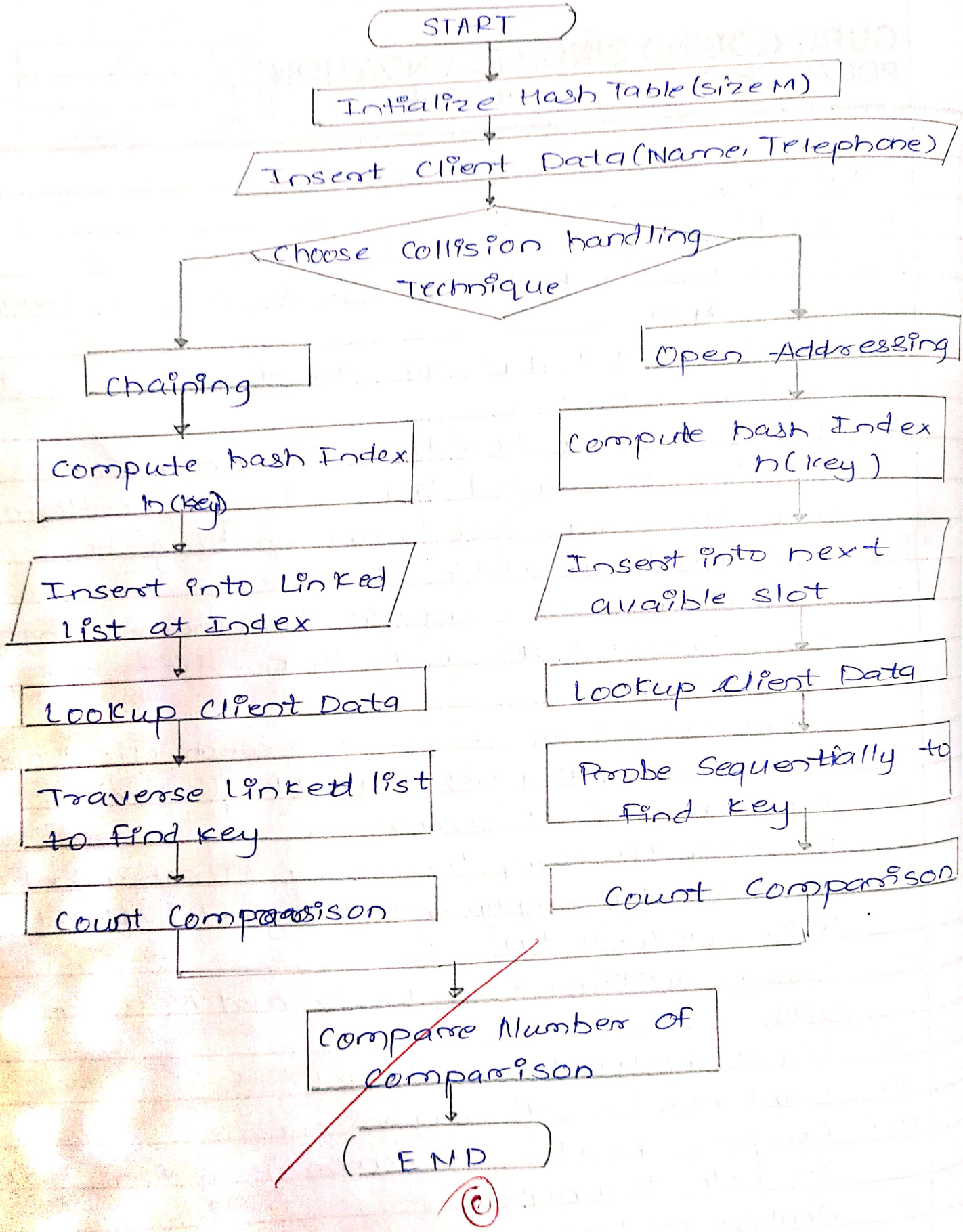
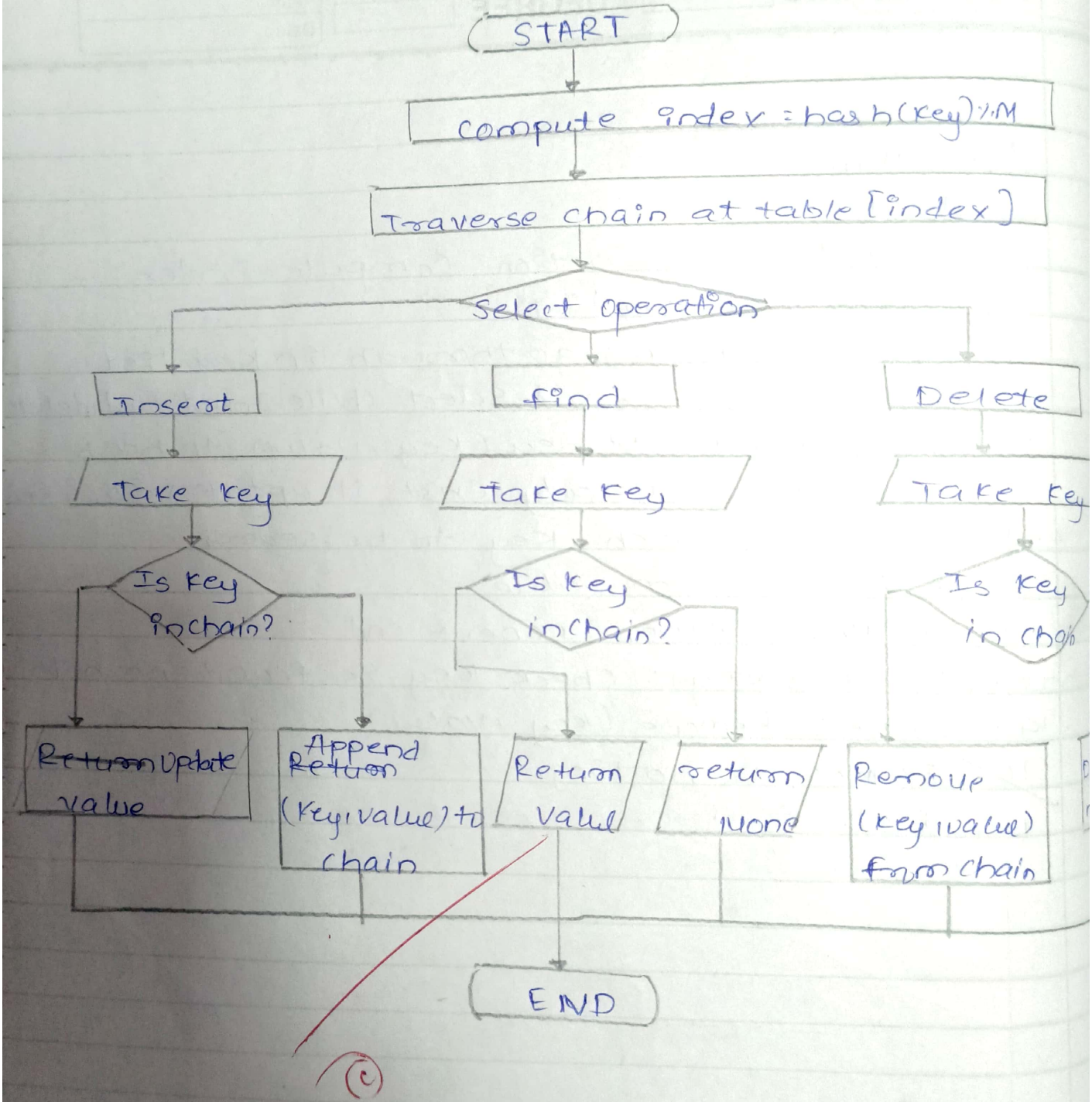
Algorithm:

1. Start
2. Initialize hash table : create a hash table of size M
3. Insert Client Data : Add client name, and telephone number.
4. Choose Collision Handling Technique:
   Chaining : Use linked list to handle collision
   Open Addressing : Use linear probing to find next available slot.
5. If Chaining then, compute Hash index using h(
6. Then, Insert Data into linked list at Compute index.
7. Lookup Client Data : Process to search client num
8. Transverse linked list to find key(index).
9. Count number of comparison.
10. Else : Open Addressing then, compute hash index h(x
11. If index slot empty, Insert key value. if not index = (index + 1) % M
12. Search (lookup) key at index match, return value.
13. If not probe next slot.
14. Count number of comparisons.
15. Compare number of comparison for lookup in both technique
16. Analyze performance.
17. Stop


START
Initialize Hash Table (size M)
Insert Client Data (Name, Telephone)
Choose Collision handling Technique
Chaining
Compute hash Index h(key)
Insert into Linked list at Index
Lookup Client Data
Traverse Linked list to find key
Count Comparison
Open Addressing
Compute hash Index h(key)
Insert into next avaible slot
Lookup Client Data
Probe Sequentially to find key
Count Comparison
Compare Number of comparison
END

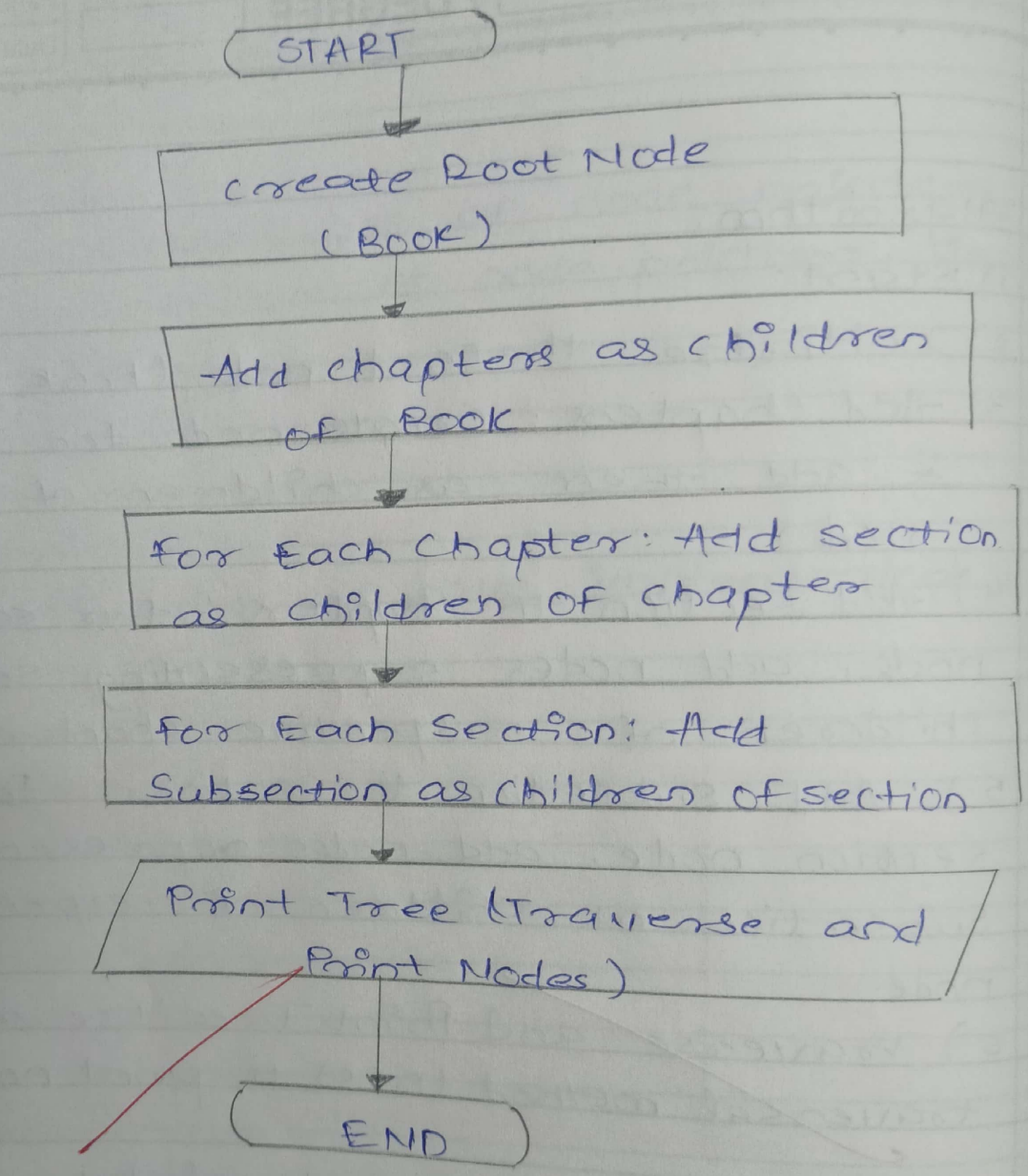
Algorithm:

1) Start
2. Hash ~~Treat~~ Function: Compute index:
   index = hash(key) % N
3. Traverse chain: go through linked list at computed index & select choice Insert, Find, delete
4. If Insert(key): If add new(key, value) to hash table. If key already exist it update value & end
5 If find(key): + search key to be search.
6. If Yes: Return value
7 If No: Return none. & end
8. If Delete (key): check key is found or not.
9. If Yes: Remove (key, value)
10 If No: Do nothing.
11. End.



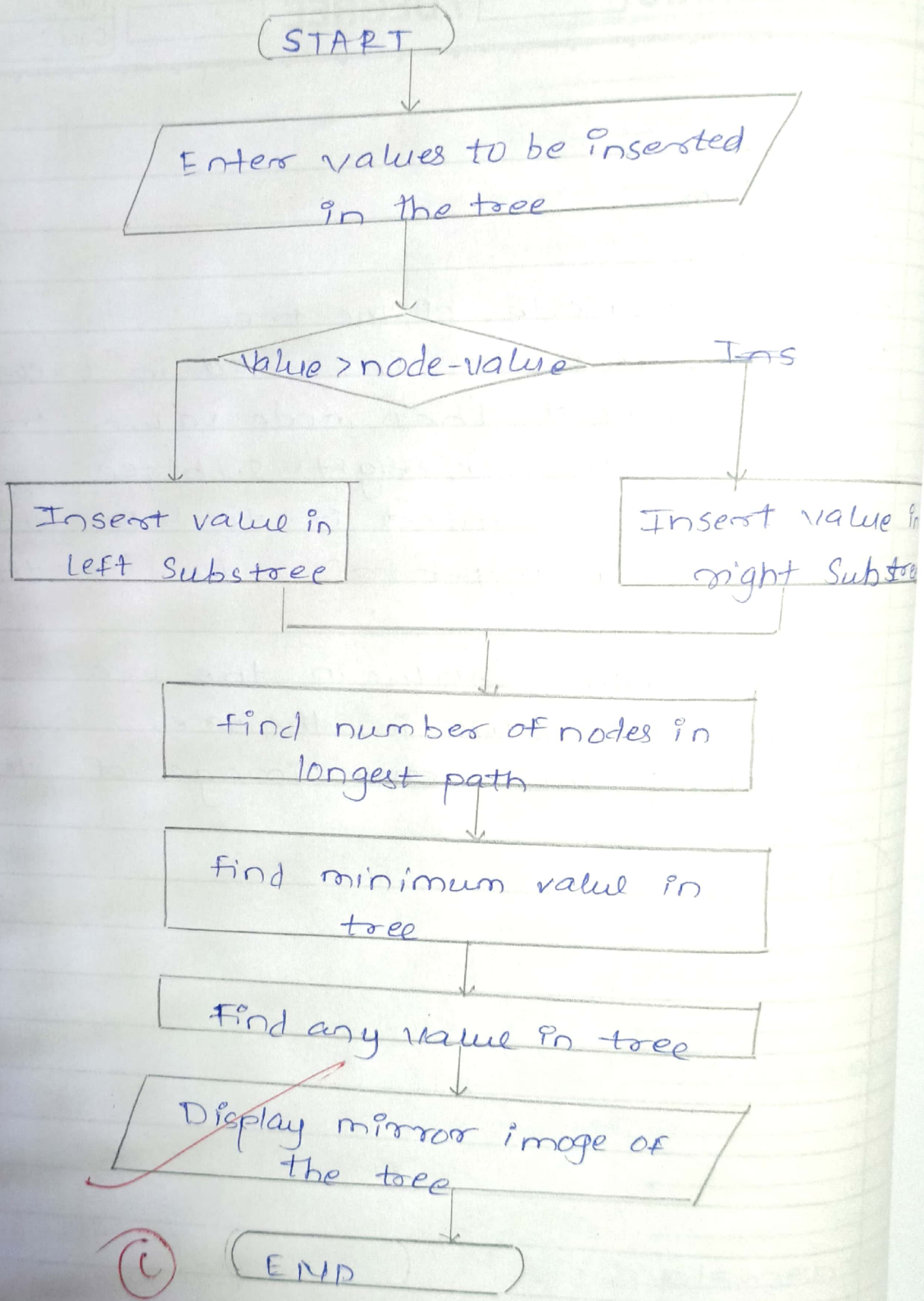
Algorithm:

1) Start.
2) Initialize the root node (book title)
3) Add chapters: create node for each chapter & add them as children of root (book node)
4) Add section to chapters: for each chapter node, add nodes representing section as children of respective chapter node.
5) Add subsection to sections: for each section node, add nodes representing subsections as children of respective section node.
6) Traverse and Print Tree: Use recursive traversal method (DFS) to print node)
7) End: tree is built and printed.


START
create Root Node (BOOK)
Add chapters as children of BOOK
For Each Chapter: Add section as children of chapter
For Each Section: Add Subsection as children of section
Print Tree (Traverse and Print Nodes)
END

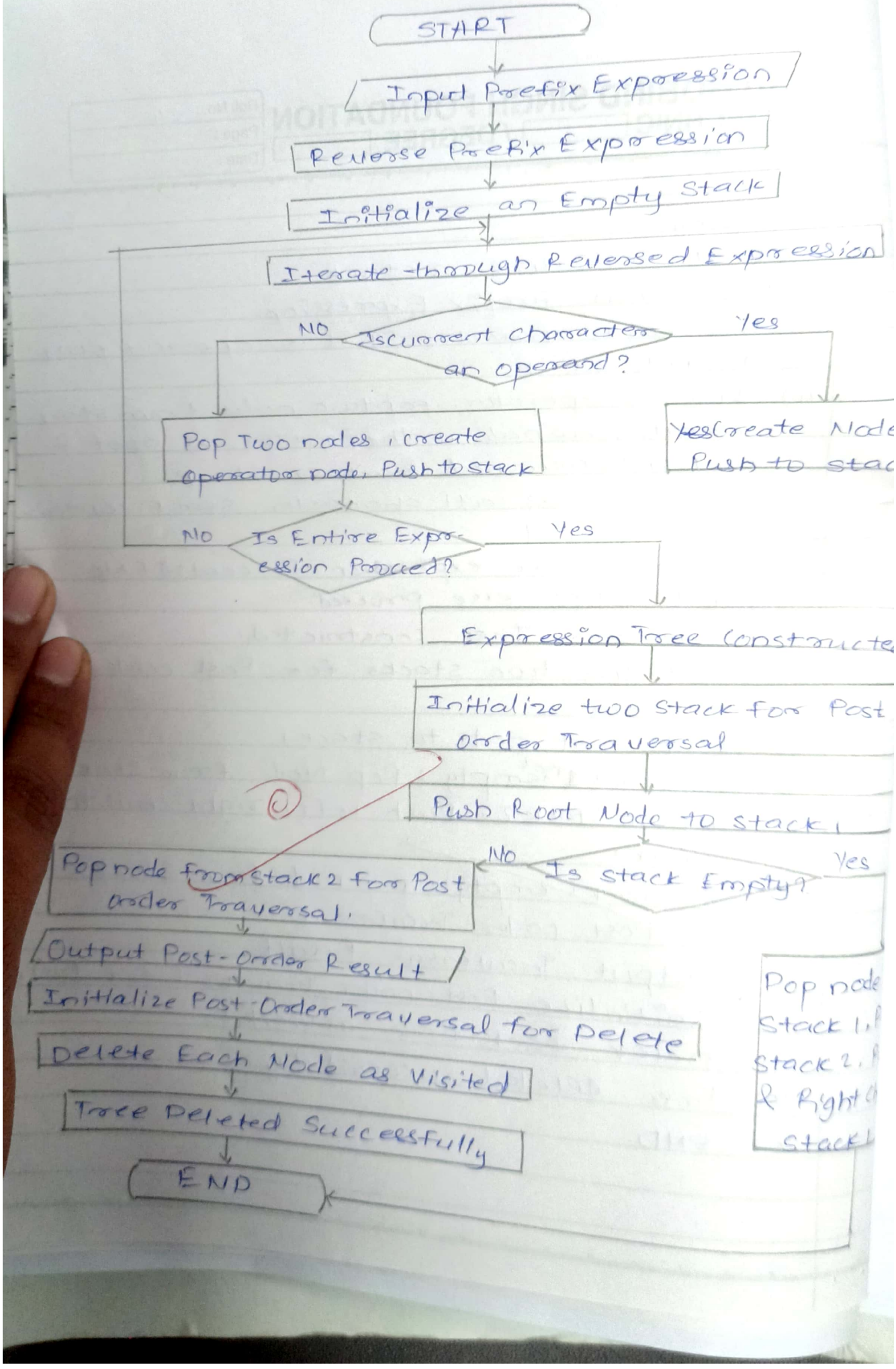
Algorithm

1) Start.
2) create node of the tree
3) Enter values to inserted in tree.
4) If value goreat than node value than insert value to right subtree
5) If not the insert it to left subtree
6) Find the number of nodes in the longest path.
7) Find minimum value in the tree.
8) Find any value in the tree.
9) Display the mirror image of the tree.
10) stop


START
Enter values to be inserted in the tree
value > node-value
Insert value in left Substree
Insert value in right Subtree
find number of nodes in longest path
find minimum value in tree
Find any value in tree
Display mirror image of the tree
END

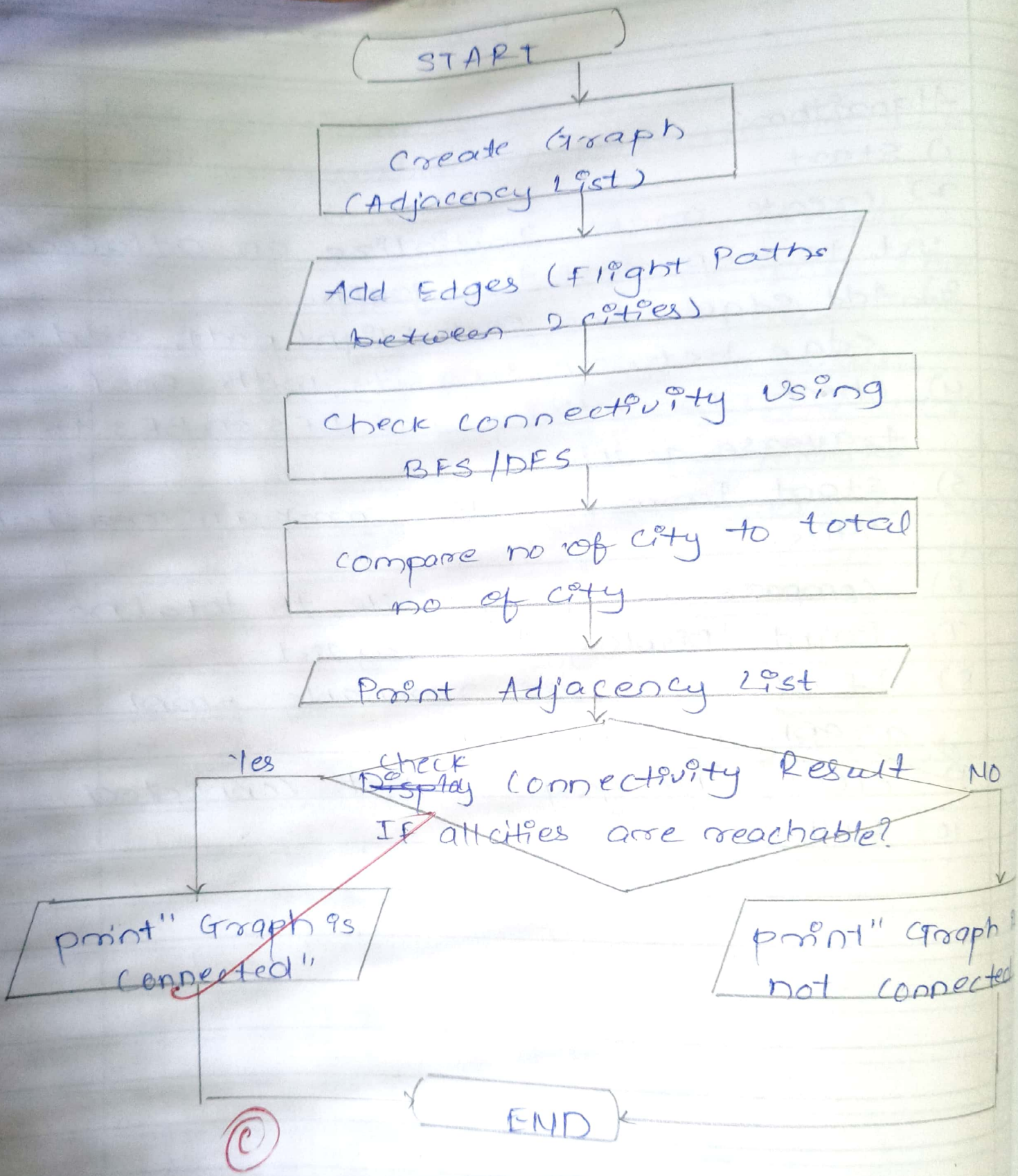
Algorithm

1) Start : Initialize a stack.
2) Take Input Prefix Expression.
3) For each character if it is operand push it into stack.
4) If it is operator, pop two nodes from stack. create new node with operator as root & push into new node back on stack.
5) After process all character stack contains only root node
6) check Entire expression Proceed ? IF NO : go to step5 else proceed.
7) Expression Tree constructed.
8) Initialize two stacks for Post-order Traversal.
9) Push root node to stack1
10) IF stack1 not empty : Pop Node from Stack1 Push to Stack2, Push left, right child to stack1
11) If stack1 empty : pop node from stack2 for Post order Traversal
12) Output Traversal Result.
13) Initialize Post-order Traversal for Delete.
14) Delete each node as visited.
15) Tree deleted successfully.
16) END.



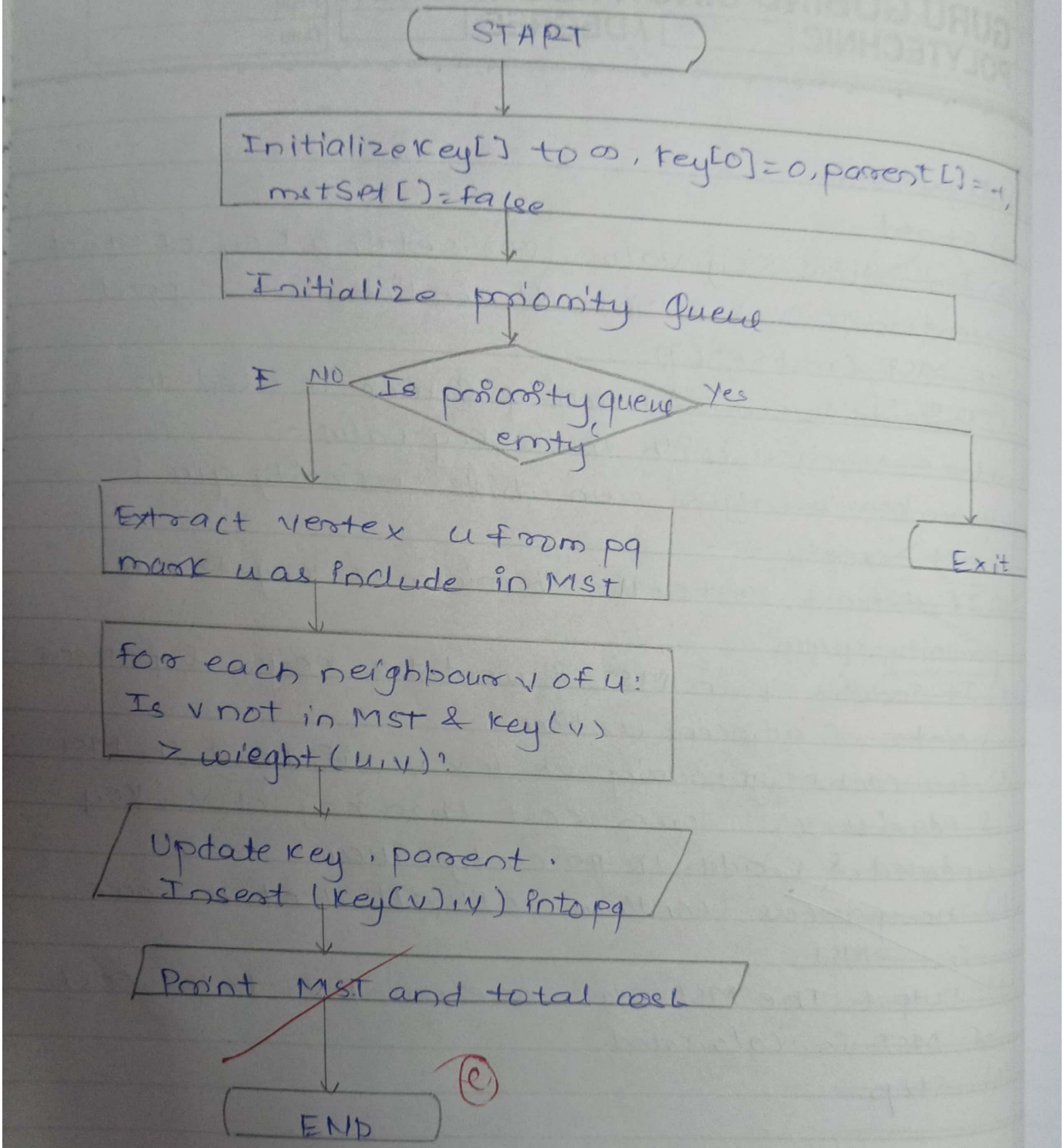
Algorithm:

1) Start
2) Create Graph: Initialize an adjacency list to represent the graph.
3) Add edges: For each flight path, add an edge between two city with cost.
4) Check connectivity: Use DFS or BFS to traverse graph
5) Start from any city & mark all reachable cities.
6) Compare no of city visite to total no.
7) Print / Display Adjacency list.
8) If all cities are reachable print graph is connected.
9) else: The graph is not connected
10) END.


START
Create Graph (Adjacency list)
Add Edges (Flight Paths between 2 cities)
Check connectivity using BFS/DFS
compare no of city to total no of city
Print Adjacency List
Check Connectivity Result
If all cities are reachable?
Yes
NO
print" Graph is connected"
print" Graph is not connected
END

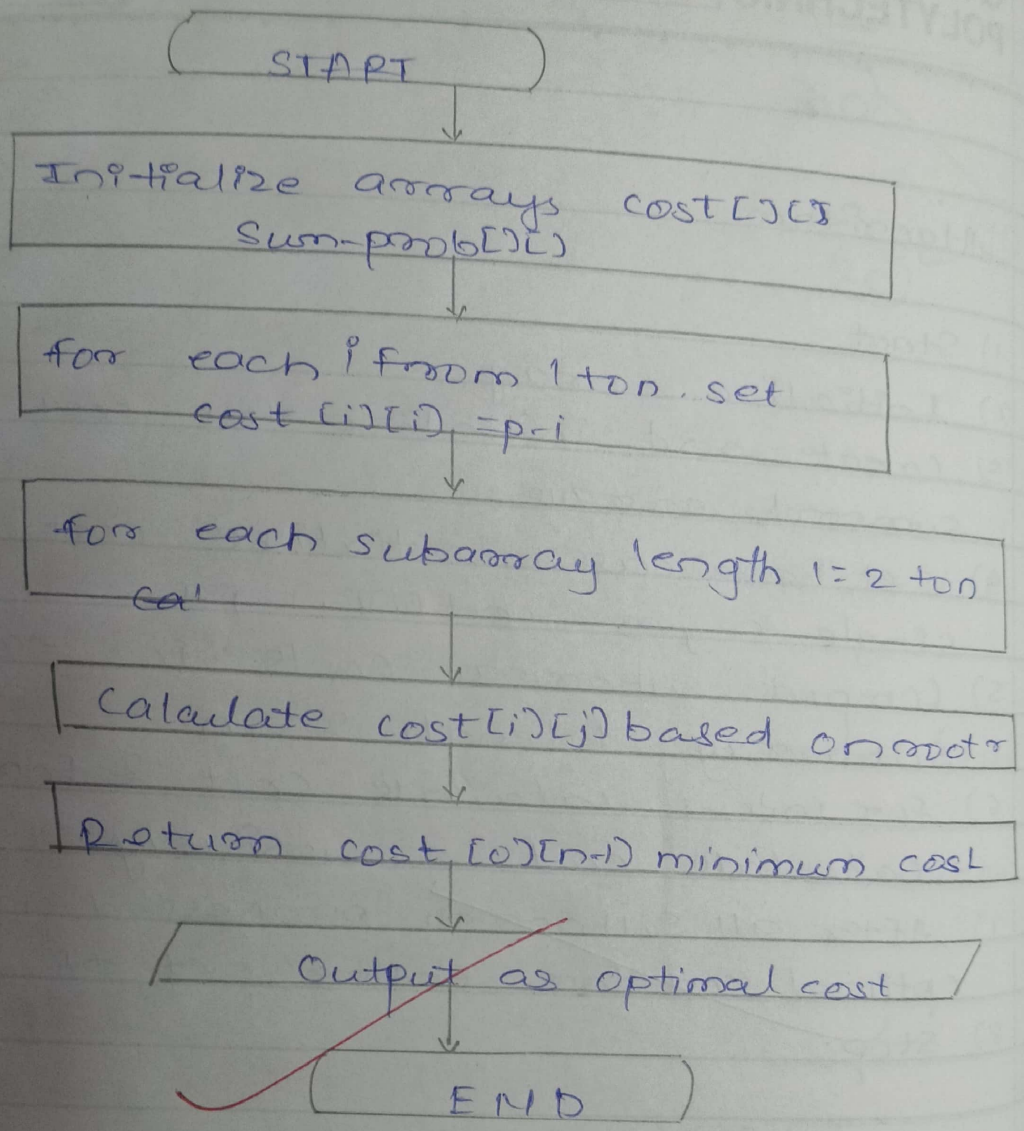
Algorithm:

1) Start
2) Initialize key value (∞), setting key of start vertex to 0, & setting all vertices as not part of MST (mstset[])
3) Priority Queue: A priority queue is used to store vertices with their key value.
4) Main loop: ~~The algo~~ run while priority que is not empty
5) It extract vertex u with minimum key from priority queue
6) It includes u in MST & updates keys & parent pointer of adjacent vertices
7) for each neighbouring vertex v, if v not in MST & edge(u, v) lower cost than key of v, key updated, & v adds to priority queue
8) The process continue untill all vertices are in MST
9) Output: The MST edges are print, & total cost of MST is calculated
10) Stop


START
Initialize key[] to ∞, key[0]=0, parent[]=-1, mstSet[]=false
Initialize priority queue
Is priority queue emty
NO
Yes
Extract vertex u from pq mark u as include in MST
Exit
for each neighbour v of u: Is v not in MST & key(v) > weight(u,v):
Update key, parent. Insert (key(v),v) into pq
Print MST and total cost
END

Algorithm:

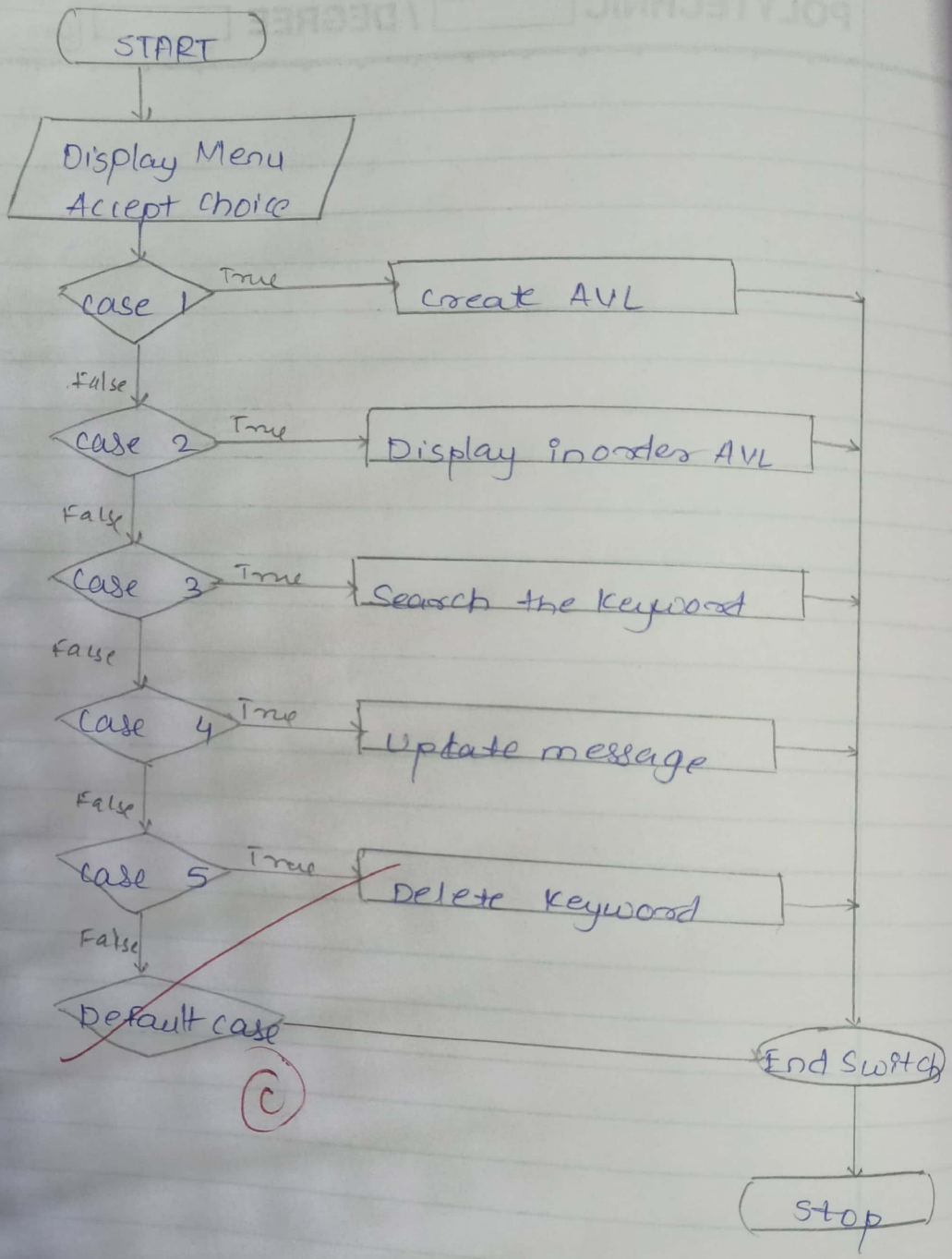
1) Start
2) Initialize Arrays.
3) Create and initialize the cost and sumprob. arrays
4) Set Base case: Initialize cost for a single key as cost[i][i] = p-i
5) Compute subarray costs: for each subarray of length 2 to n
6) For index i, calculate cost & for root k, calculate cost recursively.
7) After all subarray processed, give optimal cost for entire set of keys
8) Stop.


START
Initialize arrays cost[][] Sum-prob[][]
for each i from 1 to n, set cost[i][i] = p-i
for each subarray length l = 2 to n
Calculate cost[i][j] based on root
Return cost[0][n-1] minimum cost
Output as optimal cost
END

(c)

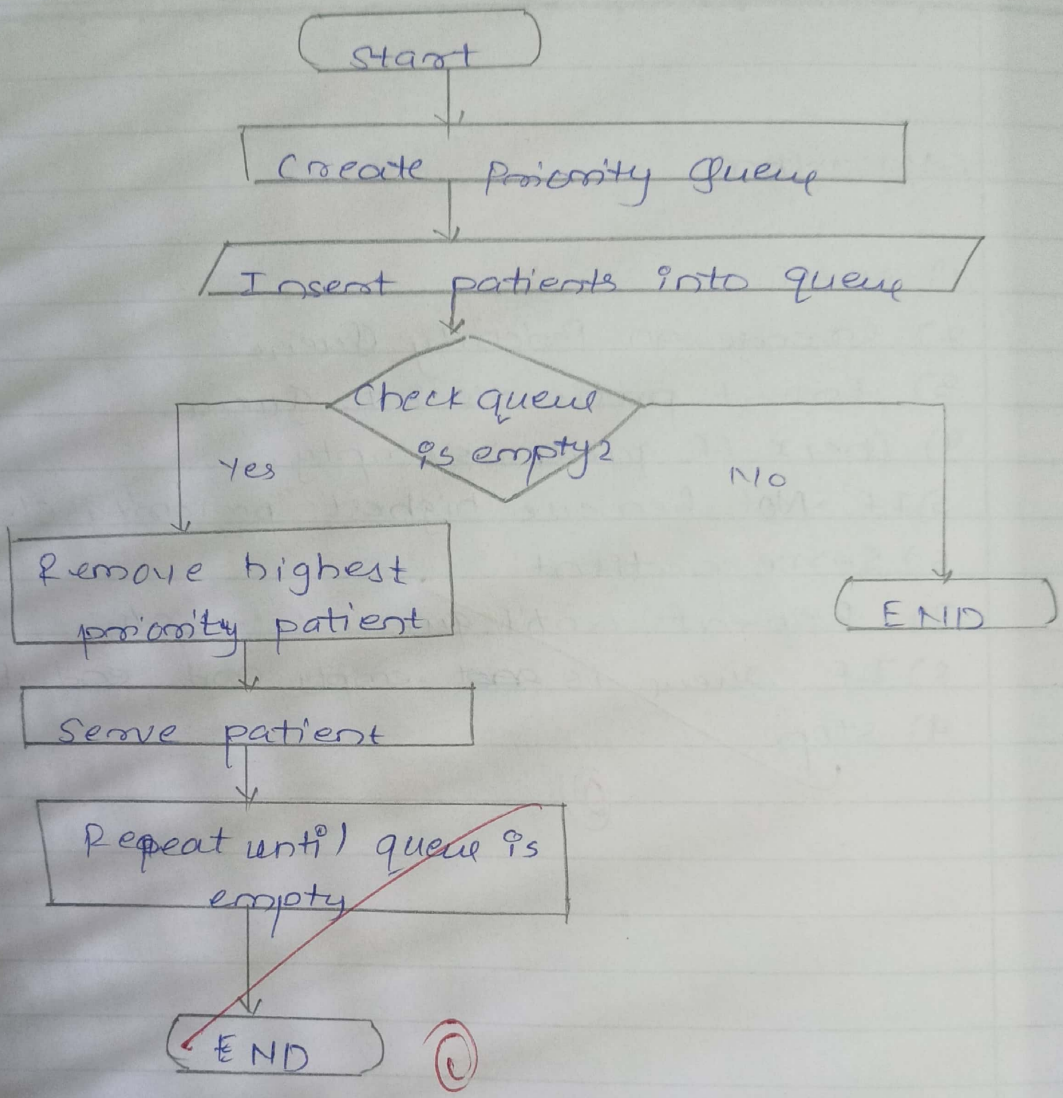
Algorithm:

1) start
2) Insert keyword: Search if keyword already exist, if not insert and rebalance the tree.
3) Delete keyword: Search for the keyword and remove it, then rebalance
4) Update keyword: Search for keyword, update the meaning.
5) Display sorted: Traverse tree in-order for ascending or reverse in-order for descending
6) Search keyword: Traverse tree to find the keyword.
7) find maximum comparisons: Measure maximum depth of tree for worst-case search
8) EXIT



Algorithm:

1) Start
2) Create an Priority Queue.
3) Insert patients into Queue.
4) Check if queue is empty.
5) If No: Remove highest priority element
6) Serve patient
7) Repeat until queue is empty
8) If queue is empty and end process
9) Stop

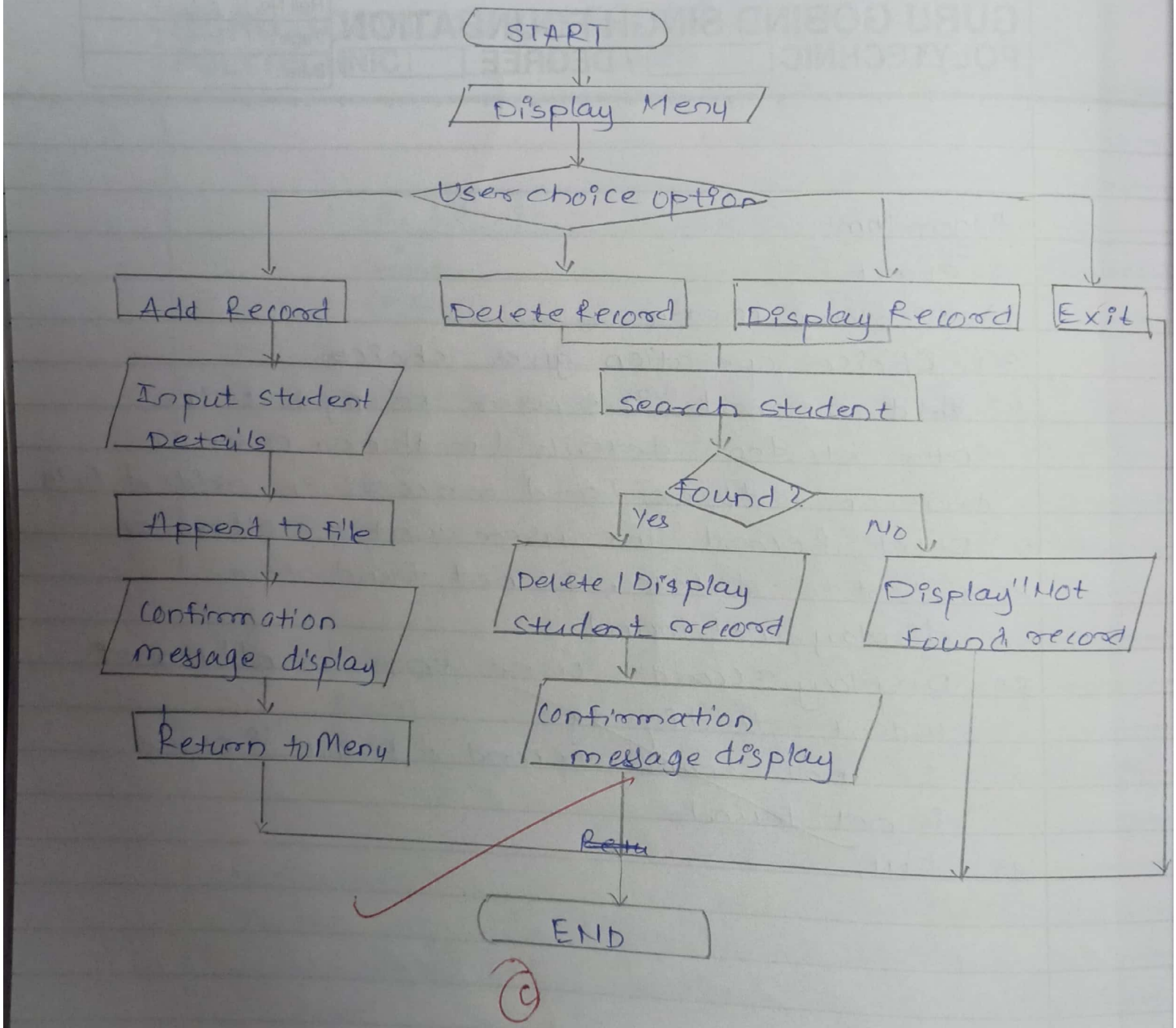

Start
Create priority Queue
Insert patients into queue
Check queue is empty?
Yes
No
Remove highest priority patient
END
Serve patient
Repeat until queue is empty
END

GURU GOBIND SINGH FOUNDATION
POLYTECHNIC [ ] / DEGREE [ ]

Roll No.: 64
Page :
Date :

Algorithm:

1) Start
2) Display menu
3) Choice operation gives choices.
4) Add records: The user prompted to enter student details (rool no, name, division, address) and record in added file
5) Delete Record The user enter roll no of student to delete, If not found then display not found.
6) Display record: user enter rool no of student to display.
8) If not found record, print "Record is not found"
9) END


START
Display Menu
User choice option
Add Record
Delete Record
Display Record
Exit
Input student Details
Append to file
Confirmation message display
Return to Menu
search student
Found ?
Yes
No
Delete / Display student record
Display "Not found" record
Confirmation message display
END

Algorithm:

1) Start
2) Input employee Details (ID, name, salary, designation)
3) Check employee ID exists in file or not
4) If Yes, then display "ID exists".
5) If No, Append employee to data file
6) Create and append index entry in index file
7) Sort the index file
9) close the file
10) END