Introduction:

A singly linked list is a linear data structure in which elements are not stored in contiguous memory locations. Instead each element called a "node" contai - ns two parts.

→ Data: The actual value being stored.
→ Next: A pointer (or reference) to the next node in the sequence.

This structure allows for dynamic memory alloc ation, meaning the list can grow or shrink as needed during program execution.

Key Characteristics:-

→ Nodes:-
- Each node holds a piece of data and a pointer.
- The pointer is the last node is typically set to NULL (or None) to indicate the end of the list.

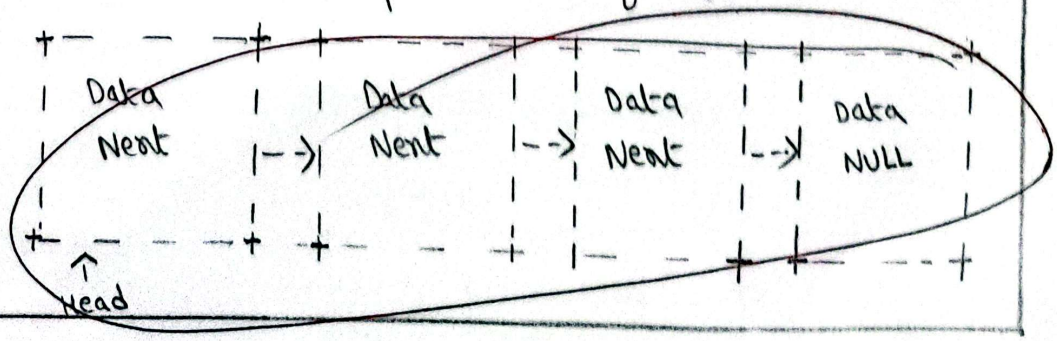
→ Head:-
- A special pointer called the "head" points to the first node in the list.
- Without the head, the entries list becomes inacces - ible.

→ Sequential Access:-
- Element can only be accessed sequentially, starti ng from the head. To each a specific node, you must traverse the list from the begining.

→ Dynamic Size:-
- Linked list can dynamically grow or shrink, makin them more memory-efficient than arrays in situations where the size of the data is unkno - wn or changes frequently.

Diagram:-

Here's a visual representation of singly linked list:



SLL intro 2

Explanation of the diagram

- Each rectangle represent a node.
- The "Data" portion of the node stores the actual information.
- The "Next" portion contains a pointer that points to the next node in the list.
- The "Head" pointer points to the first node.
- The last node's "Next" pointer is NULL, indicating the end of the list.

Algorith of singly linked list :-

1) Structure Definition
   → Define a structure Node with two fields:
      • info : An integer to store the data.
      • next : A pointer to the next 'Node'.
   → Define 'node' as a typedef for 'struct Node'.
   → Initialize a global pointer 'start' to 'NULL' (representing an empty list).

2) 'getnode()' function
   → Allocate memory for a new 'Node'.
   → If memory allocation fails, print an error message and exist.
   → Prompt the user to enter data.
   → Store the data in 'newnode->info'.
   → Set 'newnode->next' to 'NULL'.
   → Return the 'newnode' pointer.

3) 'Createlist(n)' function:
   → Iterate 'n' items
      • call 'getnode()' to create a new node.
      • If 'start' is 'NULL' (empty list), set 'start' to the 'newnode'
   → Otherwise:
      • Traverse the list to the last node.
      • Set the 'next' pointer of the last node to 'newnode'.

4) 'traverse()' function:
   → If 'start is NULL', print "List is empty".
   → Otherwise:
      • Initialize a temporary pointer 'temp' to 'start'.
      • While 'temp' is not 'NULL':
         - Print 'temp->info' followed by " --> ".

- move 'temp' to 'temp -> next'.
- Print a newline character.

5) 'Insert_first()' Function:
→ Call 'getnode()' to create a new node.
→ If 'start' is 'NULL', set 'start' to 'new node'.
→ Otherwise:
- Set 'newnode -> next' to 'start'.
- Set 'start' to 'newnode'.

6) 'Insert_after()' Function:
→ Call 'getnode()' to create a new node.
→ Prompt the user to enter the value after which to insert.
→ Initialize 'ptr' and 'preptr' to 'start'.
→ While 'ptr' is not 'NULL' and 'preptr -> info' is not the entered value:
- Set 'preptr' to 'ptr'.
- Set 'ptr' to 'ptr -> next'.
→ If the value is not found, print an error message, free the new node and return.
→ Set 'preptr -> next' to 'newnode'.
→ Set 'newnode -> next' to 'ptr'.
→ Call 'traverse()'.

7) 'Insert_before()' Function:
→ Call 'getnode()' to create a new node.
→ Prompt the user to enter the value before which to insert.
→ Initialize 'ptr' and 'preptr' to 'start'.
→ If 'start' is not 'NULL' and 'start -> info' is the entered value:
- Set 'newnode -> next' to 'start'.
- Set 'start' to 'newnode'.
- Call 'traverse()' and return.
→ While 'ptr' is not 'NULL' and 'ptr -> info' is not the entered value:
- Set 'preptr' to 'ptr'.
- Set 'ptr' to 'ptr -> next'.
→ If the value is not found, print an error, free the new node and return.
→ Set 'preptr -> next' to 'newnode'.
→ Set 'newnode -> next' to 'ptr'.

→ Call 'traverse()'

8) 'Insert_last()' Function:
→ Call 'getnode()' to create a new node.
→ If 'start' is 'NULL', set 'start' to 'newnode'.
→ Otherwise:
- Traverse to the last node.
- Set the 'next' pointer of the last node to 'newnode'.

9) 'delete_first()' Function:
→ If 'start' is 'NULL', print "List is empty".
→ Otherwise:
- Store 'start' in 'temp'.
- Set 'start' to 'start -> next'.
- Free the memory pointed to by 'temp'.
- Print "Node deleted".
- Call traverse();

10) 'delete_last()' Function:
→ If 'start' is 'NULL', print "List is empty".
→ If 'start -> next' is 'NULL', free start, set start to null, print "Node deleted" and call traverse.
→ otherwise:
- Initialize 'temp' and 'prev' to 'start'.
- Traverse to the last node, keeping track of the previous node in 'prev'.
- Set 'prev -> next' to 'NULL'.
- Free the memory pointed to by 'temp'.
- print "Node deleted".
- call traverse().

11) 'delete_after()' Function:
- Prompt the user to enter the value after which to delete.
- Initialize 'ptr' and 'preptr' to 'start'.
- Traverse the list until 'preptr -> info' matches the entered value.
- If the value or next node is not found, print an error and return.
- Set 'preptr -> next' to 'ptr -> next'.
- Free the memory pointed to by 'ptr'.
- Call traverse();

12) 'deletenode ( )' Function:
- → Prompt the user to enter the value to delete.
- → Initialize 'ptr' and 'preptr' to 'start'.
- → If 'start' is not 'NULL' and 'start -> info' is the entered value:
  - Set 'start' to 'start -> next'.
  - Free 'ptr'.
  - Call 'traverse ( )' and return.
- → Traverse the list until 'ptr -> info' matches the entered value.
- → If the value is not found, print an error and return.
- → Set 'preptr -> next' to 'ptr -> next'.
- → Free the memory pointed to by 'ptr'.
- → Call 'traverse( )'.

13) 'main ( )' Function:
- → Display a menu of options.
- → Get user input for the choice.
- → Use a 'switch' statement to call the appropriate function.
- → Repeat until the user chooses to exist.
- → Call 'getch( )' and return 0.