SLL@ intro1

Introduction:-

A singly linked list can be effectively used to implement a queue data structure. This approach leverages the dynamic memory allocation capabilities of linked lists, which can be advantageous over array-based queues that have a fixed size.

Key Concepts:-

→ FIFO (First-In, First-Out):
Like a traditional queue, elements are added (enqueued) at the rear and removed (dequeued) from the front.

→ Nodes:
- Each element in the queue 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 queue.

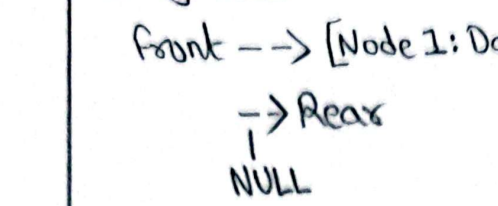
→ Front and Rear pointer:
- We maintain two pointers:
  - Front: Points to the first node in the queue (where dequeuing occurs).
  - Rear: Points to the last node in the queue (where enqueuing occurs).

Operations:

→ Enqueue (Insertion):
- A new node is created and added to the rear of the list.
- The rear pointer is updated to point to the new node.
- If the queue is empty, both front and rear pointers are set to the new node.

→ Dequeue (Deletion):
- The node at the front of the list is removed.
- The front pointer is updated to point to the next node.
- If the queue becomes empty, both front and rear pointers are set to NULL.

Diagram:



Explanation of the Diagram:

- "Front" indicates the beginning of the queue and "Rear" indicates the end.
- Each "[Node: Data]" represents a node in the linked list, storing a data element.
- The arrows represent the "next" pointers, connecting the nodes.
- The rear node next pointers points to NULL, indica-ting the end of the list.

Algorithm for Singly linked List Queue Implementation:-

1) Structure Definition:
   → Define a structure 'Node' with two fields:
      - 'info': An integer to store the data.
      - 'next': A pointer to the next 'Node'.
   → Initialize two global pointers:
      - 'front': Points to the front of the queue (initialized to 'NULL').
      - 'rear': Points to the rear of the queue (initialized to 'NULL').
2) 'main()' Function:
   → Enter an infinite loop ('while(1)').
   → Display a menu with the following options:
      -1) Enqueue operation
      -2) Dequeue operation
      -3) Display
      -4) Exit
   → Read the user's choice ('ch')
   → Use a 'switch' statement to execute the correspon-ding operation:
      • Case 1 (Enqueue):
         - Prompt the user to enter data('num').
         - Call the 'enqueue(num)' function.

- break;
- Case 2 (Dequeue):
  - Call the 'dequeue()' function.
  - break;
- Case 3 (Display):
  - Call the 'display()' function.
  - break;
- Case 4 (Exit):
  - Exit the program ('exit(0)').
  - break;
- Default:
  - (optional) Print an error message for invalid input.

→ End of 'switch' statement.
→ End of 'while' loop.
→ (optional) Wait for a key press ('getch()').
→ Return 0.

3) 'enqueue(num)' Function:
→ Allocate memory for a new 'Node' ('newnode').
→ Store the input data ('num') in 'newnode->info'.
→ If 'front' is 'NULL' (queue is empty):
- Set both 'front' and 'rear' to 'newnode'.
- Set 'front->next' and 'rear->next' to 'NULL'.

→ Else (queue is not empty):
- Set 'rear->next' to 'newnode'.
- Set 'rear' to 'newnode'.
- Set 'rear->next' to 'NULL'

→ Print a message indicating that 'num' was enqueued.

4) 'dequeue()' Function:
→ If 'front' is 'NULL' (queue is empty):
- Print "Queue is empty....".

→ Else (queue is not empty):
- Store the current 'front' node in a temporary pointer 'temp'.
- Update 'front' to point to the next node ('front = front->next').
- Print the dequeued element ('temp->info').
- Free the memory allocated for the dequeued node ('free(temp)').

5) 'display()' function:

→ If 'front' is 'NULL' (queue is empty):
- Print "Queue is empty".

→ Else (queue is not empty):
- Initialize a temporary pointer 'temp' to 'front'.
- Print "Contents are":

→ While 'temp' is not equal to 'rear':
- Print 'temp->info' followed by " <- ->".
- Move 'temp' to the next node ('temp = temp->next').

→ Print the 'temp->info' (which is the last element of the queue).