1. Create a BST and delete an element

InsertNode(root, key):
1. If root is NULL:
    Create a new node with key and return it
2. If key < root->key:
    root->left = InsertNode(root->left, key)
3. Else:
    root->right = InsertNode(root->right, key)
4. Return root

DeleteNode(root, key):
1. If root is NULL:
    Return root
2. If key < root->key:
    root->left = DeleteNode(root->left, key)
3. Else if key > root->key:
    root->right = DeleteNode(root->right, key)
4. Else:
    If root has one or no child:
        Return the non-NULL child or NULL
    If root has two children:
        Find inorder successor (smallest in right subtree)
        Replace root->key with successor's key
        Delete successor
5. Return root

2. Breadth-First Search (BFS)

BFS(graph, start):
1. Create an empty queue and enqueue start node
2. Mark start as visited
3. While queue is not empty:
    Dequeue a node, say current
    For each neighbor of current:
        If neighbor is not visited:
            Mark as visited and enqueue it

3. Depth-First Search (DFS)

DFS(graph, node, visited):
1. Mark node as visited
2. For each neighbor of node:
    If neighbor is not visited:
        Call DFS(graph, neighbor, visited)

4. Matrix Chain Multiplication

```
function MatrixChainMultiplication(p[]):
    n = length of p - 1
    let m be a 2D array of size n x n
    for i = 1 to n:
        m[i][i] = 0  // No cost to multiply one matrix

    for length = 2 to n:
        for i = 1 to n - length + 1:
            j = i + length - 1
            m[i][j] = infinity
            for k = i to j - 1:
                q = m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j]
                if q < m[i][j]:
                    m[i][j] = q
    return m[1][n]
```

5. AVL Tree Insertion and Deletion

InsertAVL(root, key):
1. Perform standard BST insertion
2. Update height of current node
3. Check balance factor:
    If unbalanced, perform appropriate rotations (LL, LR, RL, RR)
4. Return root

DeleteAVL(root, key):
1. Perform standard BST deletion
2. Update height of current node
3. Check balance factor:
    If unbalanced, perform appropriate rotations (LL, LR, RL, RR)
4. Return root

6. Create and Delete from a Red-Black Tree

InsertRBT(root, key):
1. Perform standard BST insertion
2. Fix violations using rotations and recoloring

DeleteRBT(root, key):
1. Perform standard BST deletion
2. Fix violations using rotations and recoloring

7. Heap Sort in Ascending and Descending Order

HeapSort(A):
1. Build a max-heap from the array
2. For i = n-1 to 1:
    Swap A[0] and A[i]
    Heapify(A, 0, i)

ReverseHeapSort(A):
1. Build a min-heap from the array
2. For i = n-1 to 1:
    Swap A[0] and A[i]
    Heapify(A, 0, i)

8. Prim's Algorithm

Prim(graph):
1. Initialize all nodes as unvisited
2. Start from any node and add it to the MST
3. While MST doesn't include all nodes:
    Select the smallest edge connecting visited and unvisited nodes
    Add the selected edge to the MST
    Mark the new node as visited

9. Red-Black Tree Creation and Deletion

// algorithm for deletion in Red-Black Tree

```
rb_delete(t, z)
   if z->left = NULL or z->right = NULL
        y ← z
   else y ← tree-successor(z)
   if y->left ≠ NULL
        x ← y->left
   else x ← y->right
   x->p ← y->p
   if y->p = NULL
        t->root ← x
   else if y = y->p->left
        y->p->left ← x
   else y->p->right ← x
   if y 3≠ z
        z->key ← y->key
   //copy y's satellite data into z
   if y->color = BLACK
        rb_delete_fixup(t, x)
   return y

  rb_delete_fixup(t, x)
   while x ≠ t->root and x->color = BLACK
      do if x = x->p->left
           w ← x->p->right
      if w->color = RED
           w->color ← BLACK //Case 1
      x->p->color ← RED //Case 1
      left-rotate(t, x->p) //Case 1
      w ← x->p->right //Case 1
      if w->left->color = BLACK and w->right->color = BLACK
           w->color ← RED //Case 2
      x ← x->p //Case 2
      else if w->right->color = BLACK
           w->left->color ← BLACK                //Case 3
      w->color ← RED          //Case 3
      right-rotate(t, w)        //Case 3
```

```
w ← x->p->right        //Case 3
w->color ← x->p->color //Case 4
x->p->color ← BLACK //Case 4
w->right->color ← BLACK //Case 4
left-rotate(t, x->p) //Case 4
x ← t->root /*Case 4*/
else (same as then clause with "right" and "left" exchanged)
   x->color ← BLACK
```

10. Strassen's Matrix Multiplication

```
Strassen(A, B):
1. If size of A or B is 1x1:
     Return A * B
2. Divide A and B into 4 submatrices each
3. Compute 7 matrix products using Strassen's formula:
     P1 = A11 * (B12 - B22)
     P2 = (A11 + A12) * B22
     ...
4. Combine results to form the resultant matrix
```

11. Create a Binary Tree from Preorder and Inorder Traversals

```
BuildTree(preorder, inorder):
1. If inorder is empty:
     Return NULL
2. Root = preorder[0]
3. Find root in inorder, say index
4. Recursively build left subtree from inorder[0:index] and preorder
5. Recursively build right subtree from inorder[index+1:end] and preorder
6. Return root
```

12. Merge Sort

```
MergeSort(A, left, right):
1. If left < right:
     mid = (left + right) // 2
     MergeSort(A, left, mid)
     MergeSort(A, mid+1, right)
     Merge(A, left, mid, right)

Merge(A, left, mid, right):
1. Create temporary arrays for left and right parts
2. Merge elements back into A in sorted order
```

13. BST Inorder Traversal

```
InorderTraversal(root):
1. If root is not NULL:
      Call InorderTraversal(root->left)
      Print root->key
      Call InorderTraversal(root->right)
```

14. Dijkstra's Algorithm

```
Dijkstra(graph, source):
1. Initialize distances of all nodes to ∞, distance[source] = 0
2. Create a priority queue and add source
3. While queue is not empty:
      Extract node with minimum distance, say u
      For each neighbor v of u:
         If distance[u] + weight(u, v) < distance[v]:
            Update distance[v]
            Add v to the queue
```