CS 303 Algorithms and Data Structures
Final Exam

105 points. 150 minutes. **Closed book/notes.**

Name: _____

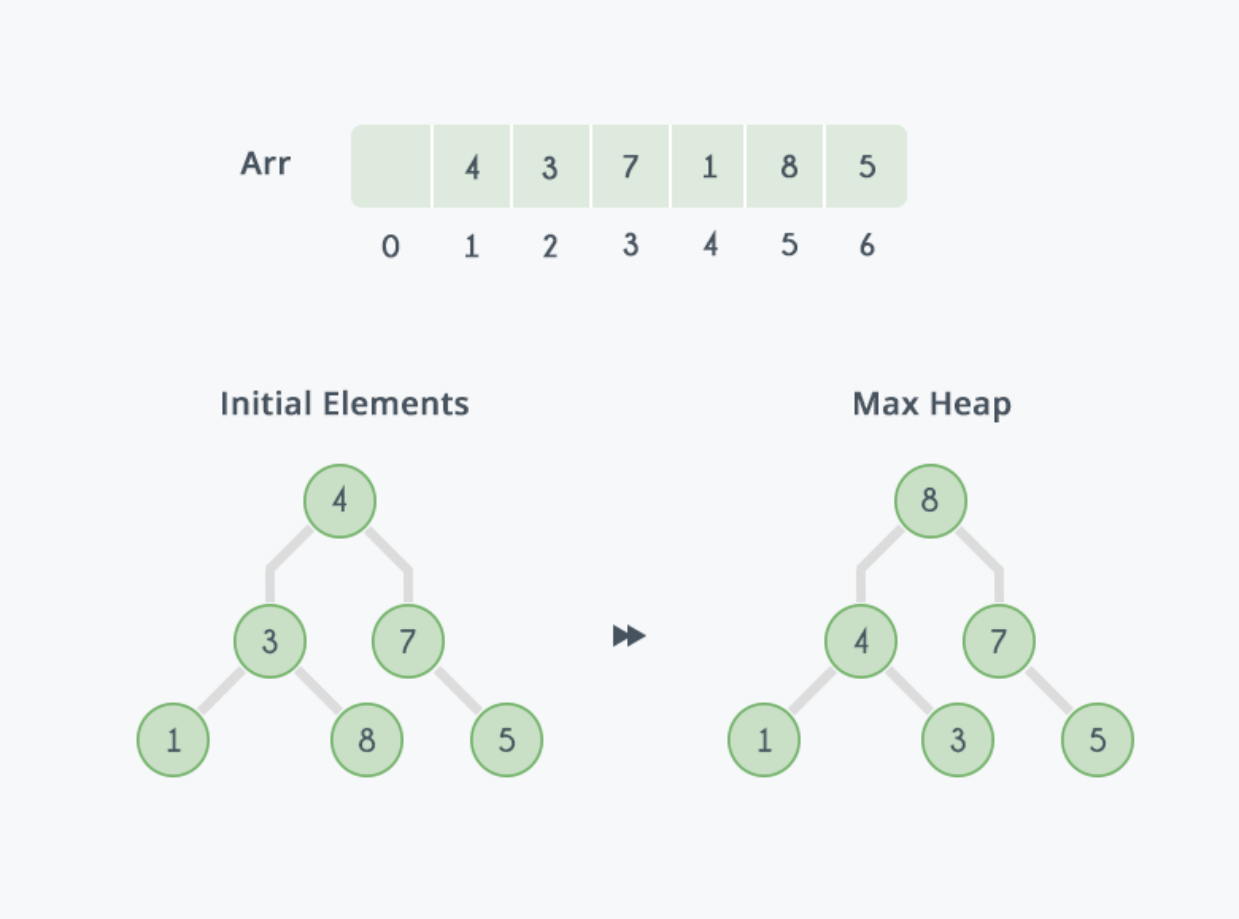
1. Illustrate the operation of sorting the sequence **4, 3, 7, 1, 8, 5** using **heapsort**. **[10 points]**

   The heapsort algorithm is given below.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)

BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```


Arr
4 3 7 1 8 5
0 1 2 3 4 5 6
Initial Elements
Max Heap
4
3 7
1 8 5
8
4 7
1 3 5


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

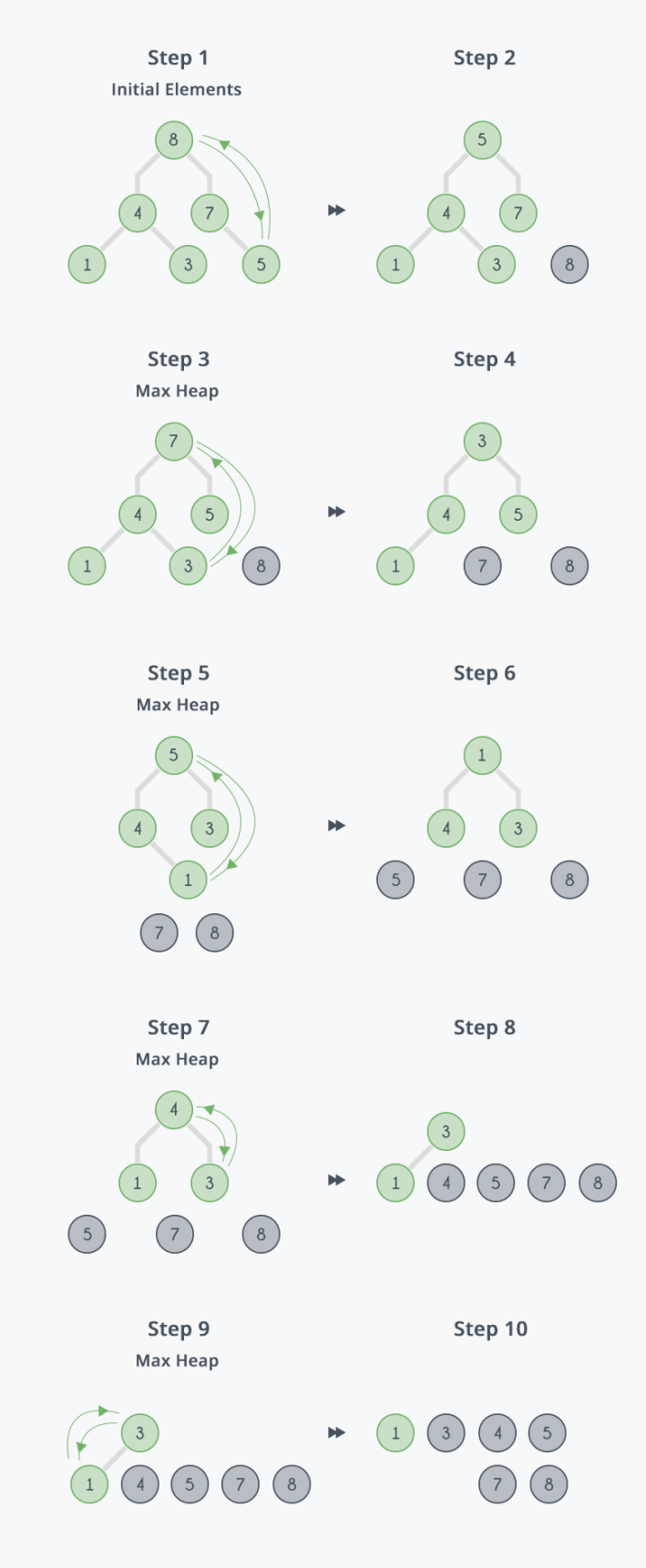
- Heap sort is a **comparison-based algorithm based on the binary heap** (complete binary tree) data structure, where **every level of the tree contains values that are stored in a defined and definitive order**, such that the value in the parent nodes must be greater or smaller than those in the two child nodes.

- Heap sort is **not stable** because operations in the heap can **change the relative order of equivalent keys**.

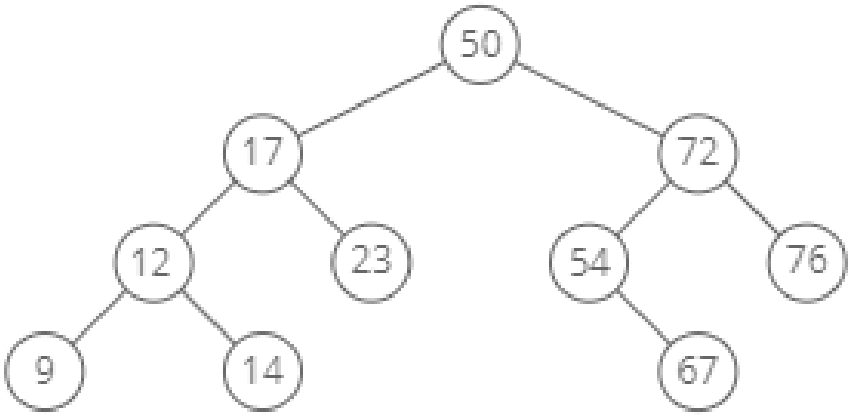
- The binary heap **can be represented using array-based methods to reduce space and memory usage**.

- Heap sort is an **in-place algorithm**, where inputs are overwritten using no extra data structures at runtime.

- The algorithm **divides the input unto sorted and unsorted regions** and **finds the largest element, placing it at the end of the sorted list**, repeating this procedure for the remaining elements in the unsorted list



2. Show what the tree would look like after the following changes are made to the following binary search tree (make sure to include all the intermediate steps and explain each step): **[12 points]**
   a) **Delete** node with value 17
   b) **Insert** node with value 56
   c) **Insert** node with value 88
   d) **Delete** node with value 72



The algorithm for deleting a node in a binary search tree is given below:

```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)          // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)           // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)          // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this subtree.
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

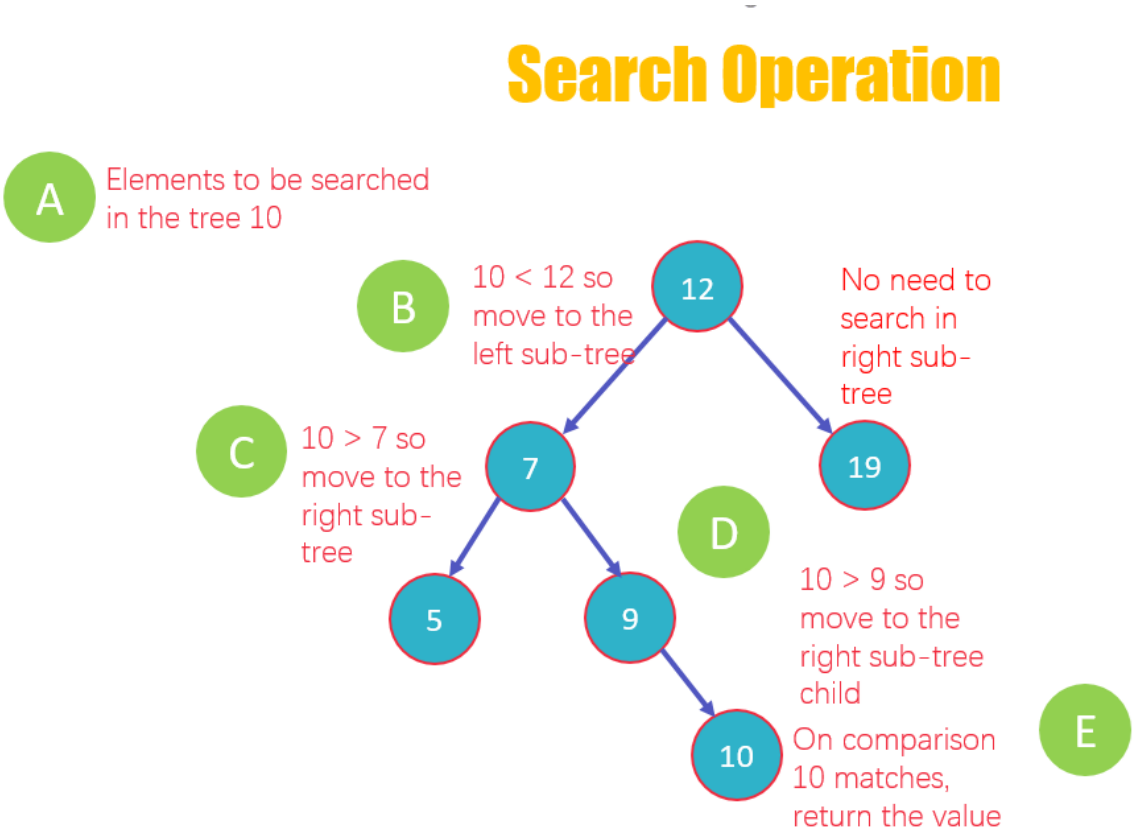
```
TRANSPLANT(T, u, v)
  if u.p == NIL
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  if v ≠ NIL
      v.p = u.p
```

An important special kind of binary tree is the **binary search tree (BST)**. In a BST, each node stores some information including a unique **key value**, and perhaps some associated data. A binary tree is a BST iff, for every node n in the tree:

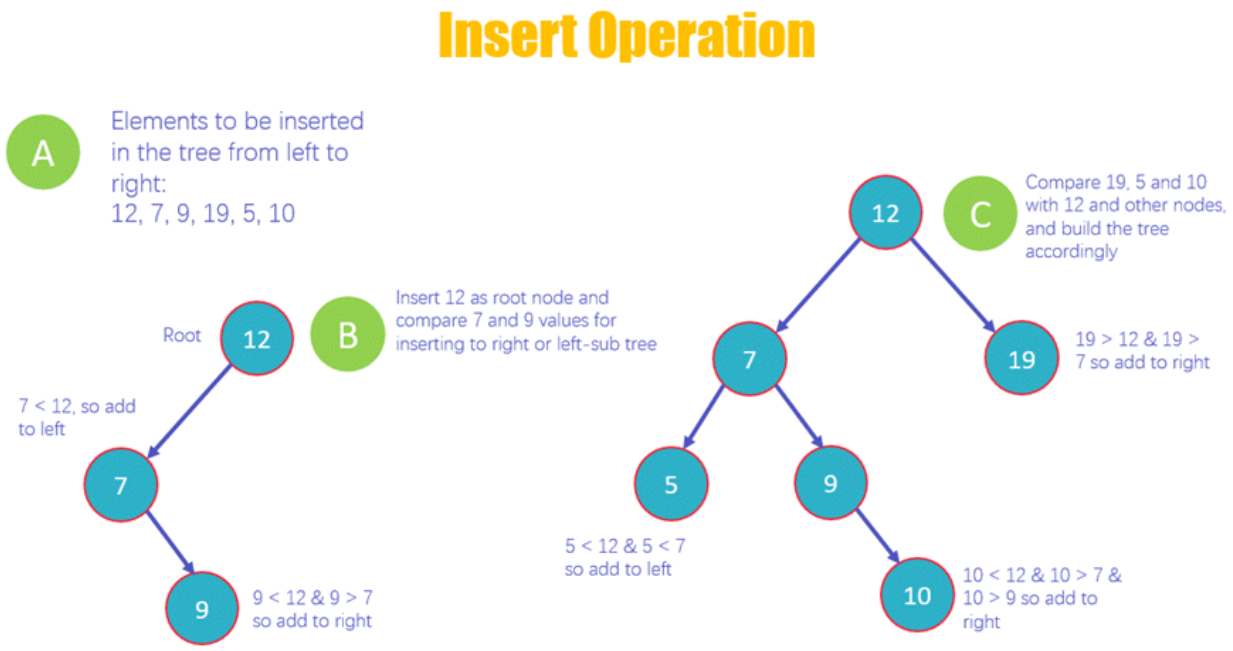
- All keys in n's left subtree are less than the key in n, and
- all keys in n's right subtree are greater than the key in n.

Note: if duplicate keys are allowed, then nodes with values that are equal to the key in node n can be either in n's left subtree or in its right subtree (but not both). In these notes, we will assume that duplicates are not allowed.

Search:

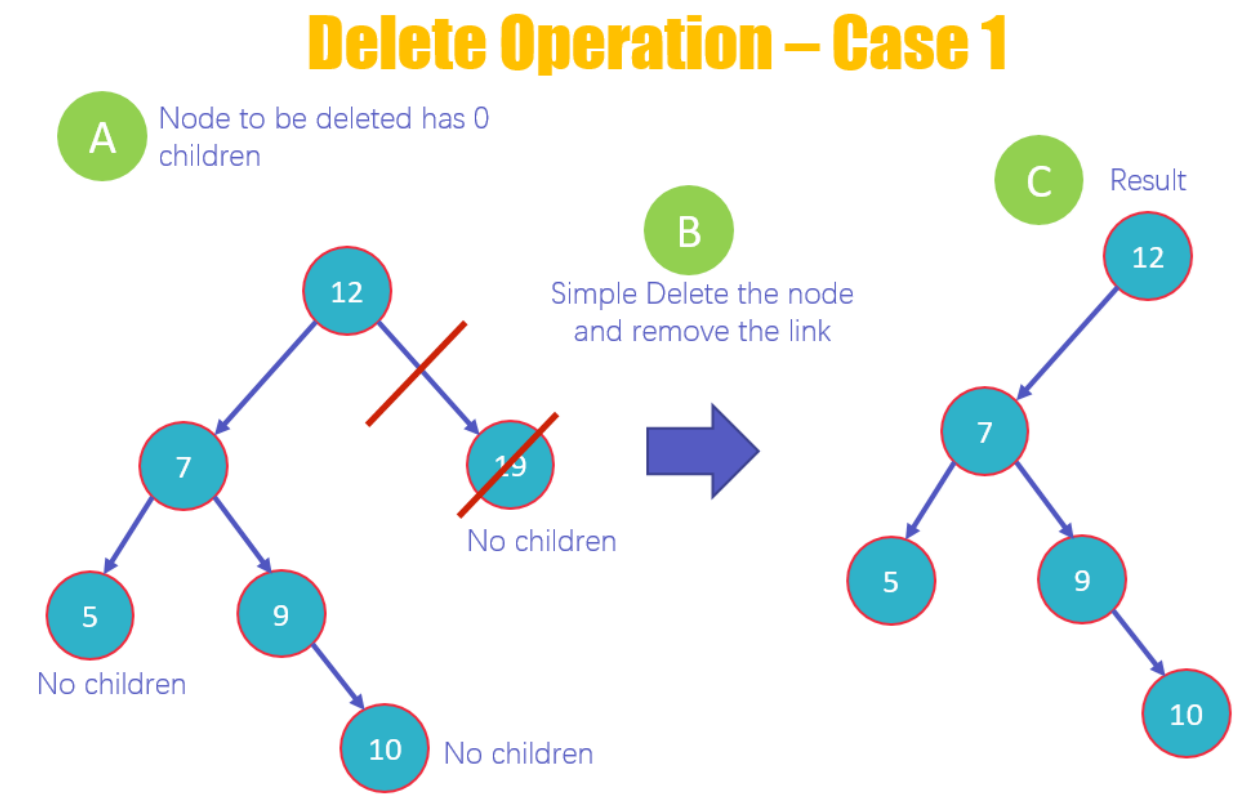


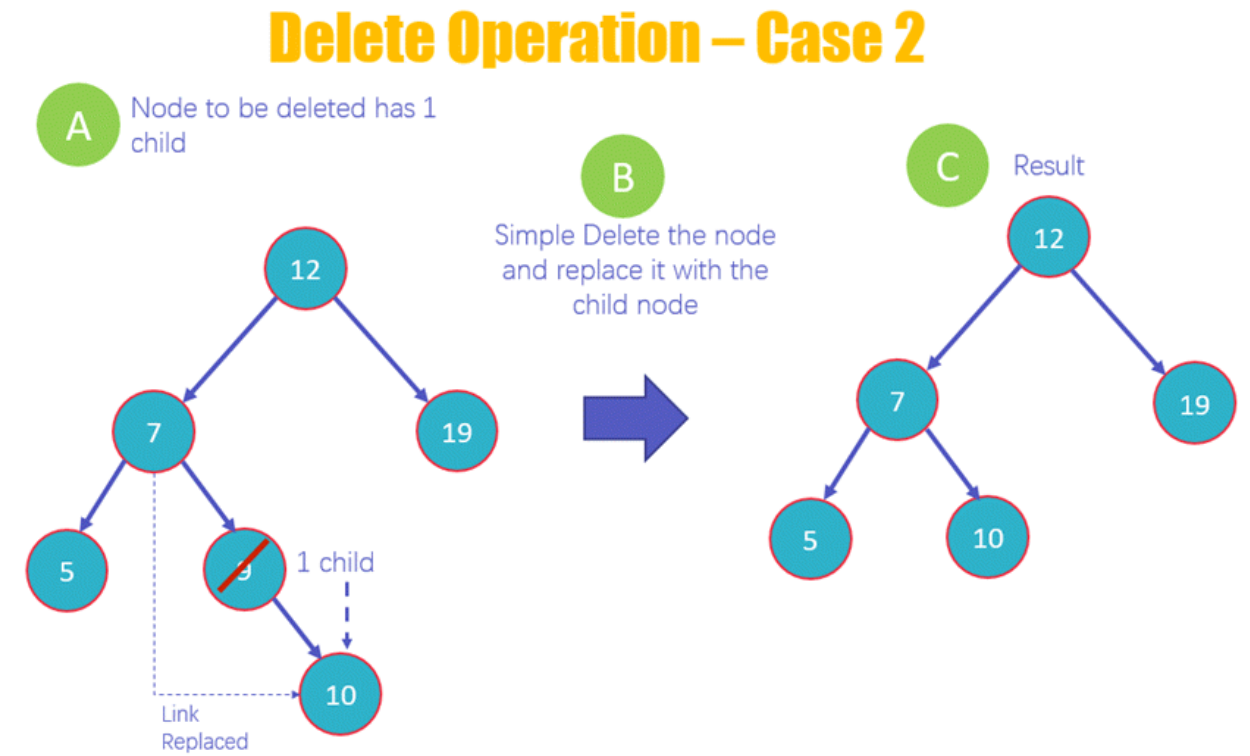
Insert:

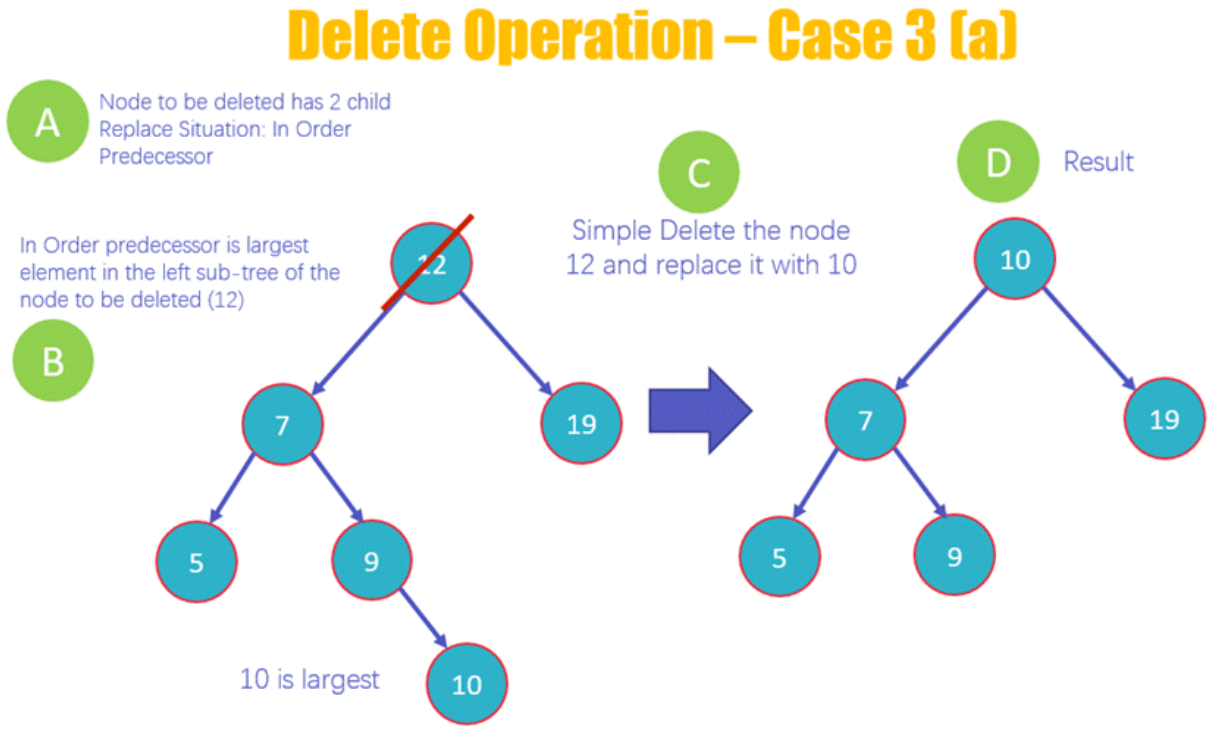


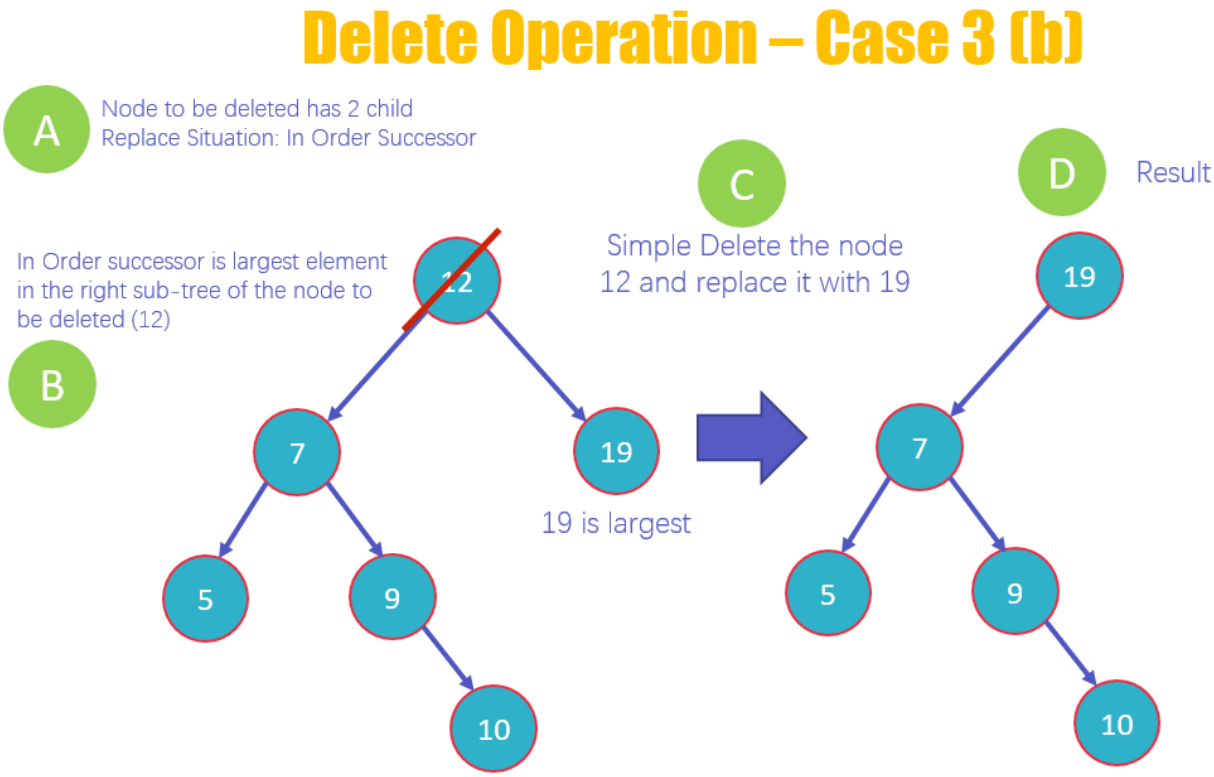
Deletion:

- **Case 1**- Node with zero children: this is the easiest situation; you just need to delete the node which has no further children on the right or left.
- **Case 2** – Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value.
- **Case 3** Node with two children: this is the most difficult situation, and it works on the following two rules
  - **3a – In Order Predecessor**: you need to delete the node with two children and replace it with the largest value on the left-subtree of the deleted node
  - **3b – In Order Successor**: you need to delete the node with two children and replace it with the smallest value on the right-subtree of the deleted node


Delete Operation – Case 1
A
Node to be deleted has 0 children
B
Simple Delete the node and remove the link
C
Result
12
7
19
5
9
10
No children
No children
No children
12
7
5
9
10

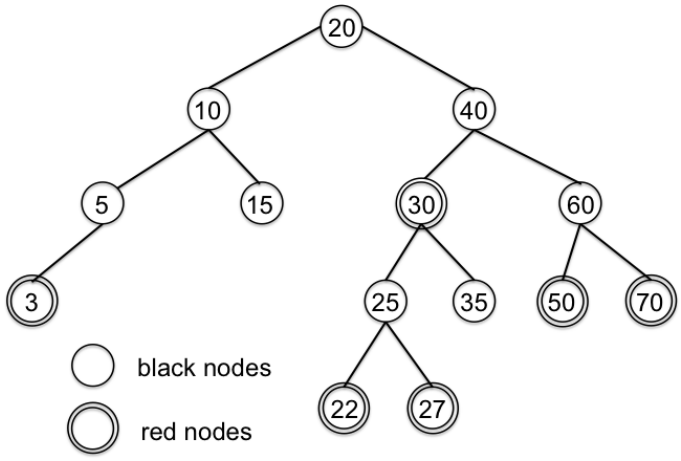

Delete Operation – Case 2
A
Node to be deleted has 1 child
B
Simple Delete the node and replace it with the child node
C
Result
12
7
19
5
9
10
1 child
Link Replaced
12
7
19
5
10


Delete Operation – Case 3 (a)
A
Node to be deleted has 2 child
Replace Situation: In Order
Predecessor
In Order predecessor is largest element in the left sub-tree of the node to be deleted (12)
B
C
Simple Delete the node
12 and replace it with 10
D
Result
12
7
19
5
9
10
10 is largest
10
7
19
5
9


Delete Operation – Case 3 (b)
A
Node to be deleted has 2 child
Replace Situation: In Order Successor
In Order successor is largest element in the right sub-tree of the node to be deleted (12)
B
C
Simple Delete the node
12 and replace it with 19
D
Result
12
7
19
5
9
10
19 is largest
19
7
5
9
10

3. Is the following tree a red-black tree? Explain your answer. If not, make appropriate changes to make this a valid red-black tree. [**2 points**]

   Show what the tree would look like after you insert the following values to the valid red-black tree obtained from above (make sure to explain all the steps involved and mark the red and black nodes accordingly): **[10 points]**
   a. 17
   b. 76
   c. 2



The pseudo-code for red-black tree insert is given below.

```
RB-INSERT(T, z)
  y = T.nil
  x = T.root
  while x ≠ T.nil
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == T.nil
      T.root = z
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED
  RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
  while z.p.color == RED
      if z.p == z.p.p.left
          y = z.p.p.right
          if y.color == RED
              z.p.color = BLACK
              y.color = BLACK
              z.p.p.color = RED
              z = z.p.p
          else if z == z.p.right
                  z = z.p
                  LEFT-ROTATE(T, z)
              z.p.color = BLACK
              z.p.p.color = RED
              RIGHT-ROTATE(T, z.p.p)
      else (same as then clause with "right" and "left" exchanged)
  T.root.color = BLACK
```

```
LEFT-ROTATE(T, x)
  y = x.right              // set y
  x.right = y.left         // turn y's left subtree into x's right subtree
  if y.left ≠ T.nil
      y.left.p = x
  y.p = x.p                // link x's parent to y
  if x.p == T.nil
      T.root = y
  elseif x == x.p.left
      x.p.left = y
  else x.p.right = y
  y.left = x               // put x on y's left
  x.p = y
```

4. For the given directed, edge-weighted input graph:

A	B	10
B	C	2
C	D	9
D	C	7
A	E	3
E	D	2
B	E	4
E	B	1
C	E	8

   a. Draw the graph that corresponds to the given input. **[7 points]**

   b. Draw the adjacency matrix representation for this graph. **[5 points]**

c. Draw the adjacency list representation for this graph. **[5 points]**

d. Which algorithm would you use to compute the shortest path between node A and all other nodes in the graph. Use that algorithm to compute the shortest path from node A to all other nodes (show the steps involved in computing the shortest paths) and write down the sequence of nodes traversed by the shortest paths. **[10 points]**

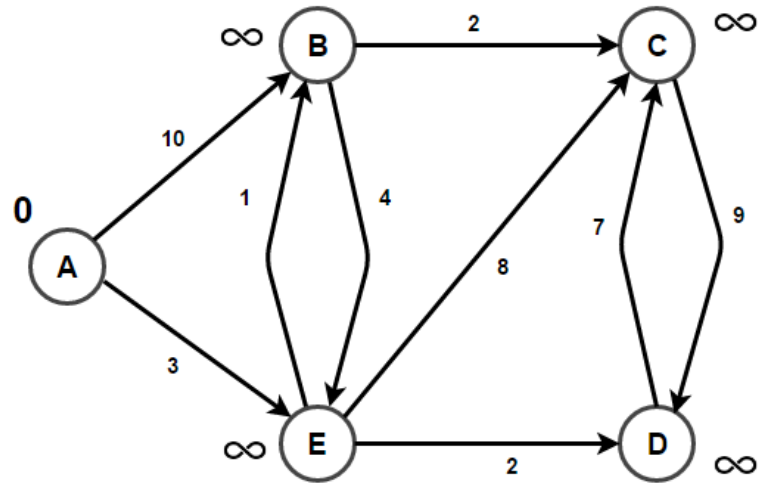
Dijkstra's:

Dijkstra's Algorithm is an algorithm for finding the shortest paths between nodes in a graph. For a given source node in the graph, the algorithm finds the shortest path between that node and every other node. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the fastest route to the destination node has been determined.

Dijkstra's Algorithm is based on the principle of relaxation, in which more accurate values gradually replace an approximation to the correct distance until the shortest distance is reached. The approximate distance to each vertex is always an overestimate of the true distance and is replaced by the minimum of its old value with the length of a newly found path. It uses a priority queue to greedily select the closest vertex that has not yet been processed and performs this relaxation process on all of its outgoing edges.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, **one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm**, we **find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source**.

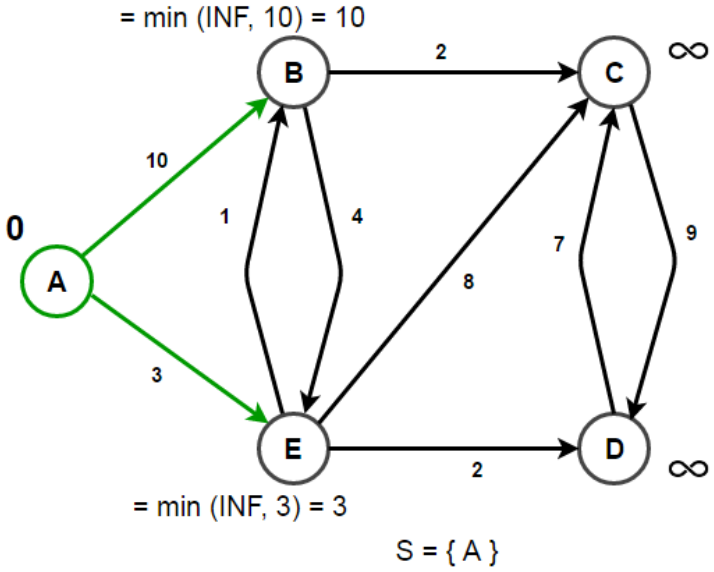
For instance, consider the graph. We will start with vertex `A`, So vertex `A` has a distance 0, and the remaining vertices have an undefined (infinite) distance from the source. Let `S` be the set of vertices whose shortest path distances from the source are already calculated.



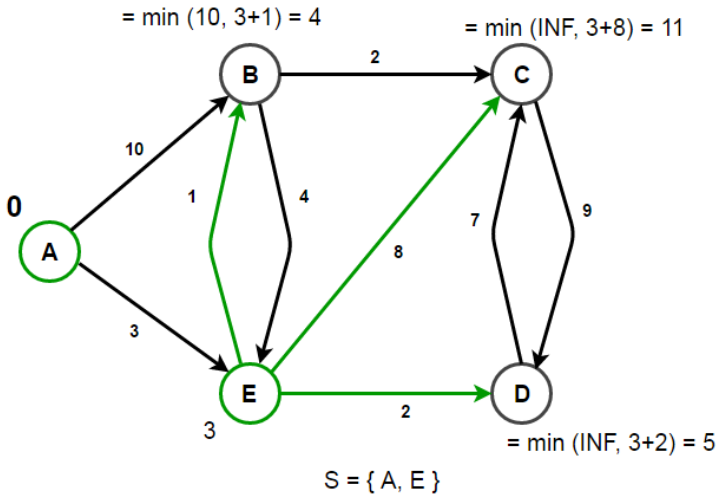
S = { }

Initially, `S` contains the source vertex. `S = {A}`.
We start from source vertex `A` and start relaxing `A's` neighbors. Since vertex `B` can be reached from a direct edge from vertex `A`, update its distance to `10` (weight of edge `A-B`).

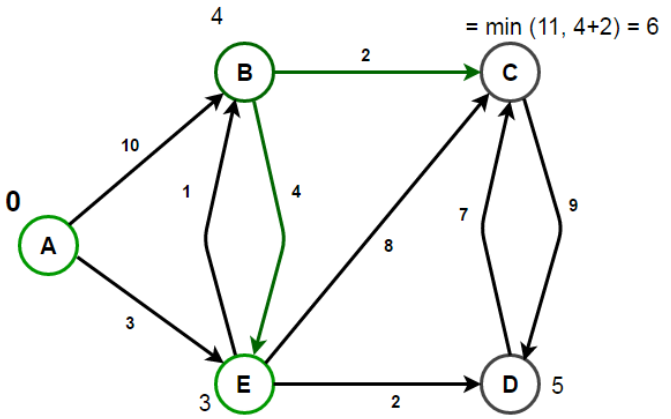
Similarly, we can reach vertex `E` through a direct edge from `A`, so we update its distance from `INFINITY` to `3`.



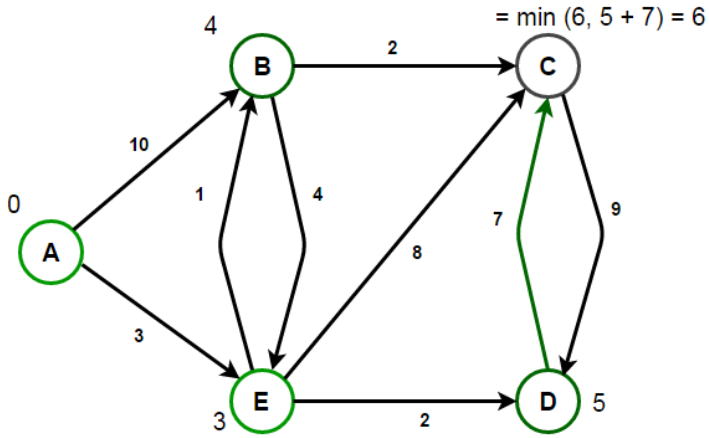
After processing all outgoing edges of `A`, we next consider a vertex having minimum distance. `B` has a distance of `10`, `E` has distance `3`, and all remaining vertices have distance `INFINITY`. So, we choose `E` and push it into set `S`. Now our set becomes `S = {A, E}`. Next, we relax with `E's` neighbors. `E` has 2 neighbors `B` and `C`. We have already found one route to vertex `B` through vertex `A` having cost `10`. But if we visit a vertex `B` through vertex `E`, we are getting an even cheaper route, i.e., (cost of edge `A-E` + cost of edge `E-B`) = `3 + 1 = 4 < 10` (cost of edge `A-B`).



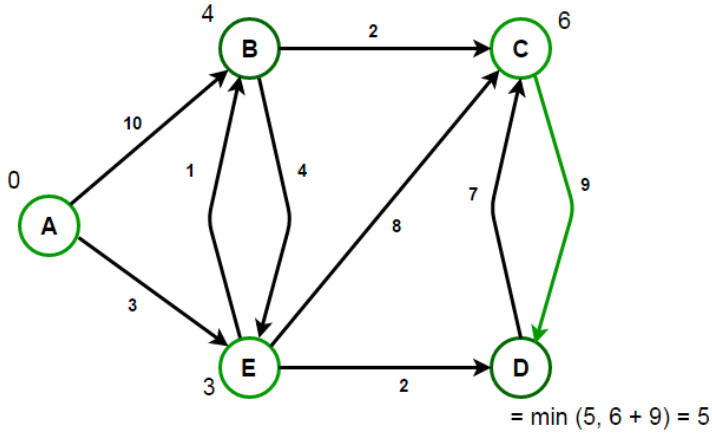
We repeat the process till we have processed all the vertices, i.e., Set `S` becomes full.



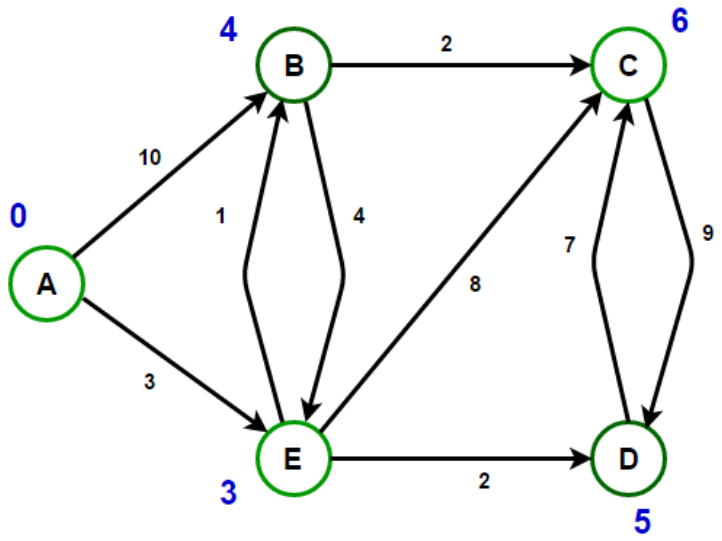
S = { A, E, B }



S = { A, E, B, D }

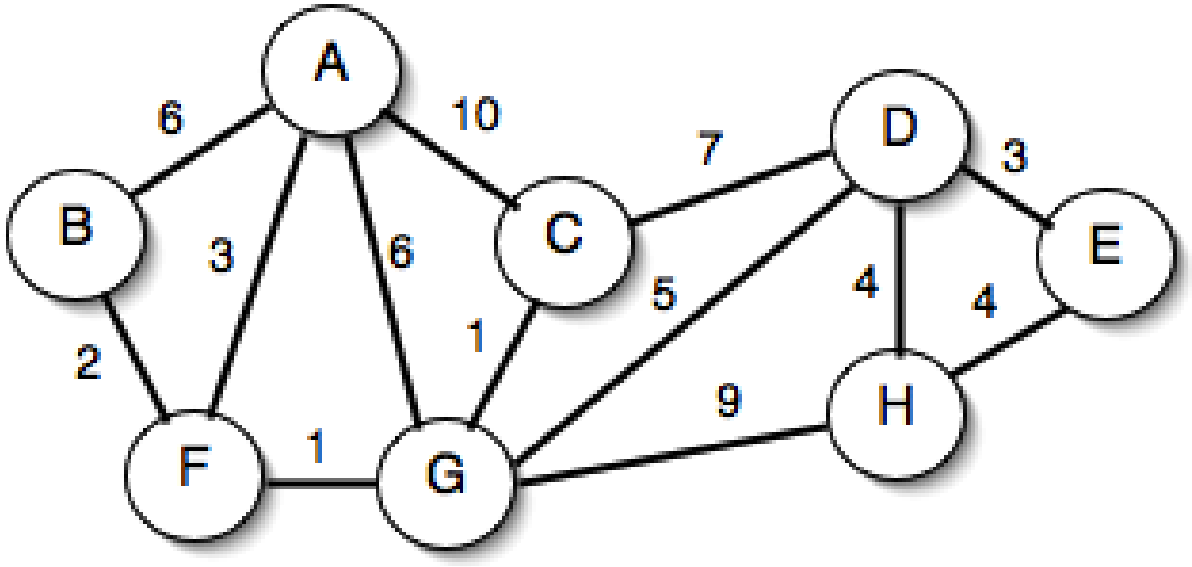


S = { A, E, B, D, C }



Vertex	Minimum Cost	Route
A –> B	4	A –> E –> B
A –> C	6	A –> E –> B –> C
A –> D	5	A –> E –> D
A –> E	3	A –> E

5. For the given edge-weighted graph compute the **minimum spanning tree**. Show the intermediate steps involved in computing the minimum spanning tree. Explain which algorithm you are using. **[10 points]**

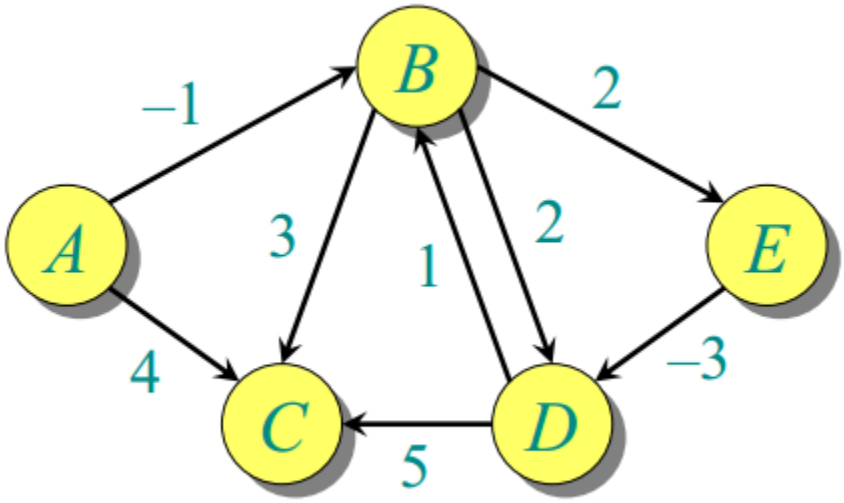


Prims or Kruskal?

Prims:

- ✓ Start with the minimum edge
- ✓ Always find the connected – minimum weighted edge
- ✓ Always check for the cycle

6. Which algorithm would you use to compute the shortest path between node A and all other nodes in the graph? Use that algorithm to compute the shortest path from node A to all other nodes (show the steps involved in computing the shortest paths). **[10 points]**

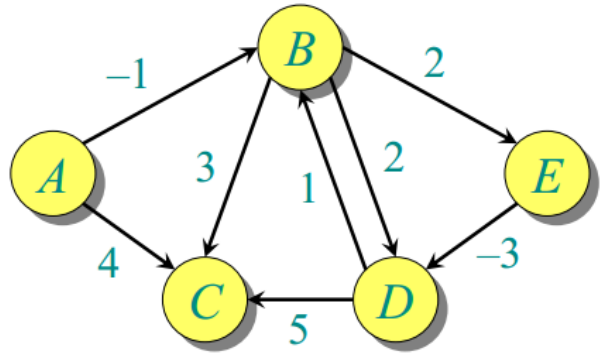


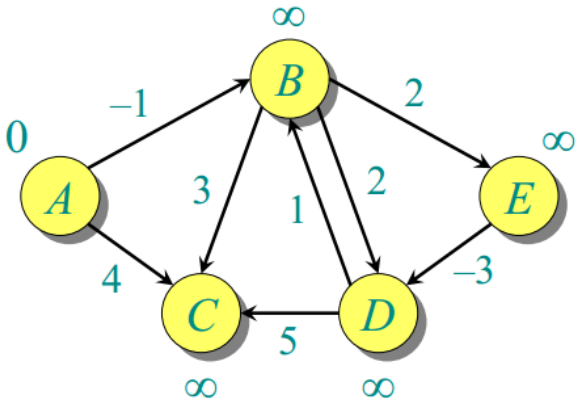
The idea is to use the Bellman–Ford algorithm to compute the shortest paths from a single source vertex to all the other vertices in a given weighted digraph. Bellman–Ford algorithm is slower than Dijkstra's Algorithm, but it can handle negative weights edges in the graph, unlike Dijkstra's.

If a graph contains a "negative cycle" (i.e., a cycle whose edges sum to a negative value) that is reachable from the source, then there is no shortest path. Any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. Bellman–Ford algorithm can easily detect any negative cycles in the graph.

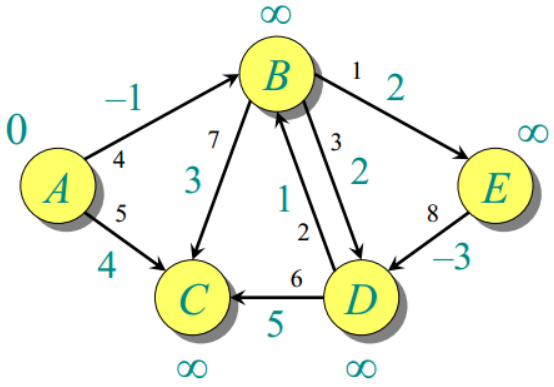
The algorithm initializes the distance to the source to `0` and all other nodes to `INFINITY`. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. At each iteration `i` that the edges are scanned, the algorithm finds all shortest paths of at most length `i` edges. Since the longest possible path without a cycle can be `V-1` edges, the edges must be scanned `V-1` times to ensure that the shortest path has been found for all nodes. A final scan of all the edges is performed, and if any distance is updated, then a path of length `|V|` edges have been found, which can only occur if at least one negative cycle exists in the graph.

The following slideshow illustrates the working of the Bellman–Ford algorithm.

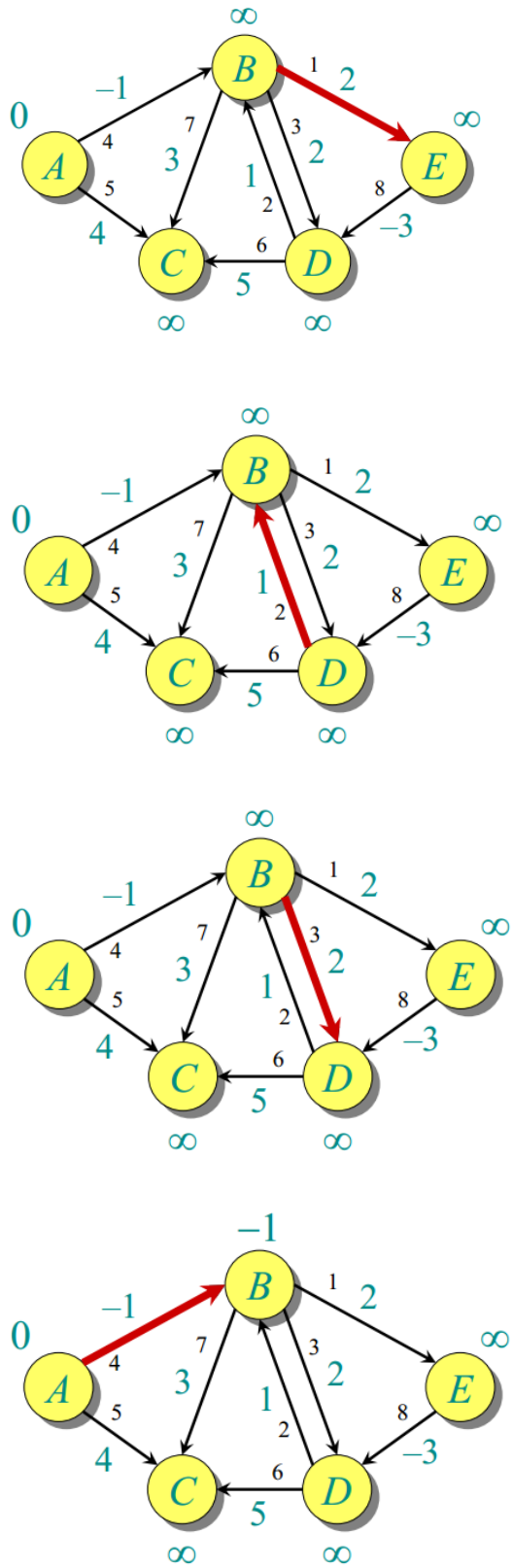


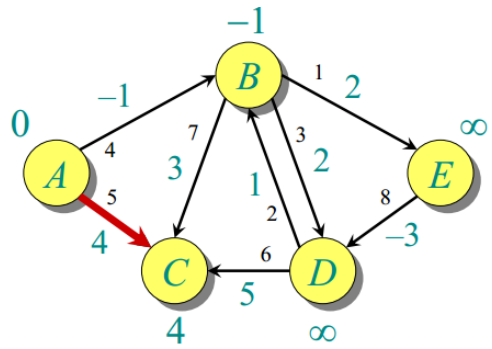


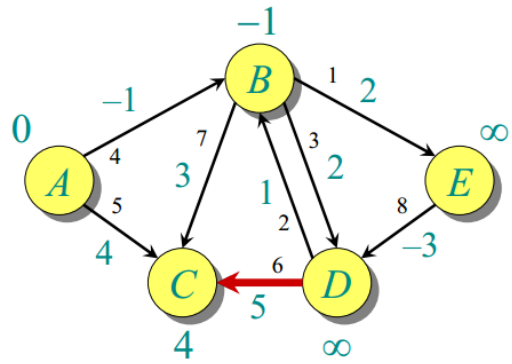
Initialization.

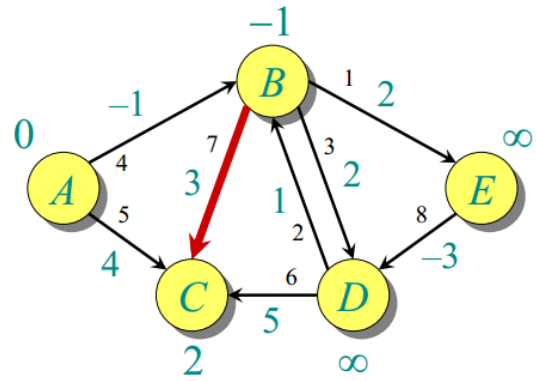


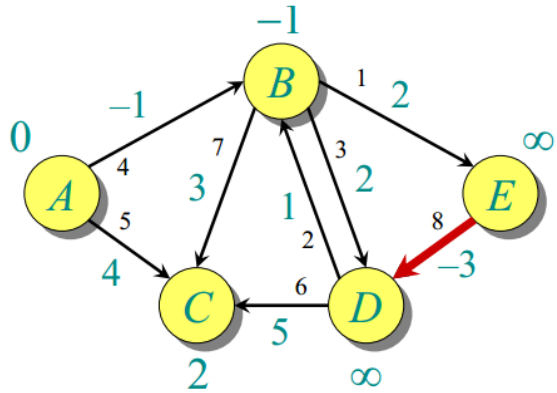
Order of edge relaxation.

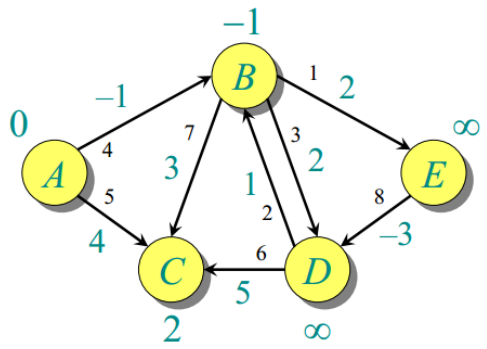

∞
B
1
2
−1
0
A
4
7
3
3
2
E
∞
5
1
2
8
4
−3
C
6
D
5
∞
∞
∞
B
1
2
−1
0
A
4
7
3
3
2
E
∞
5
1
2
8
4
−3
C
6
D
5
∞
∞
∞
B
1
2
−1
0
A
4
7
3
3
2
E
∞
5
1
2
8
4
−3
C
6
D
5
∞
∞
−1
B
1
2
−1
0
A
4
7
3
3
2
E
∞
5
1
2
8
4
−3
C
6
D
5
∞
∞


−1
B
1
2
−1
0
4
7
3
3
2
∞
A
E
5
1
2
8
4
6
−3
C
D
5
4
∞

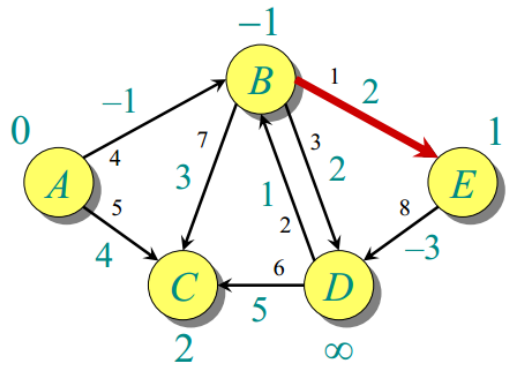

−1
B
1
2
−1
0
4
7
3
3
2
∞
A
E
5
1
2
8
4
6
−3
C
D
5
4
∞

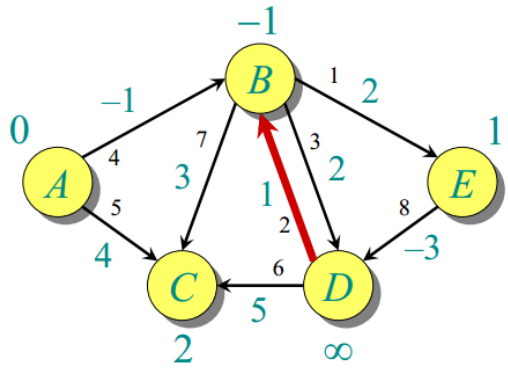

−1
B
1
2
−1
0
4
7
3
3
2
∞
A
E
5
1
2
8
4
6
−3
C
D
5
2
∞

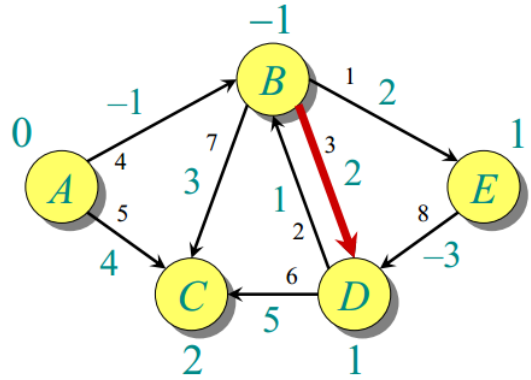

−1
B
1
2
−1
0
4
7
3
3
2
∞
A
E
5
1
2
8
4
6
−3
C
D
5
2
∞

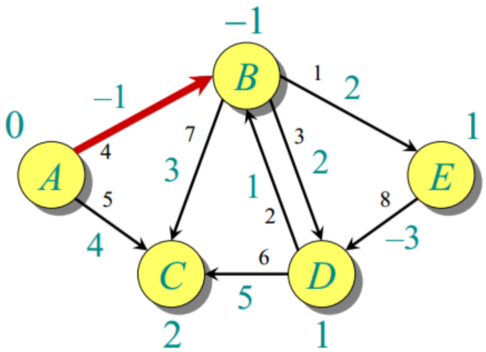

−1
B
1
2
0
−1
∞
A
4
7
3
3
2
E
5
1
2
8
4
−3
C
6
D
5
2
∞

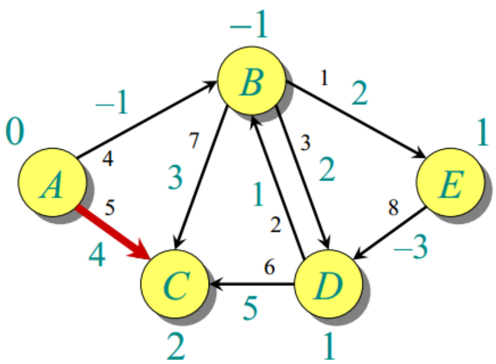
End of pass 1.

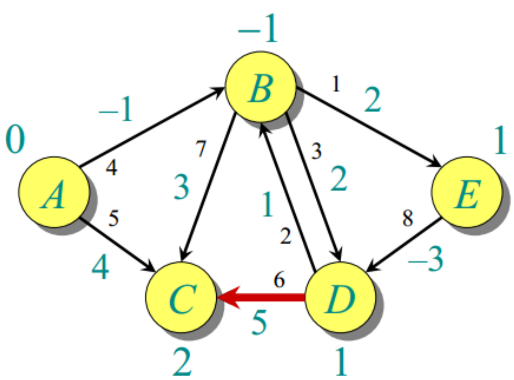

−1
B
1
2
0
−1
1
A
4
7
3
3
2
E
5
1
2
8
4
−3
C
6
D
5
2
∞

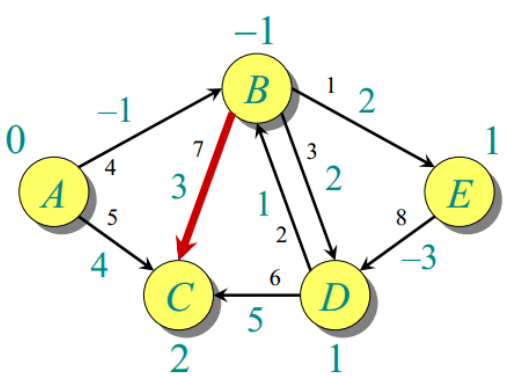

−1
B
1
2
0
−1
1
A
4
7
3
3
2
E
5
1
2
8
4
−3
C
6
D
5
2
∞

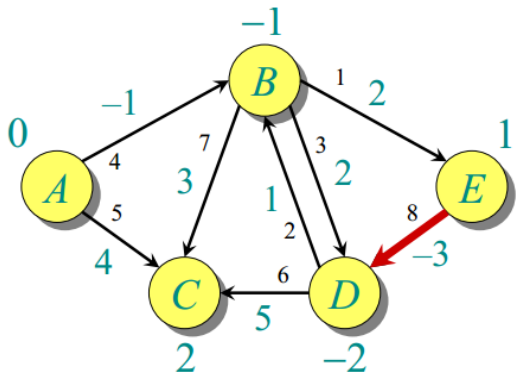

−1
B
1
2
0
−1
1
A
4
7
3
3
2
E
5
1
2
8
4
−3
C
6
D
5
2
1

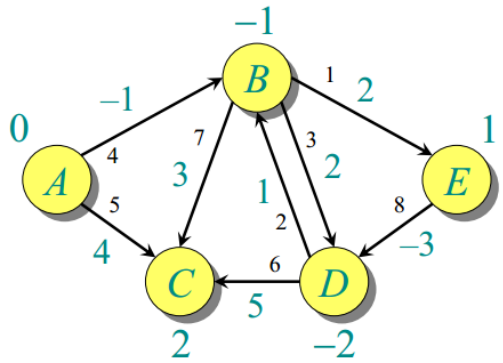

A
B
C
D
E
0
−1
1
2
1
−1
4
7
3
3
2
1
2
5
8
4
−3
6
5
2
1


A
B
C
D
E
0
−1
1
2
1
−1
4
7
3
3
2
1
2
5
8
4
−3
6
5
2
1


A
B
C
D
E
0
−1
1
2
1
−1
4
7
3
3
2
1
2
5
8
4
−3
6
5
2
1


A
B
C
D
E
0
−1
1
2
1
−1
4
7
3
3
2
1
2
5
8
4
−3
6
5
2
1





End of pass 2 (and 3 and 4).

```
Let source vertex = A,

The distance of

vertex B from vertex A is -1 and the path is [A –> B]
vertex C from vertex A is 2 and the path is [A –> B –> C]
vertex D from vertex A is -2 and the path is [A –> B –> E –> D]
vertex E from vertex A is 1 and the path is [A –> B –> E]
```

7. Explain the differences between Prim's and Kruskal's Algorithms **[8 points]**

Prim's Algorithm	Kruskal's Algorithm
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

8. Explain the differences between Dijkstra's and Bellman Ford's Algorithms **[8 points]**

The main advantage of Dijkstra's algorithm is its considerably low complexity, which is almost linear. However, when working with negative weights, Dijkstra's algorithm can't be used.

Also, when working with dense graphs, where E is close to V^2, if we need to calculate the shortest path between any pair of nodes, using Dijkstra's algorithm is not a good option

The Bellman-Ford algorithm can handle directed and undirected graphs with non-negative weights. However, it can only handle directed graphs with negative weights

9. Explain the Dynamic Programming approach and give an example that you prefer to use Dynamic Programming Approach to solve **[8 points]**

- ✓ Typically, all the problems that require maximizing or minimize certain quantities or counting problems that say to count the arrangements under certain conditions or certain probability problems can be solved by using Dynamic Programming.

- ✓ All dynamic programming problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property. Once, we observe these properties in a given problem, be sure that it can be solved using DP