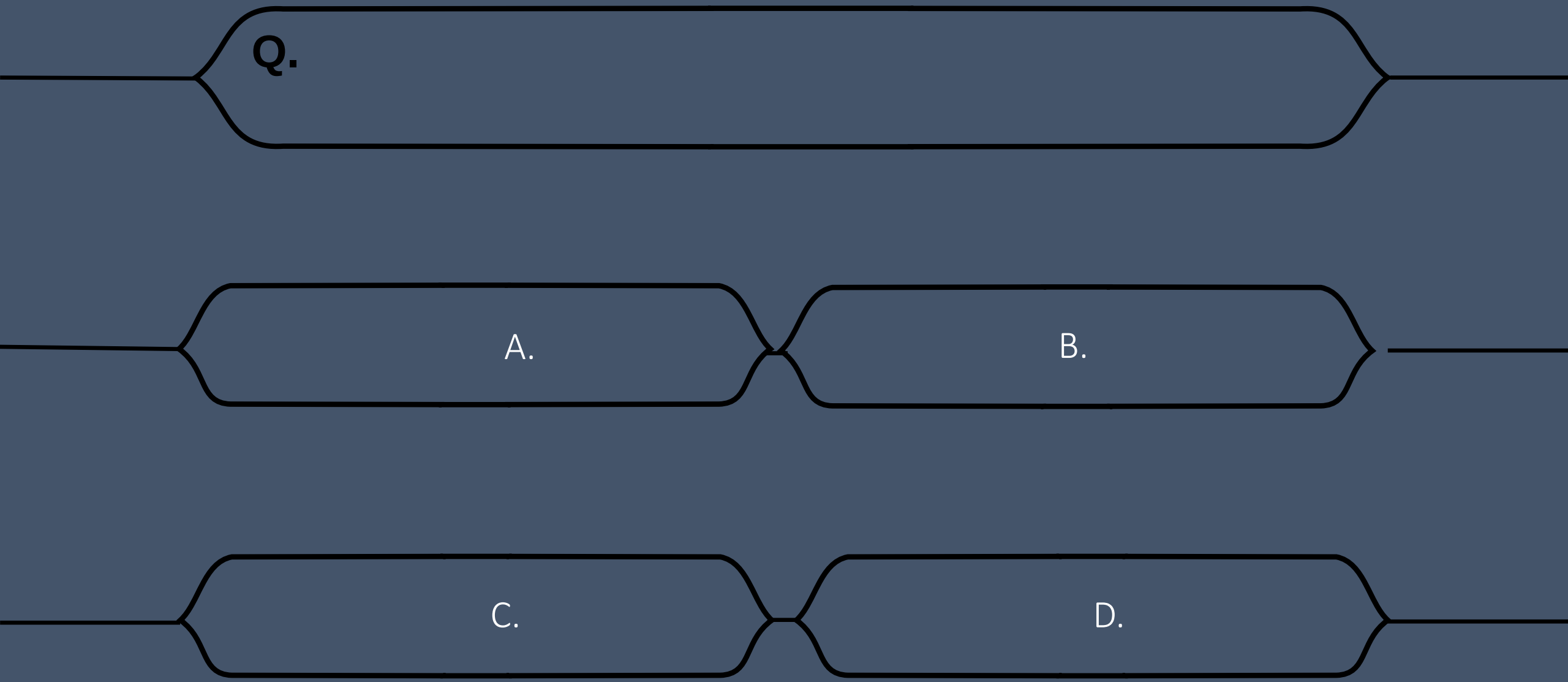
B. Algorithm that uses main memory during the sort

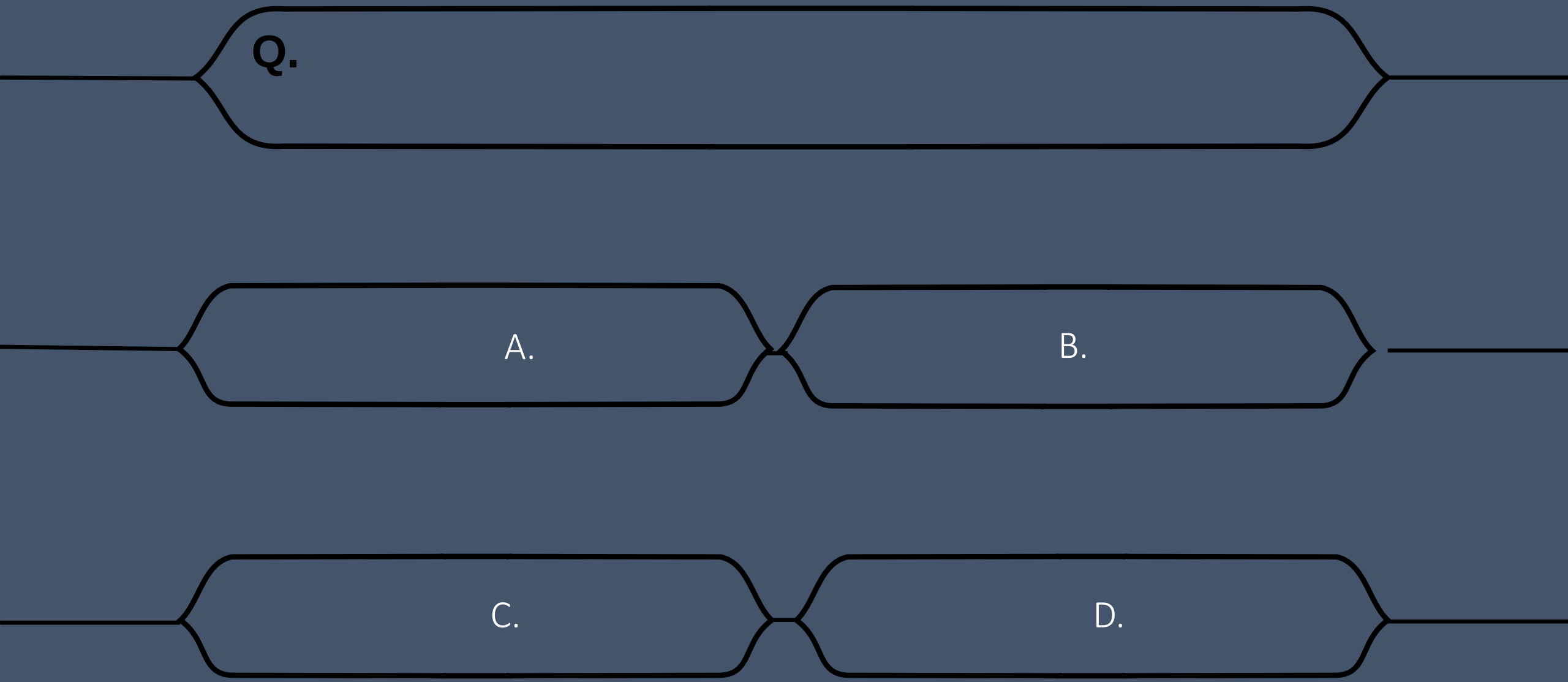
C. Algorithm that involves swapping

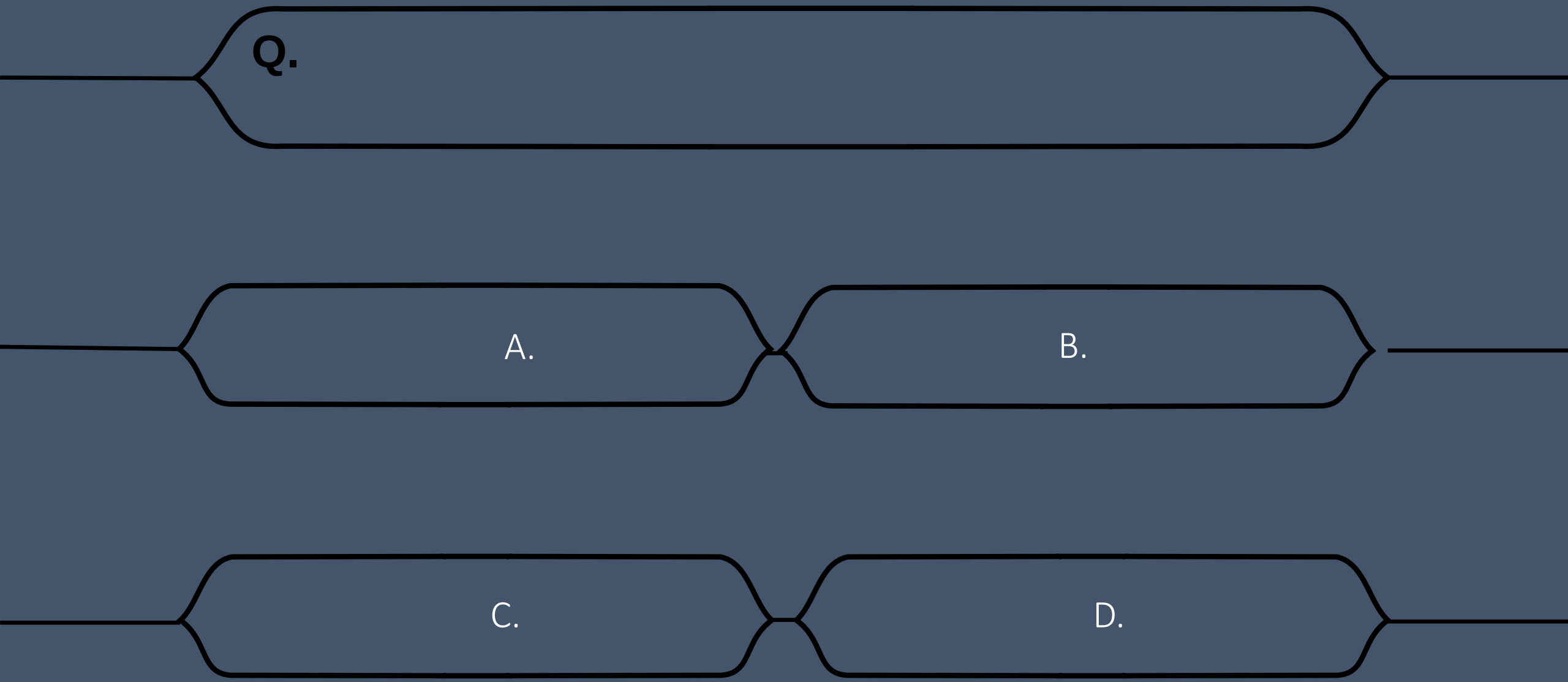
D. Algorithm that are considered ‘in place’

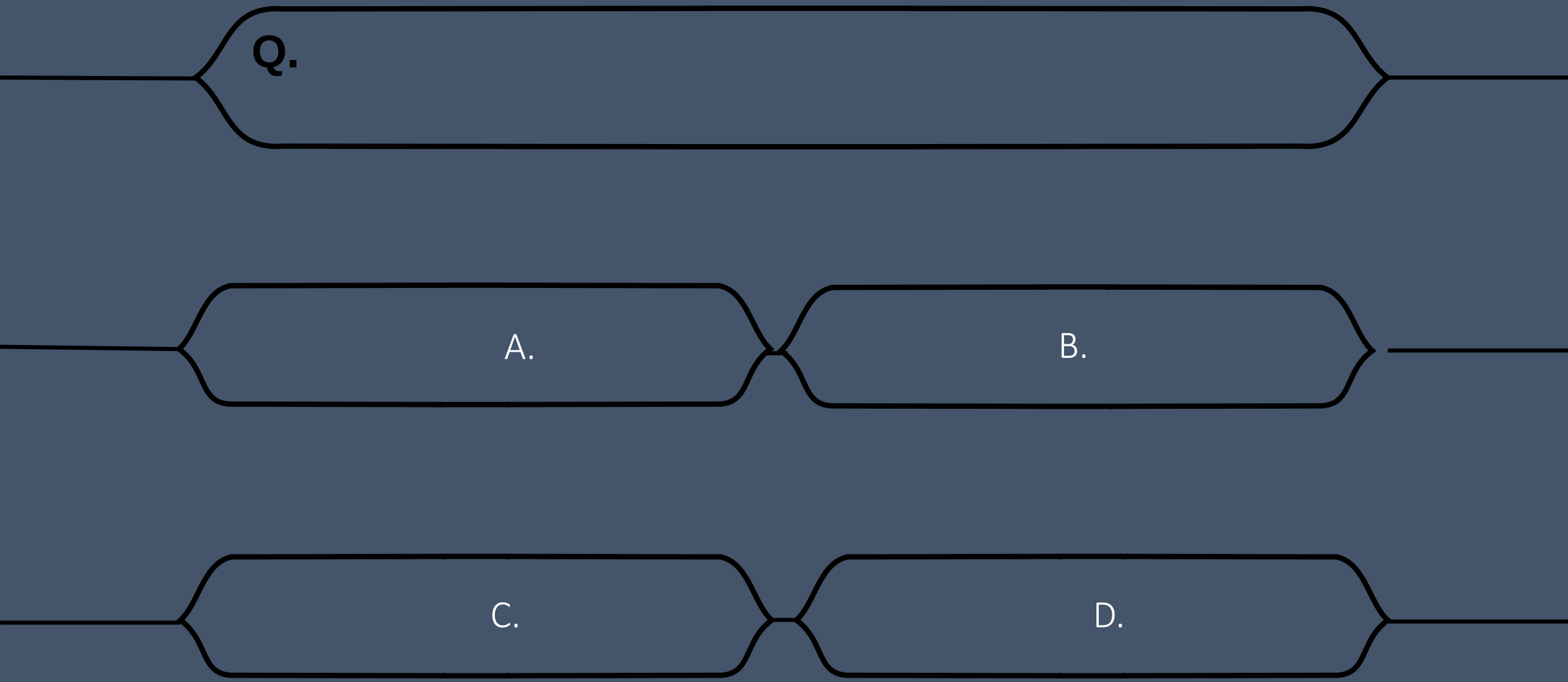

Q.
A.
B.
C.
D.

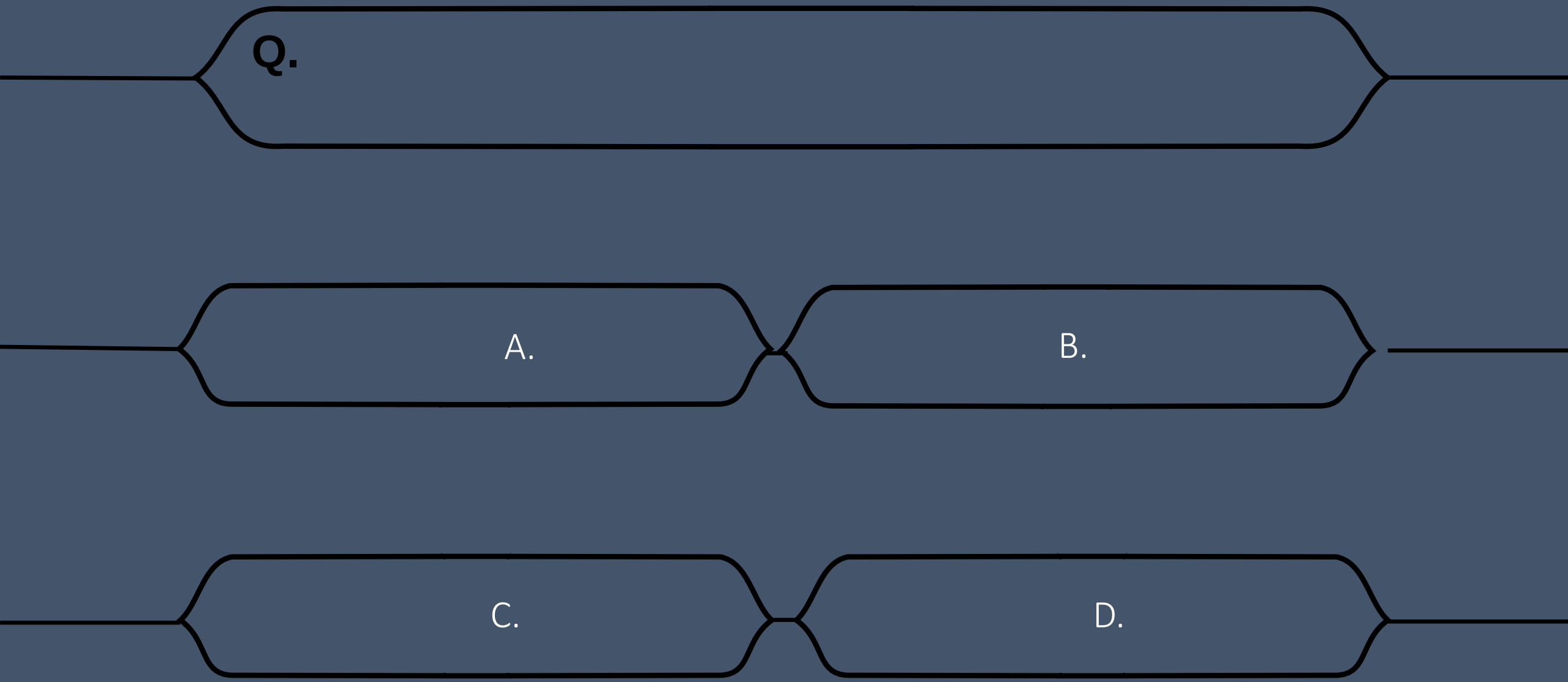

Q.
A.
B.
C.
D.

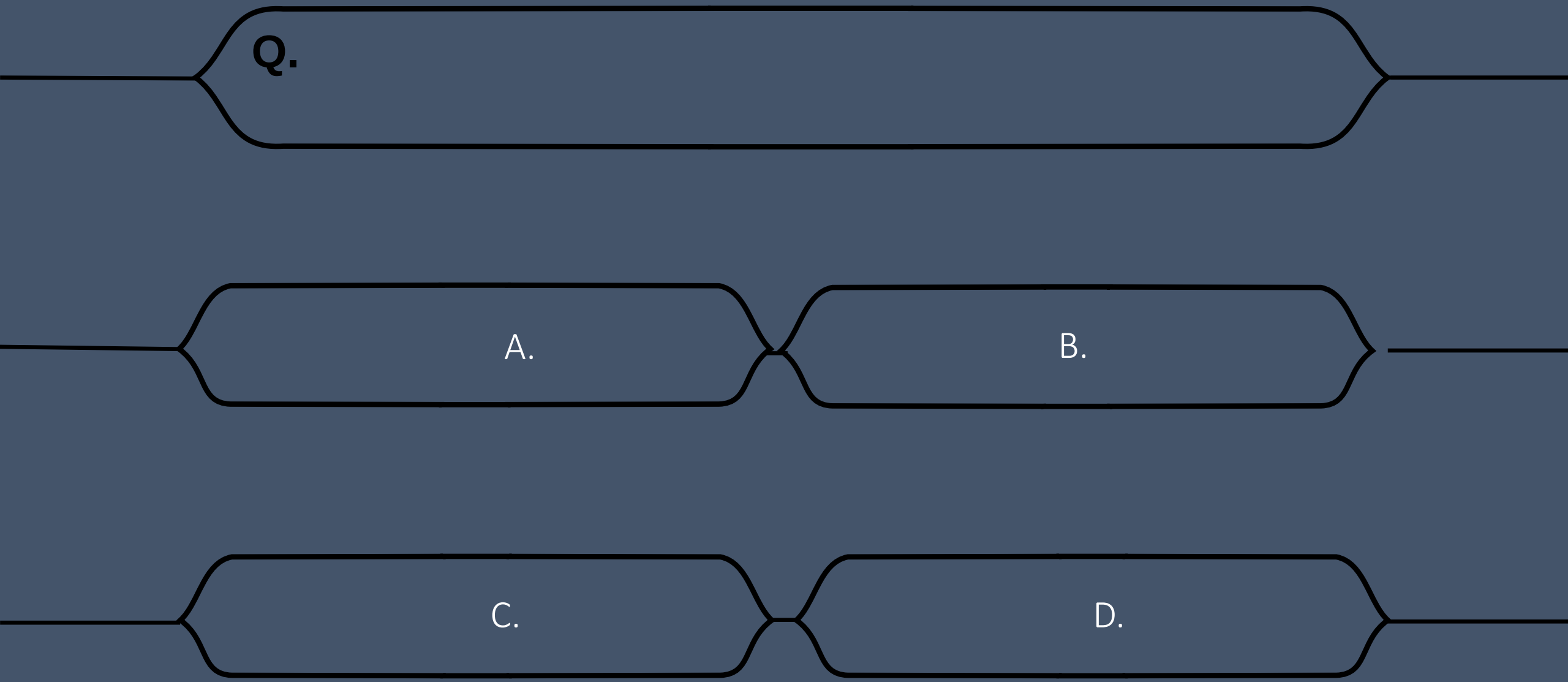

Q.
A.
B.
C.
D.

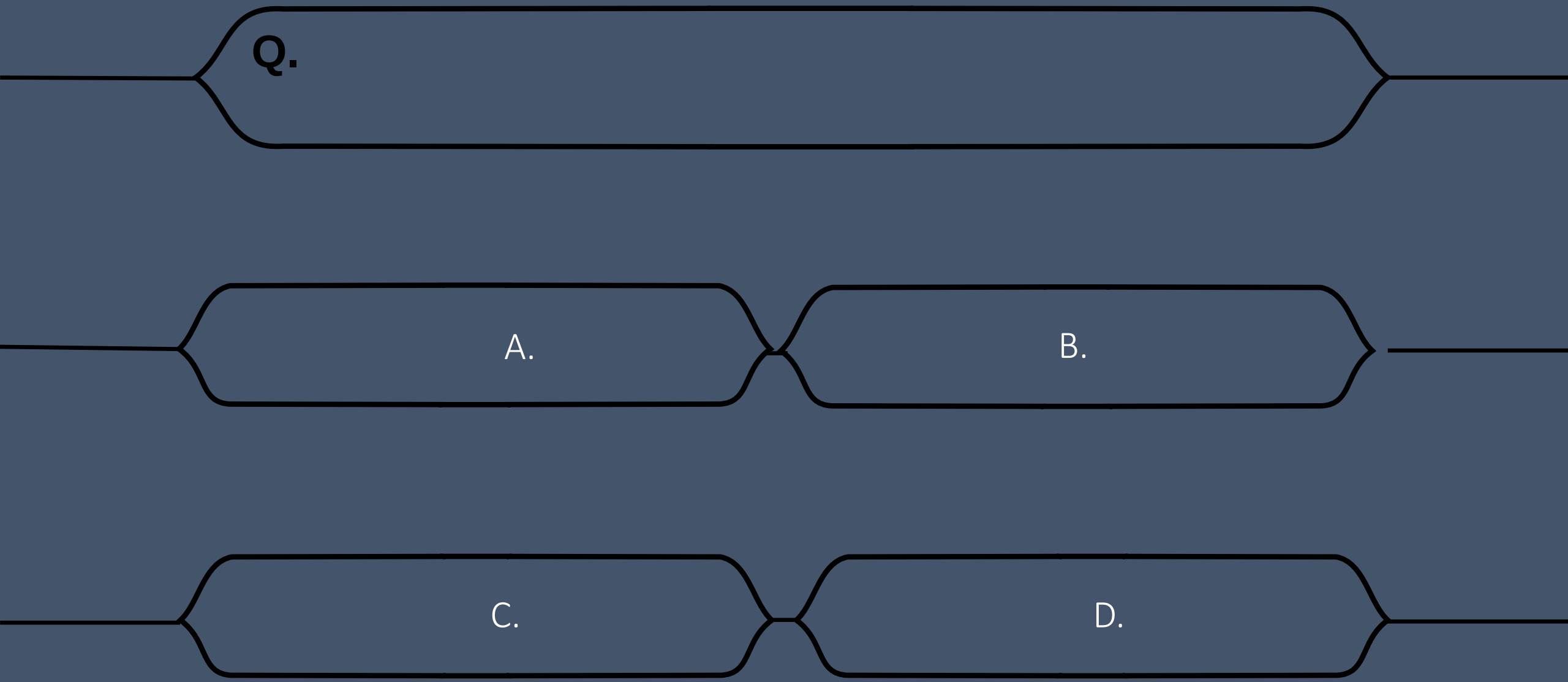

Q.
A.
B.
C.
D.

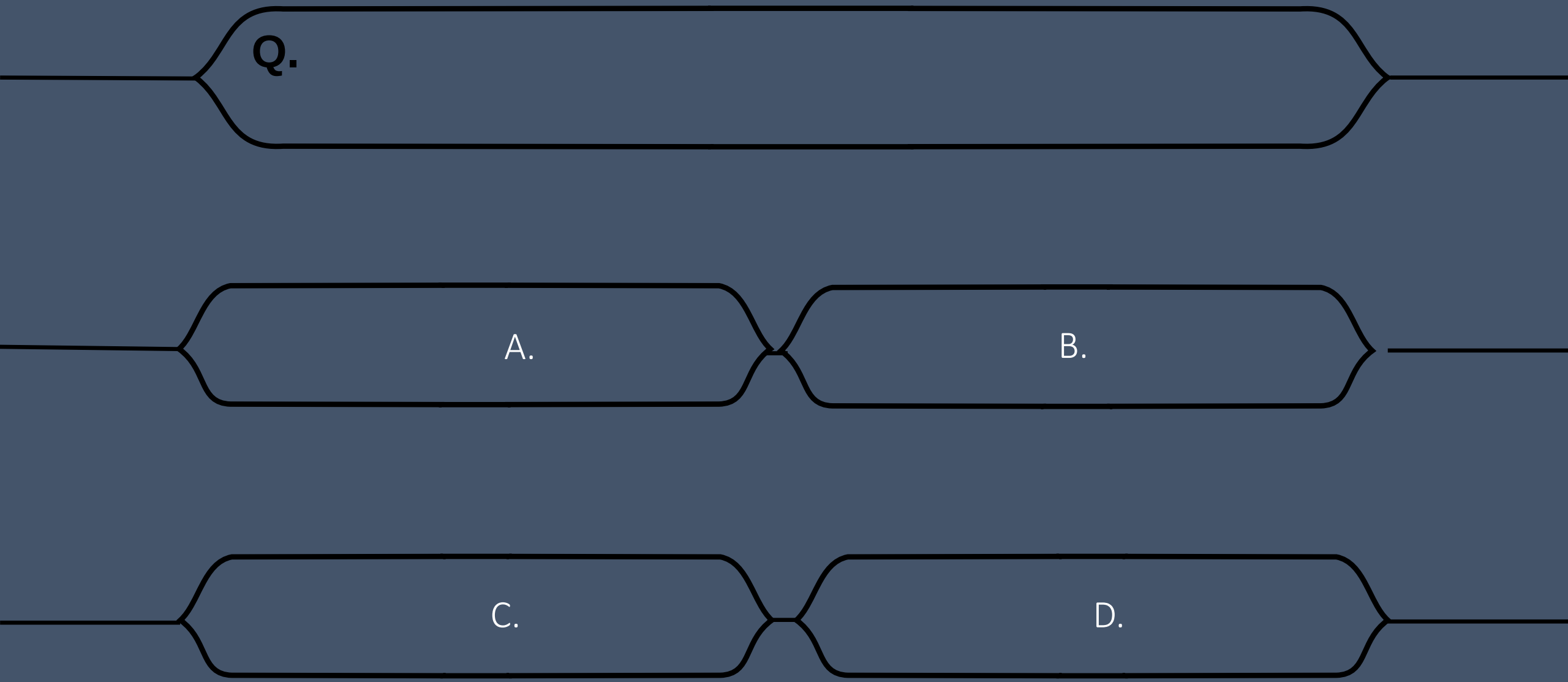

Q.
A.
B.
C.
D.

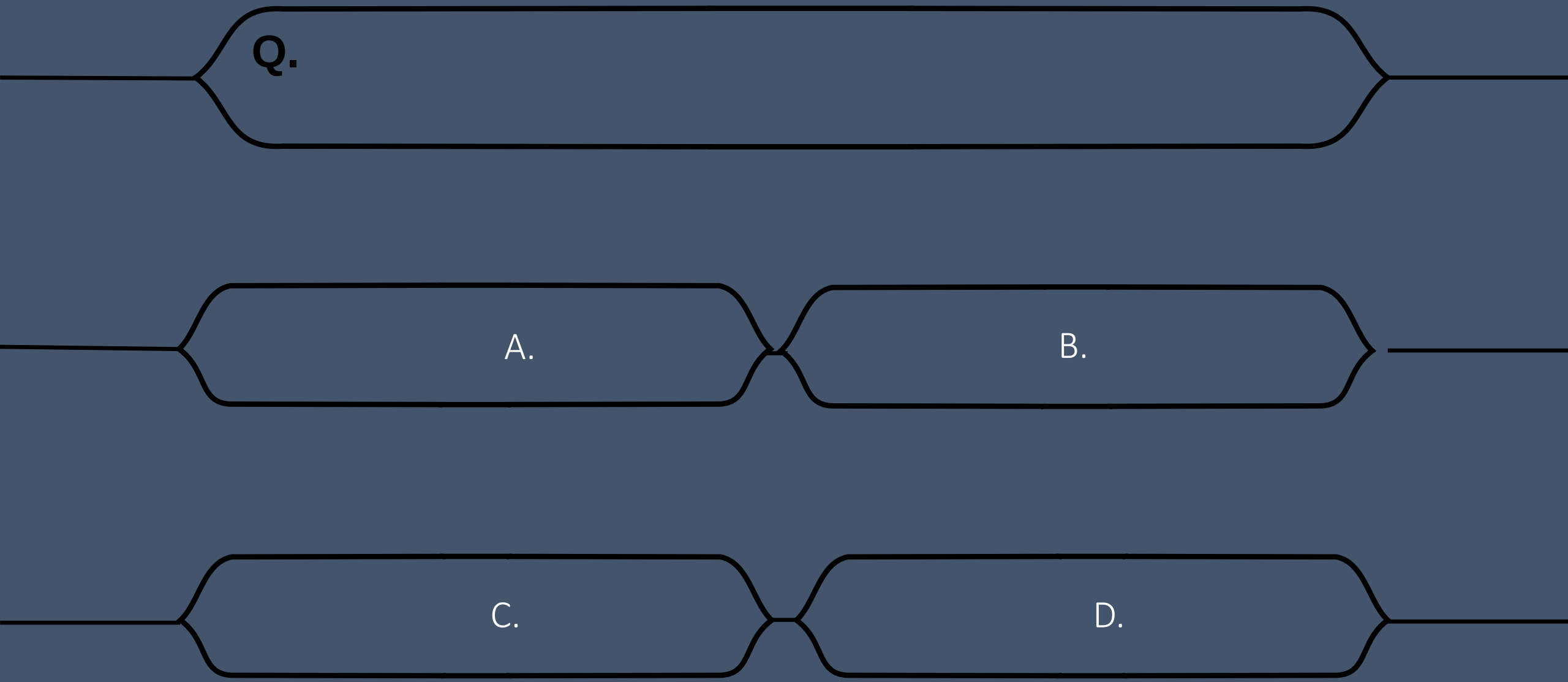

Q.
A.
B.
C.
D.

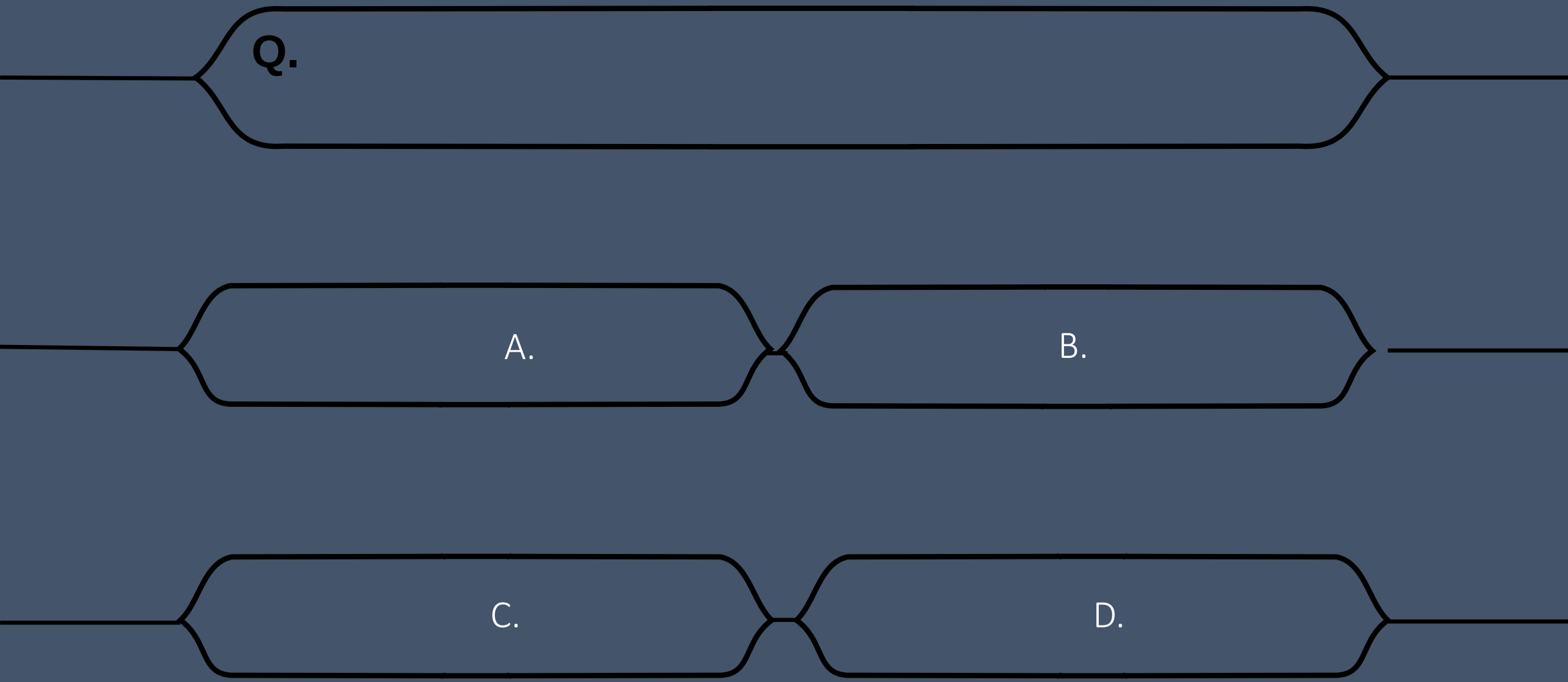

Q.
A.
B.
C.
D.

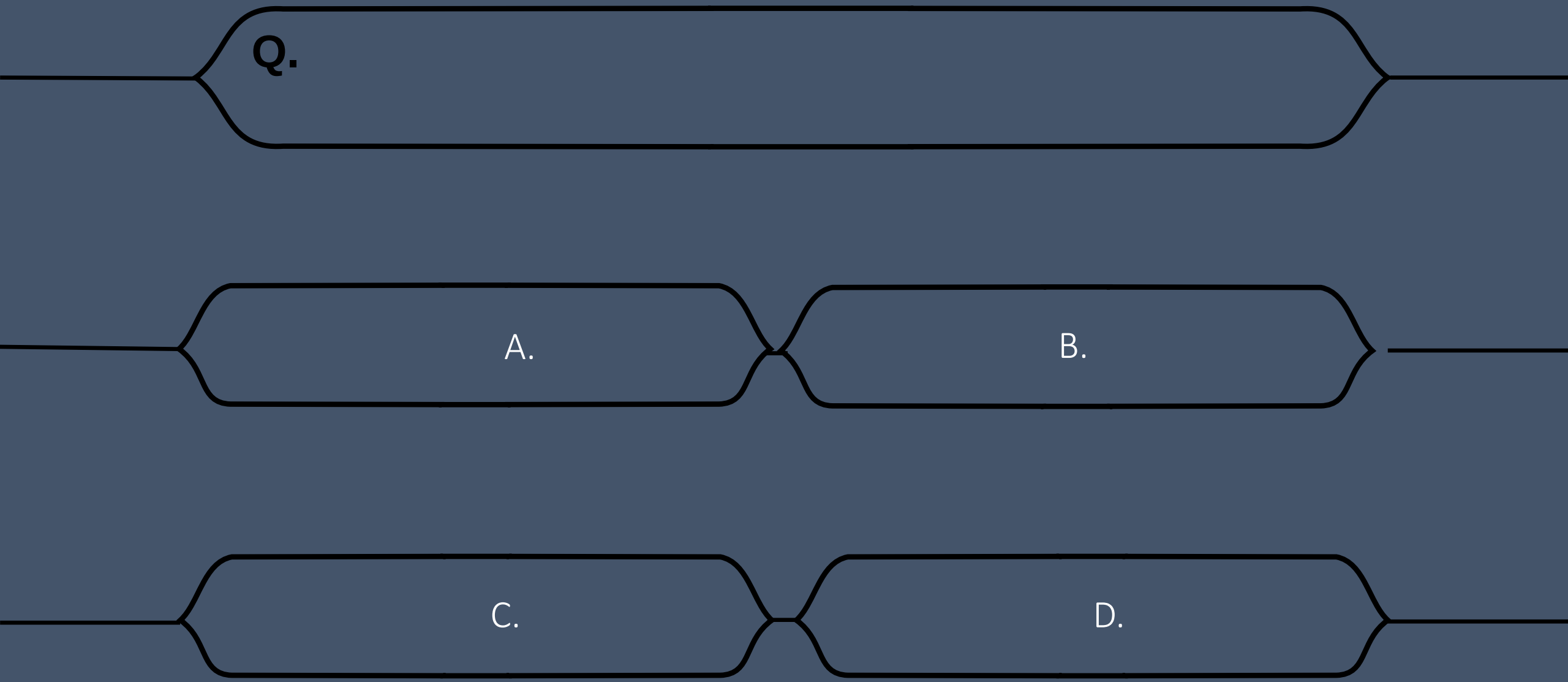

Q.
A.
B.
C.
D.

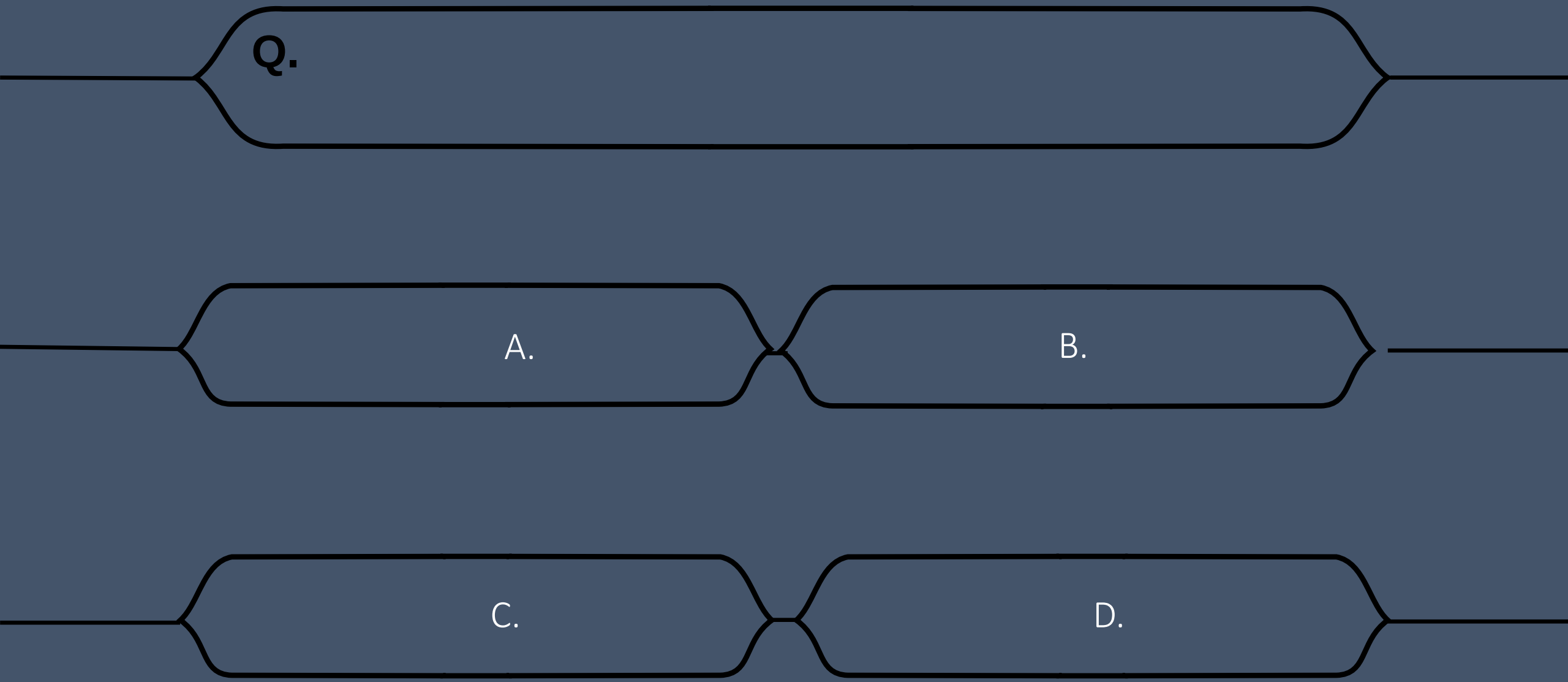

Q.
A.
B.
C.
D.

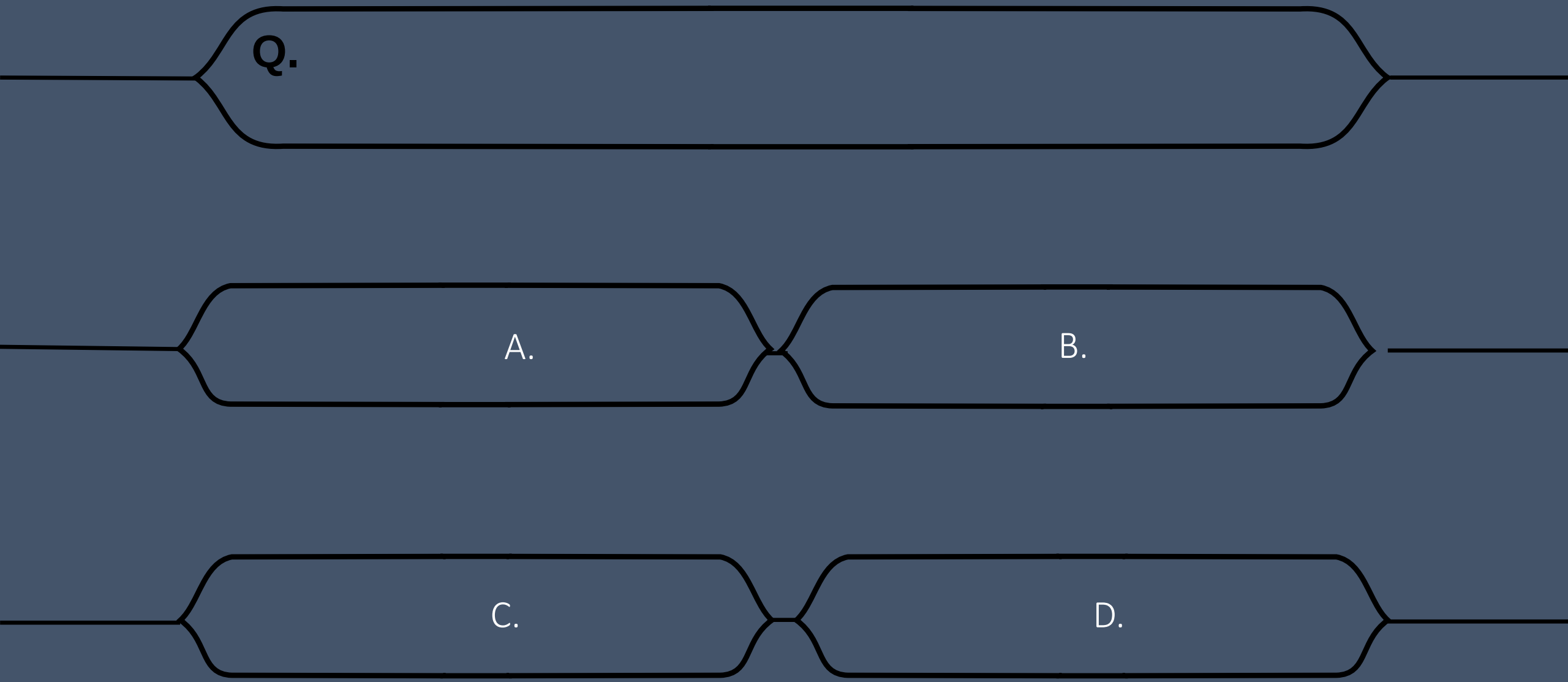

Q.
A.
B.
C.
D.

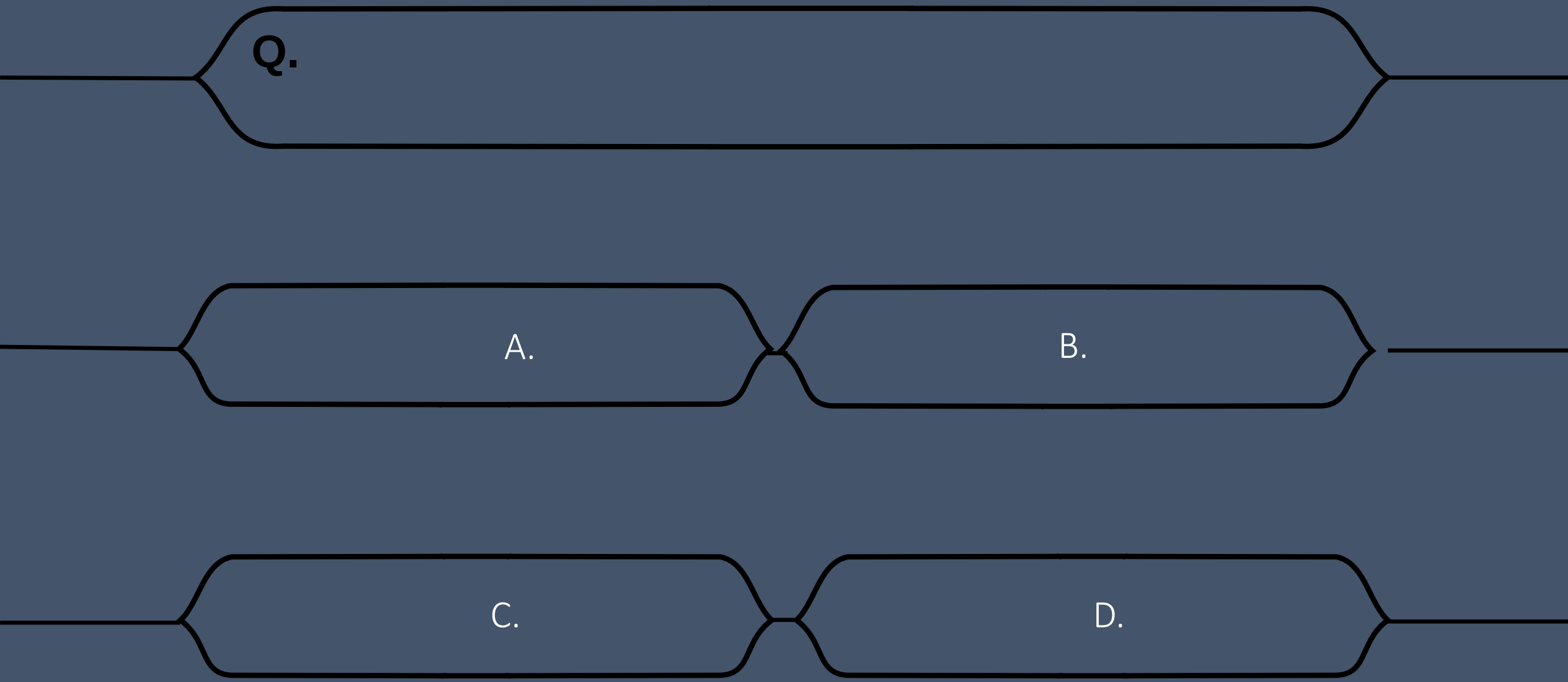

Q.
A.
B.
C.
D.

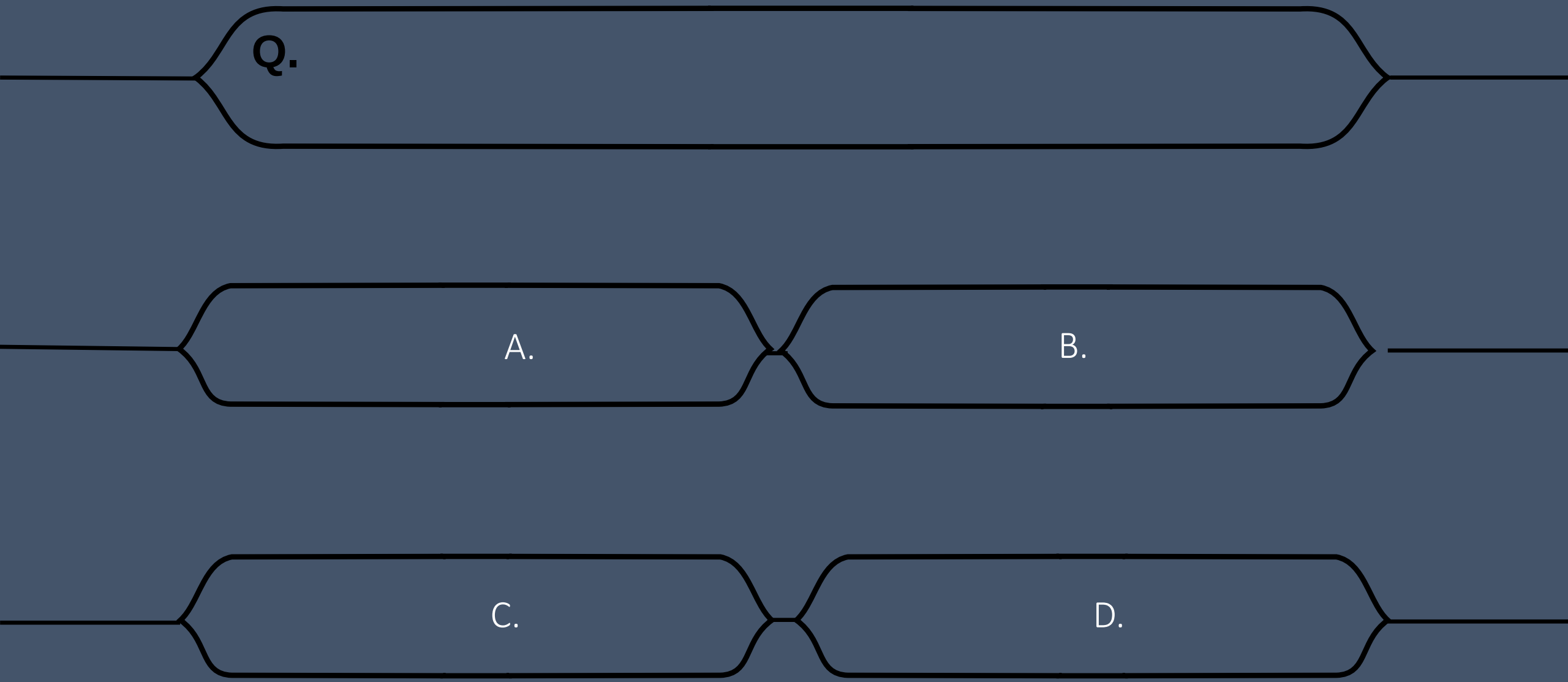

Q.
A.
B.
C.
D.

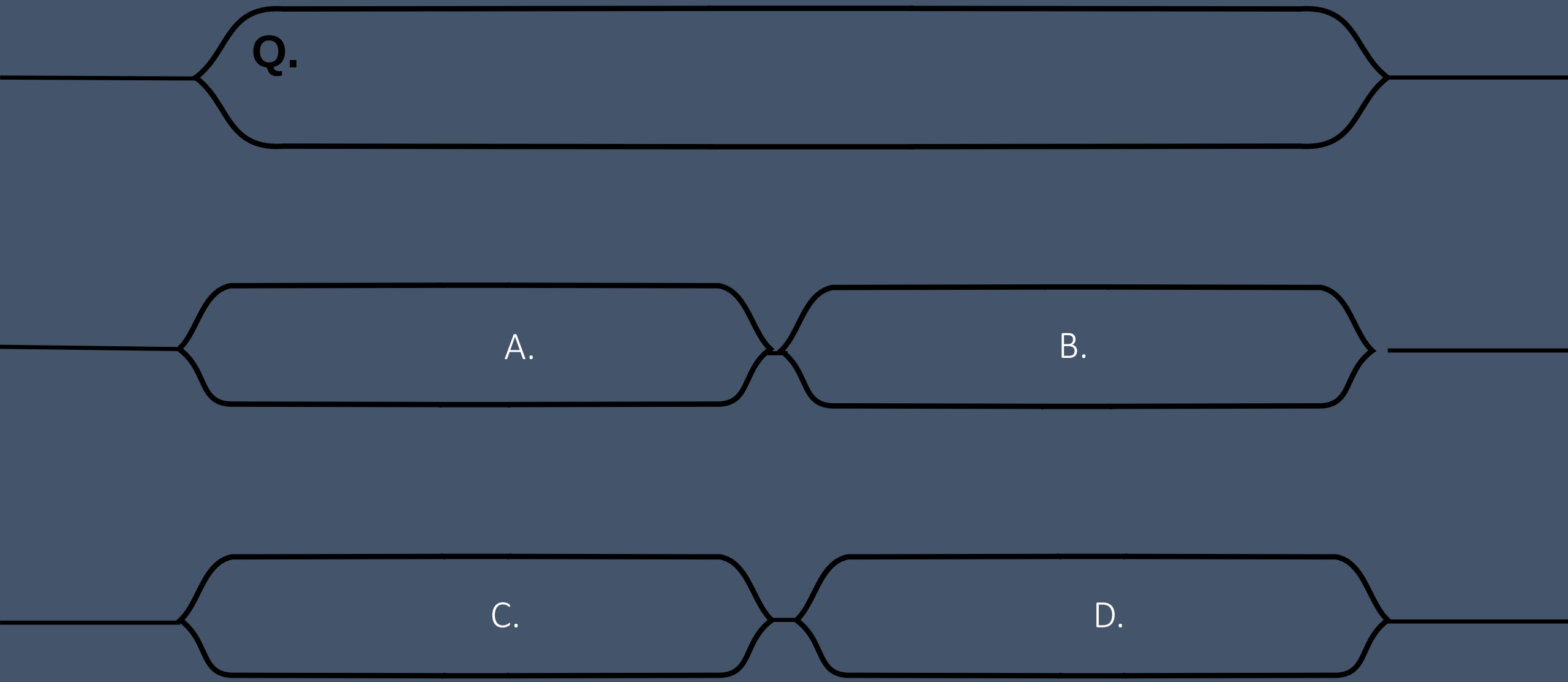

Q.
A.
B.
C.
D.

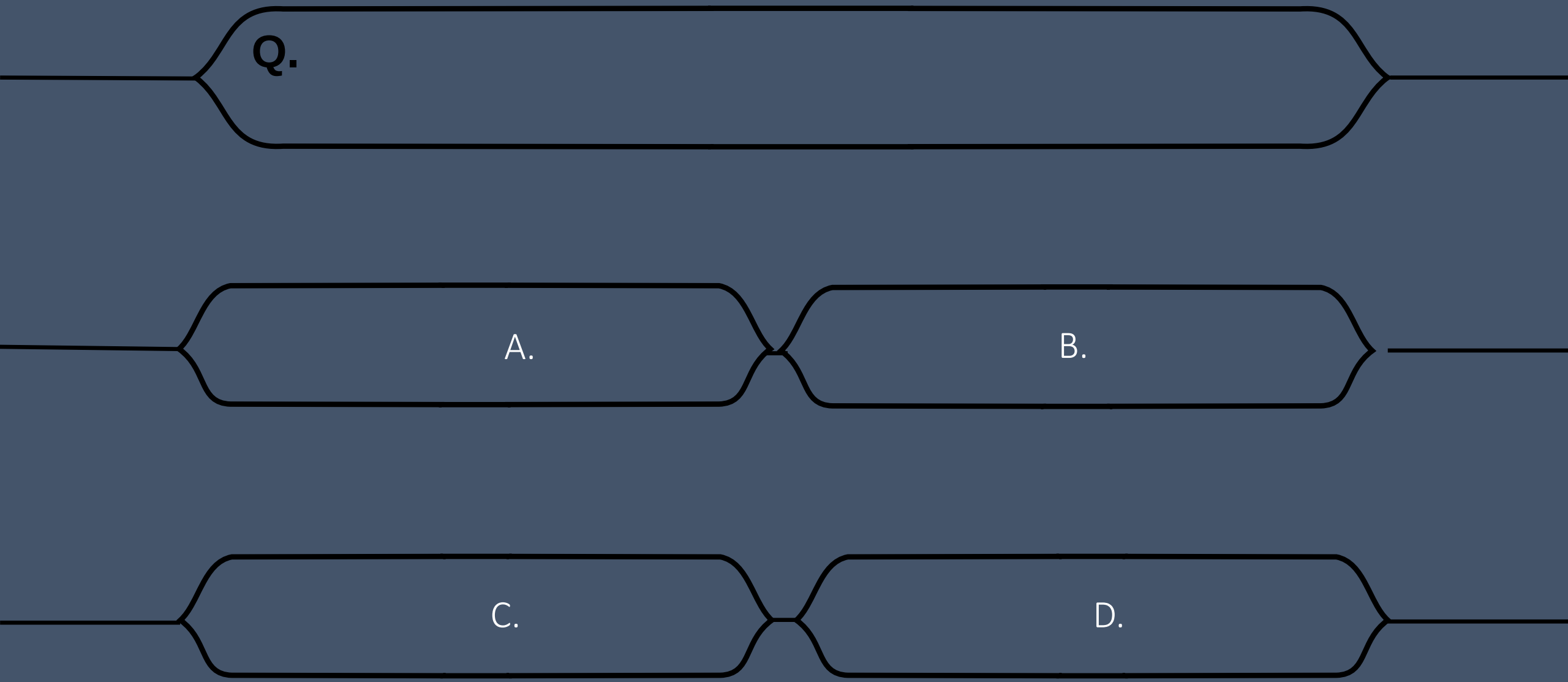

Q.
A.
B.
C.
D.

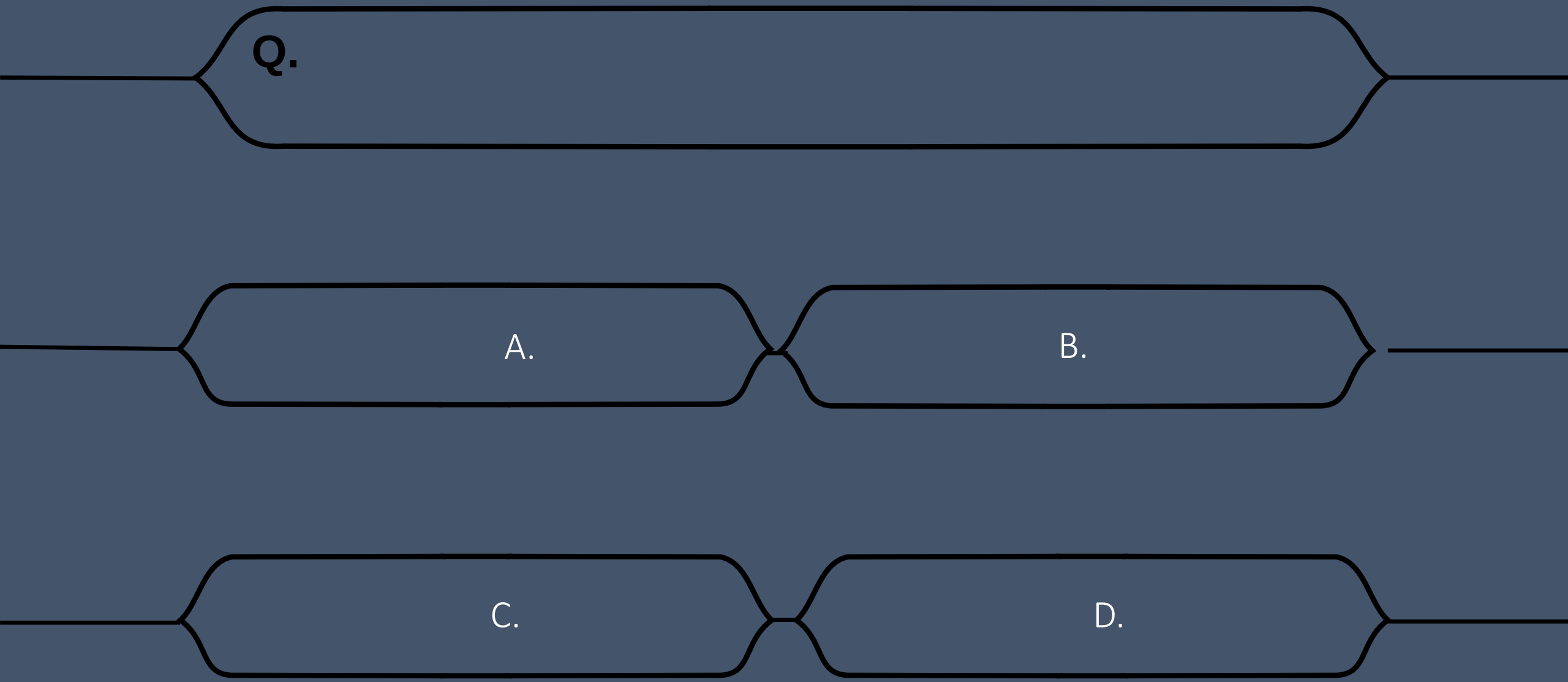

Q.
A.
B.
C.
D.

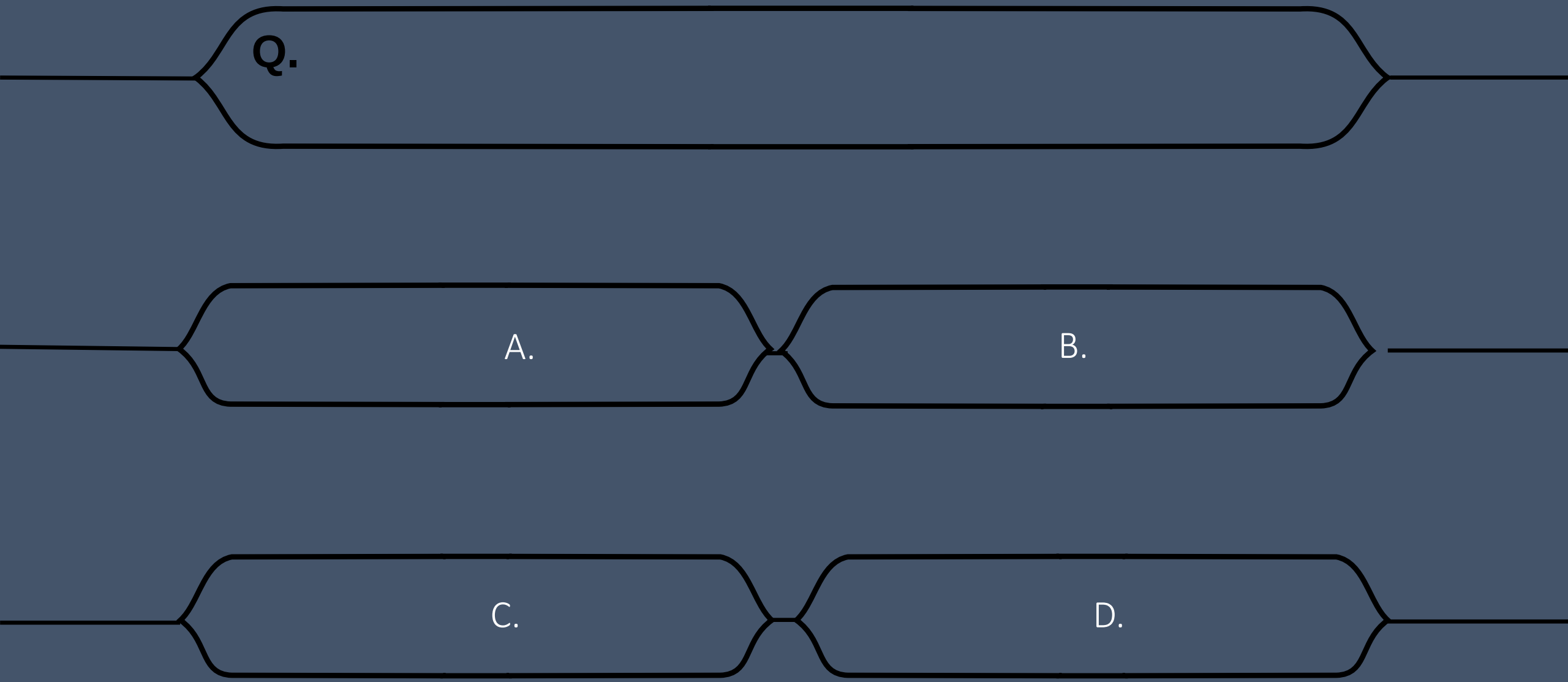

Q.
A.
B.
C.
D.

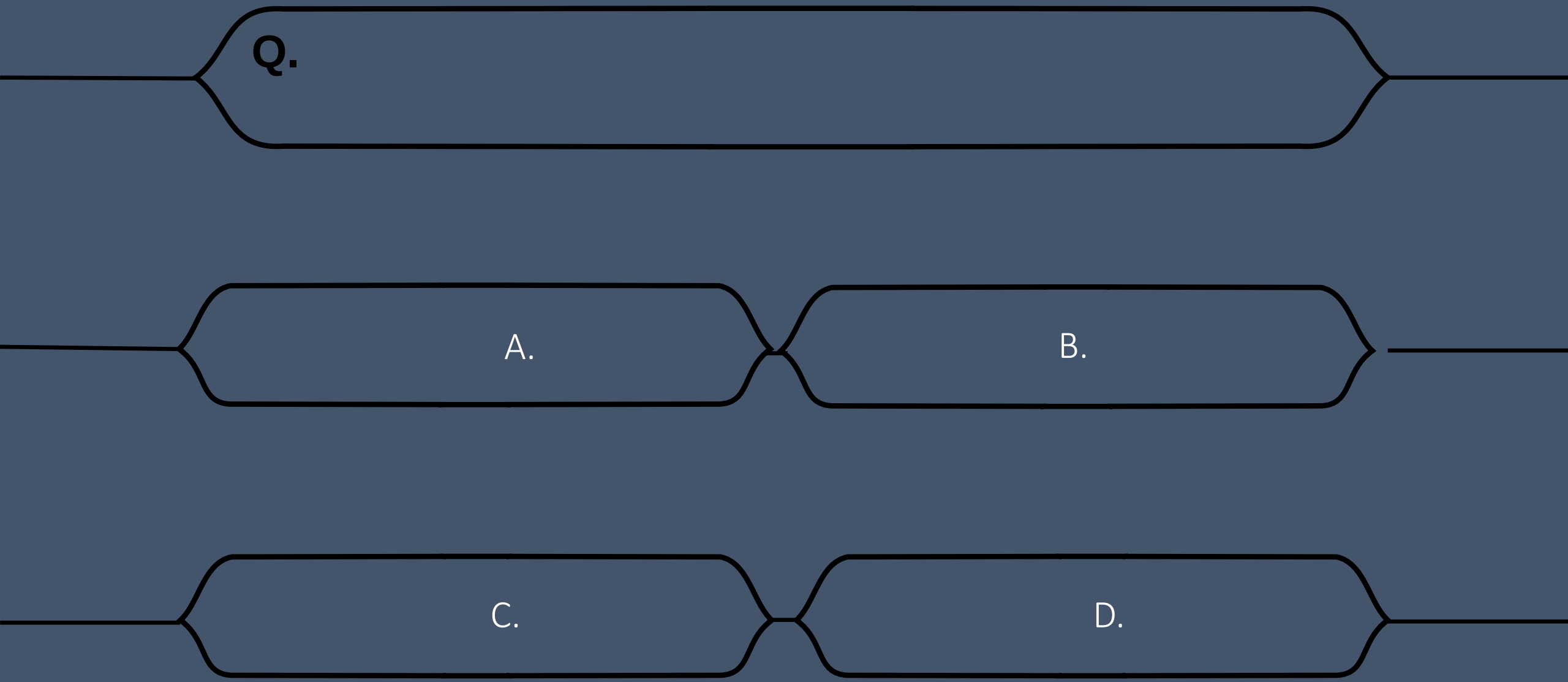

Q.
A.
B.
C.
D.

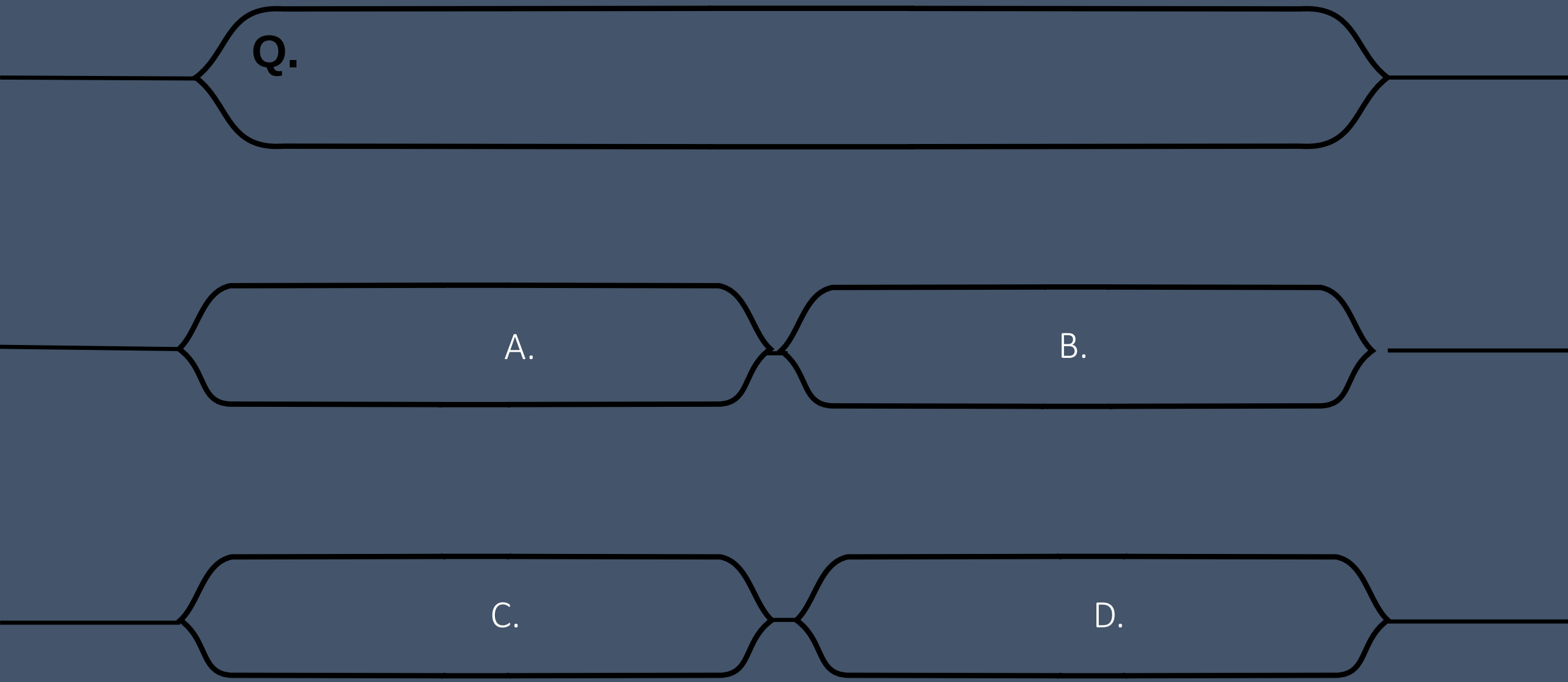

Q.
A.
B.
C.
D.

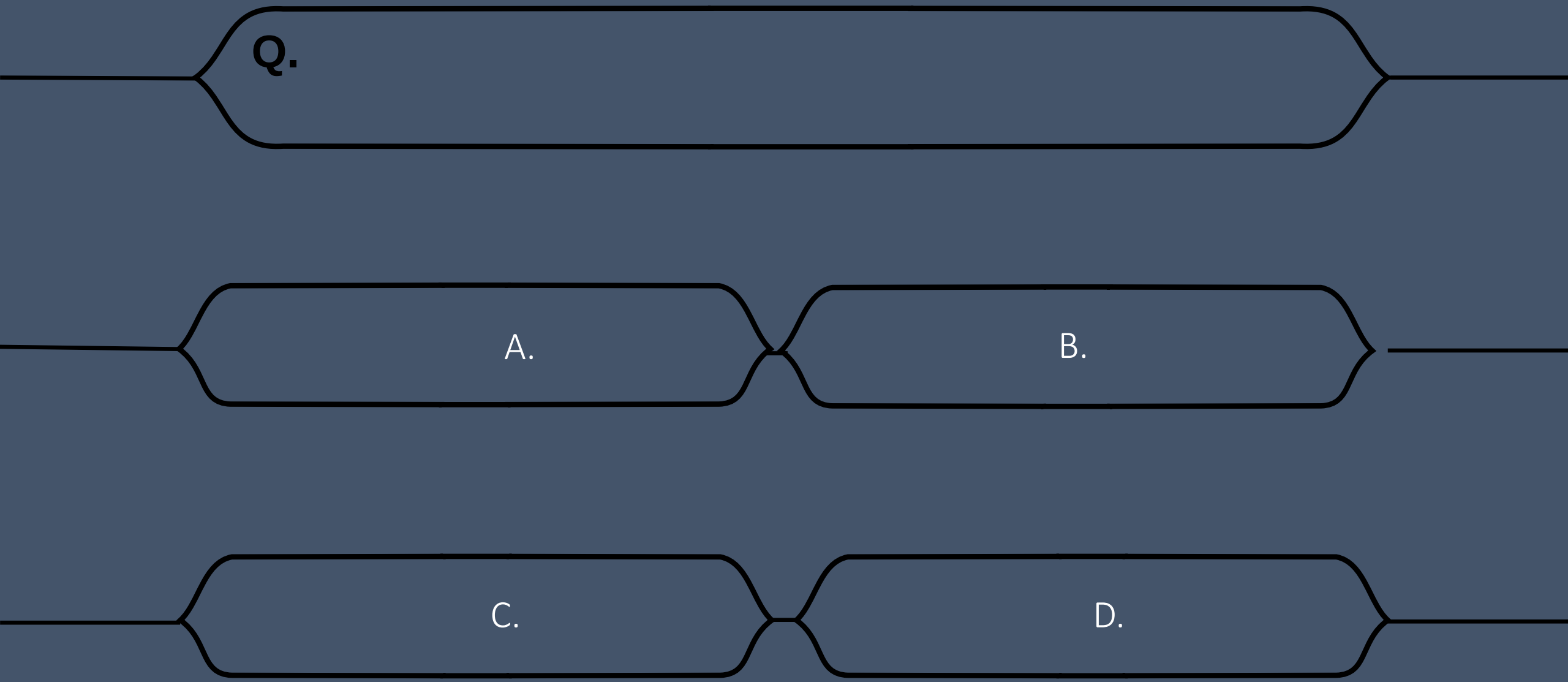

Q.
A.
B.
C.
D.

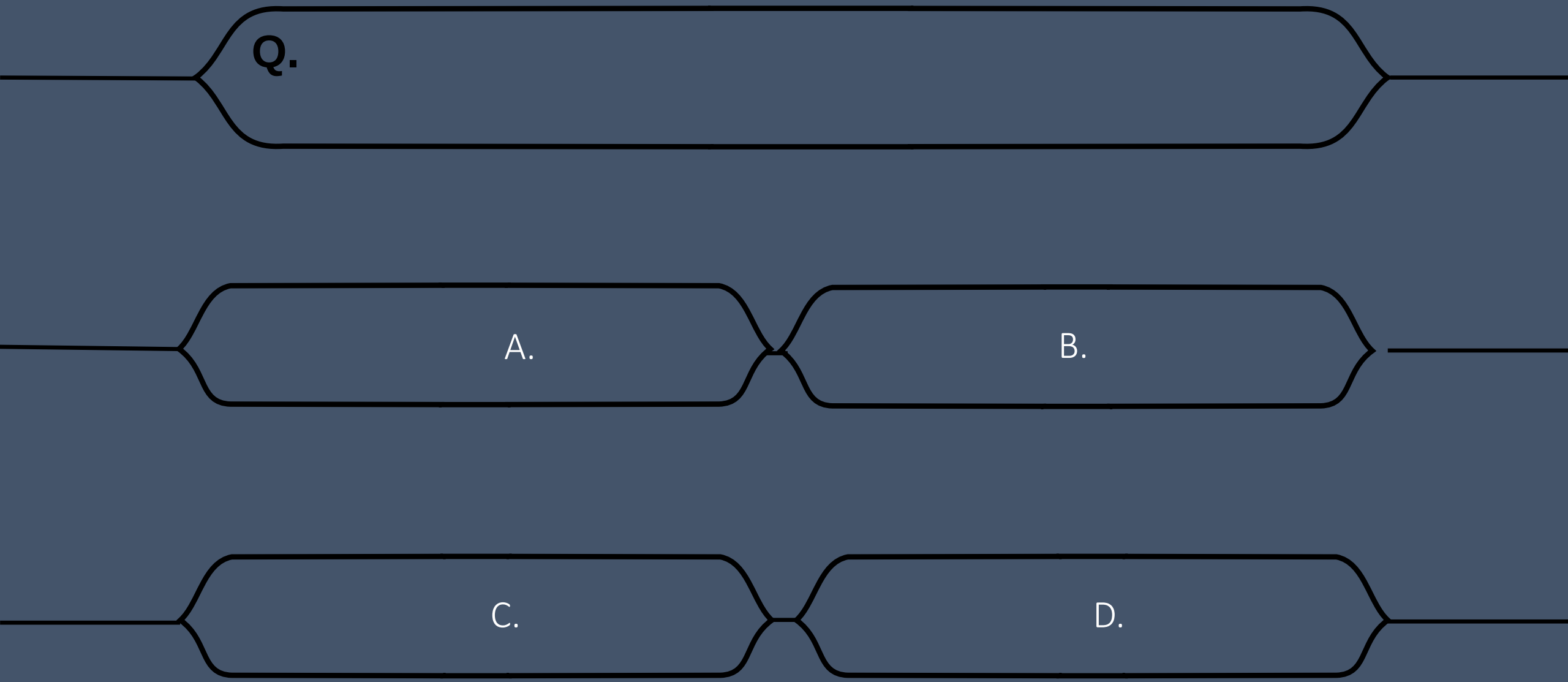

Q.
A.
B.
C.
D.

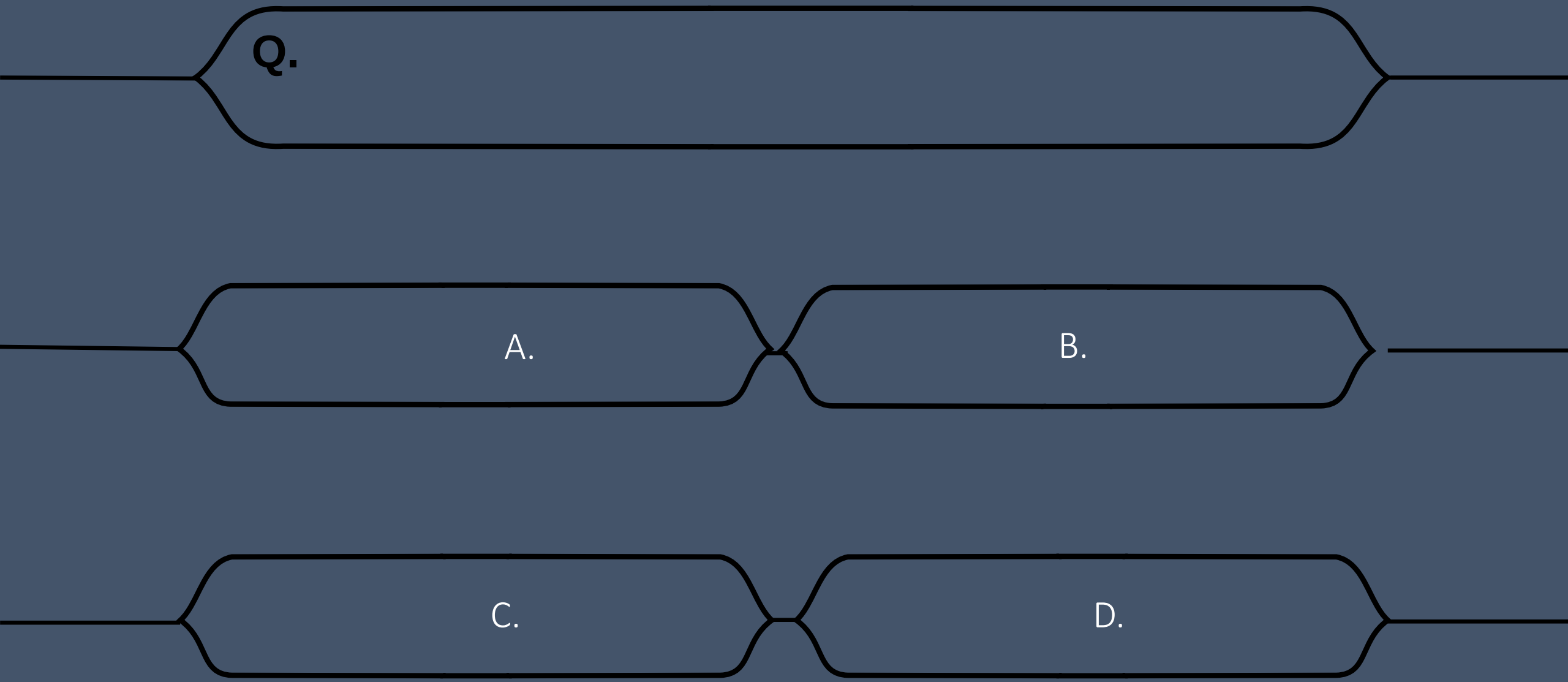

Q.
A.
B.
C.
D.

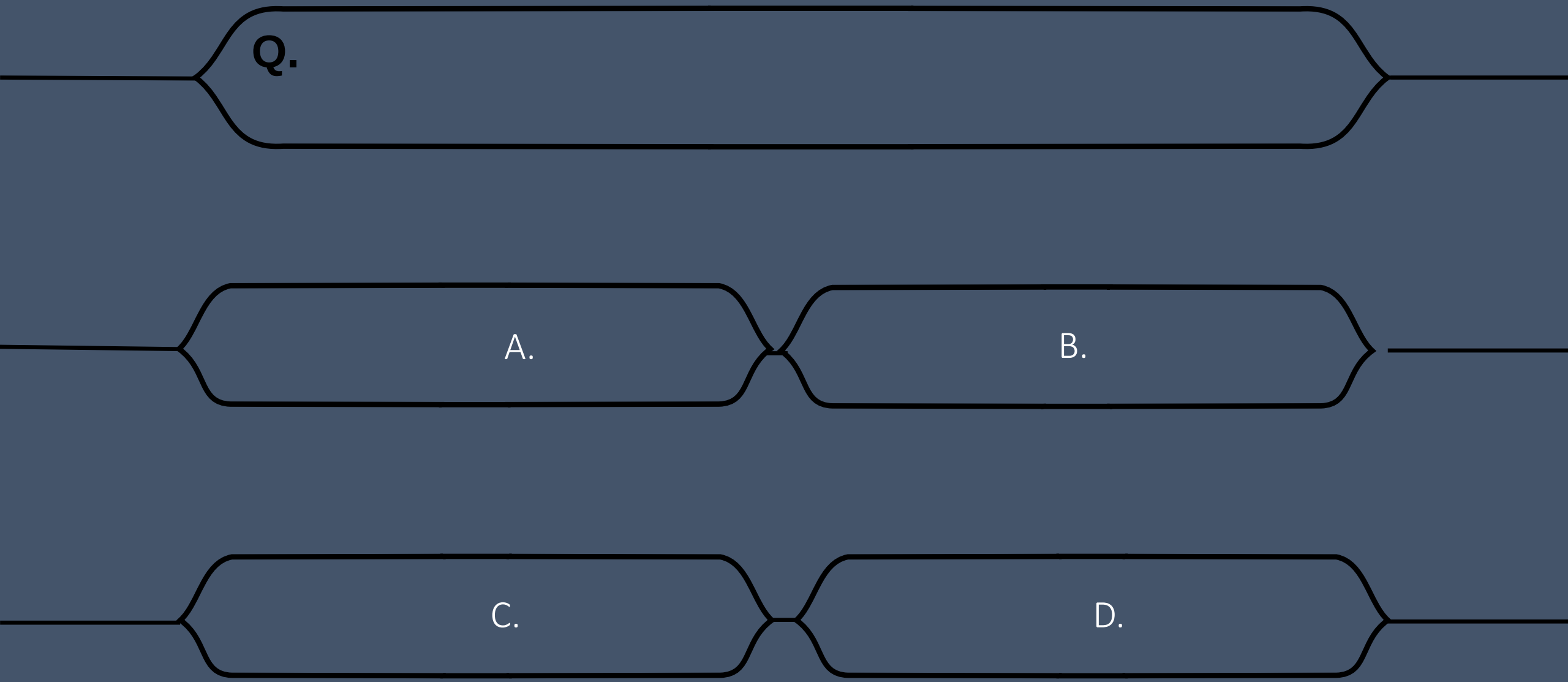

Q.
A.
B.
C.
D.

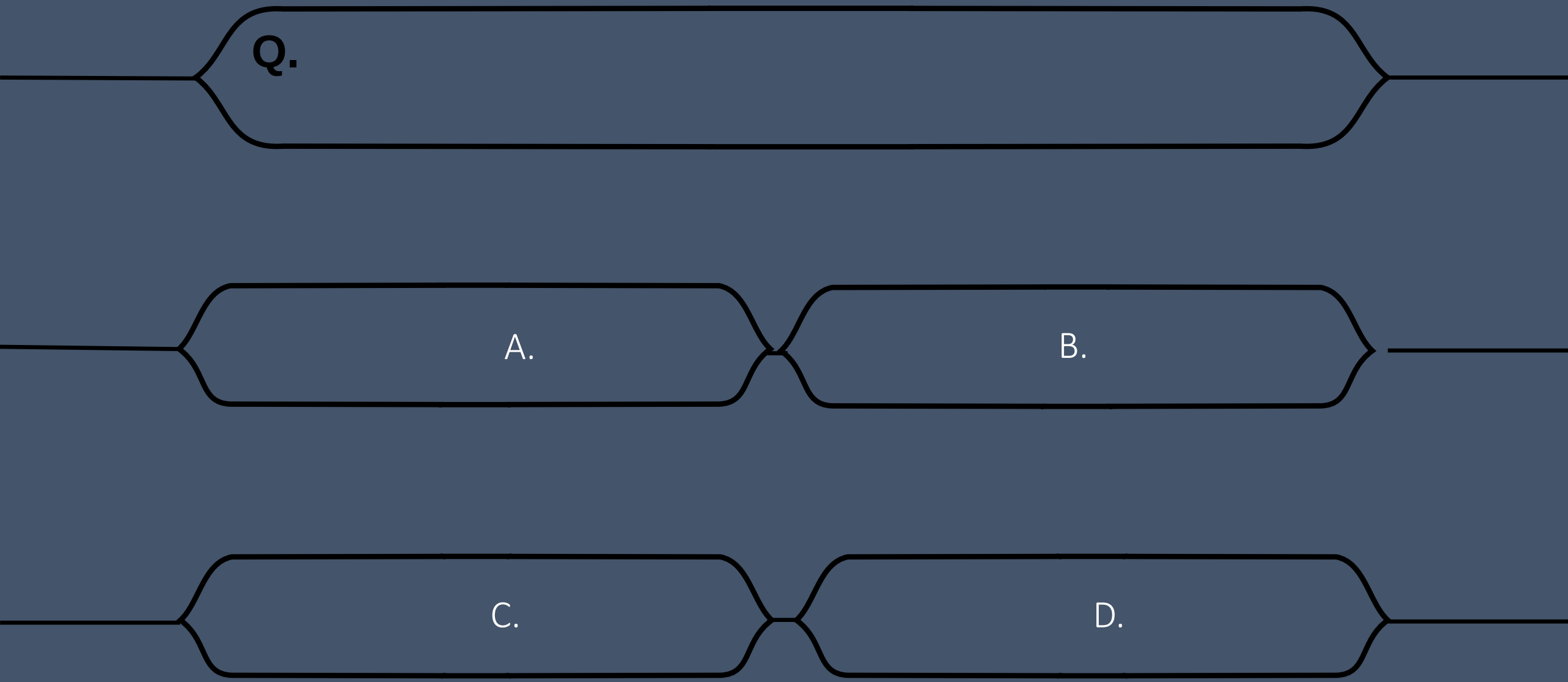

Q.
A.
B.
C.
D.

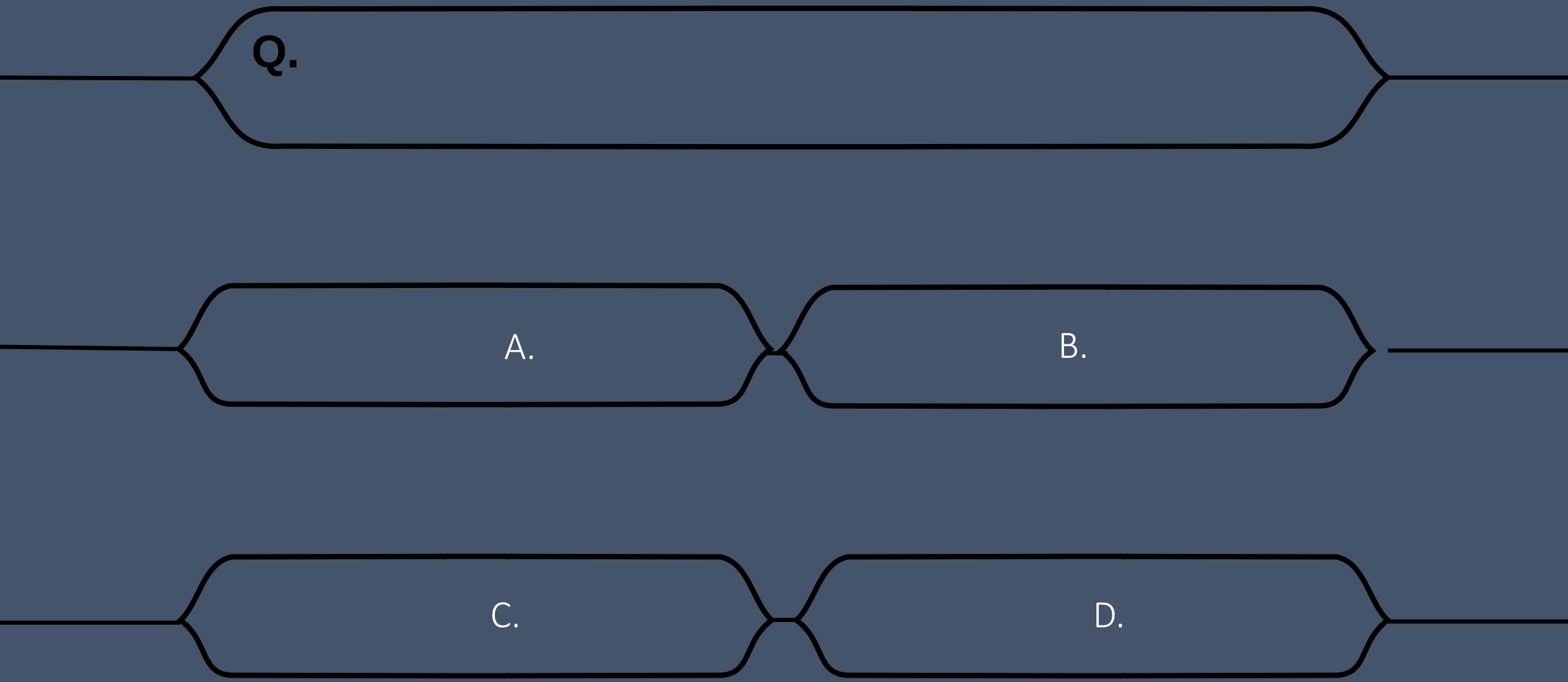

Q.
A.
B.
C.
D.


Q.
A.
B.
C.
D.


Q.
A.
B.
C.
D.


Q.
A.
B.
C.
D.


Q.
A.
B.
C.
D.


Q.
A.
B.
C.
D.