Algorithms HW3

Q1)

A1) Input : stack (stk) of size n
Input :- k (integer where $0 \leq k \leq n$)
Algorithm :- ReverseStack Elements (stk, k) :-

```
Std :: queue<object> queue ;    // initialize empty queue
for i from 1 to k do :
    element : stk.pop()          (Transferring top k elements
    queue.enqueue(element)        to queue from stack)

while( ! queue.empty()) do :     (Transfer back elements
    element = queue.dequeue()     to stack)
    stk.push(element)
```

A2) Input :- stack (stk) of size n
Input :- Integers i & j (where $0 \leq i \leq j \leq n$)
Algorithm :- Reverse Elements (stk, i, j)

```
std : queue<objects> queue    // initialize empty queue
if( i == j) or( j = i+1):
        return
for i from 1 to i :
    if( ! stk.empty()) :
        queue.enqueue(stk.pop())

for h in range( j - i) :
    stk.append( queue.pop())
while queue :
    stk.append(queue.popleft()).
```

A3 → Input:- Stack (stk) of size n
Input:- Indexes i, j (where $1 \le i \le j \le n$)
Algorithm:- ReverseSwap (stk, i, j):-

```
    queue<object> queue;
    i = i - 1        (convert 1 indexto 0)
    j = j - 1
    while stk:
        queue.append(stk.pop())
    temp_i = None
    temp_j = None
    for h in range(len(queue)):
        element = queue.popleft()
        if h == i:
            temp_i = element
        else if h == j:
            temp_j = element
        queue.append(element)
    for k in range(len(queue)):
        element = queue.popleft()
        if k == i:
            stk.append(temp_j)
        elsif k == j:
            stk.append(temp_i)
        else:
            stk.append(element)
    while stk:
        queue.append(stk.pop())
    while queue:
        stk.append(queue.popleft())
```

Ans → Input : node (root of current subtree)
Algorithm PostOrderTraversal(node) :

```
if node == null :
    return
child = node.firstChild
while( !child == null) do :
    PostOrderTraversal(child)
    child = child.nextSibling

visit(node)
```

So, node becomes value → first_child → next_sibling.

Ans →

a) We can add a pointer to modify a BST so it identifies min in O(1). We know left side of tree is min value so, we can create a pointer to point the value. It will update every process like insertion, updation or deletion.

b) For insertion, we know BST has minimum value on left side node as left as possible in left-subtree. If we insert value in left side node as it will be minimum value, we can update that pointer which contain that value which changes for every insertion.

Modified Pseudocode :- If new value is less than or equal to current node's value, recursively insert into left subtree. After inserting, update node to point to newly inserted value if it's smaller than current min value.
Time Complexity will remain same..

c) If deleted node is minimum node, after deleting current node, new minimum will be leftmost node in BST.
To update minNode, traverse left subtree from root to find new min node.
Time complexity will be same.

Modified Pseudocode:- Delete node as in regular BST, and then update minimum node using traversing left from root.
Time complexity will be same as height remains same.