Introduction:-

A singly linked list can be used to efficiently implement a stack data structure. In this implementation, the "top" of the stack corresponds to the head of the linked list. This approach offers advantages in terms of dynamic memory allocation, allowing the stack to grow or shrink as needed.

Key concepts:-

→ LIFO (Last-In, First-Out):
- Like a traditional stack, elements are added (pushed) and removed (popped) from the top of the stack.

→ Nodes:
- Each element in the stack is represented by a node in the linked list.
- Each node contains:
  - Data: The value of the element.
  - Next: A pointer to the next node in the stack.

→ Top pointer:
- We maintain a pointer called "top" that points to the first node in the list, which represents the top of the stack.

Operations:

→ Push (Insertion):
- A new node is created and added to the beginning of the linked list.
- The "top" pointer is updated to point to the new node.
- This operation is efficient, as it only involves updating pointers.

→ Pop (Deletion):
- The node at the top of the list is removed.
- The "top" pointer is updated to point to the next node in the list.
- This operation is also efficient, as it only involves updating pointers.

Diagram :

Top --> [Node 3 : Data] --> [Node 2 : Data] --> [Node 1 : Data] --> NULL

Explanation of the Diagram :

- "Top" indicates the top of the stack, which is the head of the linked list.
- "[Node : Data]" represents a node in the linked list, storing a data element.
- The arrows represent the "next" pointers, connecting the nodes.
- The last node's "next" pointer is NULL, indicating the bottom of the stack.
- In this example, Node 3 is the top of the stack meaning it was the last node added.

Algorithm for singly linked list using stack :-

1) Initialization :
   - Declare a structure 'Node' with fields 'info' (integer data) and 'next (pointer to next node).
   - Create a type alias 'node' for 'struct Node'.
   - Initialize a global pointer 'top' to 'NULL' (representing an empty stack).
2) Main function
   - Declare integer variables 'ch' (choice) and 'num' (input number).
   - Enter an infinite loop ('while (1)').
   - Display a menu with options:
     1) Push operation
     2) Pop Operation
     3) Display
     4) Exit
   - Read the user's choice ('ch').
   - Use a 'switch' statement based on the choice:
     - Case 1 (Push):

- Prompt the user to enter data to push.
- Read the input number ('num')
- Call the 'push(num)' function.
- Break.

• Case 2 (Pop):
  - Call the 'pop()' function.
  - Break.
• Case 3 (Display):
  - Call the 'display()' function.
  - Break.
• Case 4 (Exit):
  - Call 'exit(0)' to terminate the program.
  - Break.
• Default: (Implicitly handles invalid input, but could be added for better error handling.)

- End of 'switch' statement.
- End of 'while' loop.
- Wait for a key press using 'getch()'.
- Return 0 (successful execution).

3) Push function ('Push(int num)'):
- Allocate memory for a new node using 'malloc()'.
- If memory allocation fails, print an error message and exit.
- Set the 'info' field of the new node to 'num'.
- Set the 'next' field of the new node to the current 'top'.
- Update 'top' to point to the new node.
- Print a message indicating the pushed element.

4) Pop function ('Pop()'):
- Declare a pointer 'temp'
- If 'top' is 'NULL' (stack is empty), print an 'stack is empty" message.
- Otherwise:
  • Set 'temp' to 'top'.
  • Update 'top' to point to the next node ('top->next').
  • Print the 'info' field of the popped node ('temp->info').
  • Free the memory occupied by the popped node using 'free (temp)'.

5/ Display Function ('display()'):

- Declare a pointer 'temp'.
- If 'top' is 'NULL' (stack is empty), print an 'stack is empty" message.
- Otherwise.
  - Set 'temp' to 'top'.
  - Print "Contents are :".
  - Enter a loop that continues as long as 'temp' is not 'NULL'.
  - Print the 'info' field of the current node followed by " → ".
  - Move 'temp' to the next node ('temp → next').
  - End of loop.
  - Print a newline character.