**Concept Of Structure Of A Node Of Singly Linked List.**

**The basic structure of a node contains two parts: information part (NAME, ROLL of student) and Next pointer (N). The node in the algorithm is represented using a structure student. In a program, the following statements represent a node:**

|  |  |
| --- | --- |
| **Information part of a singly linked list node** |  |

**STRUCTURE OF A NODE**

**struct STUDENT**

**{**

**char NAME[20];**

**int ROLL;**

**struct STUDENT \*N;**

**};**

**Start**

**NULL**

**Algorithm To Insert A Node In The End Of A Singly Linked List.**

**INSERT\_IN\_L\_LIST()**

**This algorithm inserts a node at the end of a linked list. ‘FRESH’ is a new node to be inserted into the end of the linked list. ‘START’ pointer points towards first node of the linked list (if present).**

1. **Allocate memory to a node FRESH.**
2. **Check if FRESH == NULL, then:**

**Print "NO MEMORY RECEIVED" and return.**

1. **Set NAME[FRESH] := DATA\_NAME.**

**Set ROLL[FRESH] := DATA\_ROLL.**

**Set N[FRESH] := NULL.**

1. **Check if START = NULL, then: [If linked list is absent]**

**Set START := FRESH.**

1. **Else [If linked list is present]**
2. **Set LOC := START.**
3. **Repeat while N[LOC] != NULL.**

**Set LOC := N[LOC].**

**[End of while loop of step (a)]**

1. **Set N[LOC] := FRESH.**

**[End of step 4 if structure]**

1. **Exit.**

**Algorithm To Traverse A Singly Linked List.**

**DISPLAY\_L\_LIST()**

**This algorithm traverses and displays nodes of a linked list. ‘START’ pointer points towards first node of the linked list (if present).**

1. **Check if START = NULL, then: [If linked list is absent]**

**Print "THE LIST IS EMPTY.” and return.**

1. **Else [If linked list is present]**
2. **Set LOC := START.**
3. **Repeat while LOC != NULL.**

**Print NAME[LOC] and ROLL[LOC].**

**Set LOC := N[LOC].**

**[End of while loop of step (a)]**

**[End of step 1 if structure]**

1. **Exit.**

**Start**

**NULL**

**Algorithm To Search a ITEM In A Singly Linked List.**

**SEARCH\_L\_LIST()**

**This algorithm traverses and displays nodes of a linked list. ‘START’ pointer points towards first node of the linked list (if present).**

1. **Check if START = NULL, then: [If linked list is absent]**

**Print "THE LIST IS EMPTY.” and return.**

1. **Else [If linked list is present]**
2. **Set LOC := START.**
3. **Repeat while LOC != NULL.**

**(c) If ROLL[LOC] = ITEM, then:**

**Set PTR := LOC and return**

**[End of step (c) if structure]**

**Set LOC := N[LOC].**

**[End of while loop of step (a)]**

**[End of step 1 if structure]**

1. **Exit.**

**Start**

**NULL**

**Algorithm To Insert New Node After A Given Node In Linked List.**

**INSERT\_AFTER()**

**‘AFTER\_ROLL’ is the value after which new node ‘A\_FRESH’ will be inserted. The node containing value of ‘AFTER\_ROLL’ will be pointed by pointer ‘ALOC’.**

1. **Allocate memory to a node A\_FRESH.**
2. **Check if A\_FRESH == NULL, then:**

**Print "NO MEMORY RECEIVED" and return.**

1. **Set NAME[A\_FRESH] := DATA\_NAME.**

**Set ROLL[A\_FRESH] := DATA\_ROLL.**

**Set N[A\_FRESH] := NULL.**

1. **Set ALOC := START.**
2. **Repeat while N[ALOC] != NULL.**
3. **If ROLL[ALOC] = AFTER\_ROLL, then:**

**Goto step 6.**

**[End of step (a) if structure]**

1. **Set ALOC := N[ALOC].**

**[End of while loop of step 5]**

1. **Set N[A\_FRESH] := N[ALOC].**
2. **Set N[ALOC] := A\_FRESH.**
3. **Exit.**

**Start**

**NULL**

**A\_FRESH**

**Algorithm To Insert New Node Before A Given Node In Linked List.**

**INSERT\_BEFORE()**

**‘BEFORE\_ROLL’ is the value before which new node ‘B\_FRESH’ will be inserted. The node containing value of ‘BEFORE\_ROLL’ will be pointed by pointer ‘BLOC’.**

1. **Allocate memory to a node B\_FRESH.**
2. **Check if B\_FRESH == NULL, then:**

**Print "NO MEMORY RECEIVED" and return.**

1. **Set NAME[B\_FRESH] := DATA\_NAME.**

**Set ROLL[B\_FRESH] := DATA\_ROLL.**

**Set N[B\_FRESH] := NULL.**

1. **Set PBLOC := BLOC.**
2. **Set BLOC := START.**
3. **Repeat while N[BLOC] != NULL.**
4. **If ROLL[BLOC] = BEFORE\_ROLL, then:**

**Goto step 7.**

**[End of step (a) if structure]**

1. **Set PBLOC := BLOC and BLOC := N[BLOC].**

**[End of while loop of step 6]**

1. **Set N[B\_FRESH] := BLOC.**
2. **Check if N[PBLOC] = NULL, then:**

**Set START := B\_FRESH.**

**Else**

**Set N[PBLOC] := B\_FRESH.**

**[End of step 8 if structure]**

1. **Exit.**

**Start**

**NULL**

**B\_FRESH**

**Algorithm On Deletion Of First Node From Linked List.**

**DELETE\_BEGIN()**

1. **Check if START = NULL, then: [If linked list is absent]**

**Print "THE LIST IS EMPTY.” and return.**

1. **Else [If linked list is present]**
2. **Set LOC := START.**
3. **Set START := N[LOC].**
4. **Free the memory occupied by LOC.**

**[End of step 1 if structure]**

1. **Exit**

**Start**

**NULL**

**Algorithm On Deletion Of Last Node From Linked List.**

**DELETE\_LAST()**

1. **Check if START = NULL, then: [If linked list is absent]**

**Print "THE LIST IS EMPTY.” and return.**

1. **Else [If linked list is present]**
2. **Set LOC := START.**
3. **Repeat while LOC != NULL.**

**Set PLOC := LOC and LOC := N[LOC].**

**[End of while loop of step (a)]**

1. **Set N[PLOC] := NULL.**
2. **Free memory occupied by LOC.**

**[End of step 1 if structure]**

1. **Exit.**

**Start**

**NULL**

**Algorithm On Concept Of Delete After A Given Node In Linked List.**

**DELETE\_AFTER()**

**‘AFTER\_ROLL’ is the value after which the node will be deleted. The node containing value of ‘AFTER\_ROLL’ will be pointed by pointer ‘LOC’.**

1. **Check if START = NULL, then: [If linked list is absent]**

**Print "THE LIST IS EMPTY.” and return.**

1. **Else [If linked list is present]**
2. **Set POS := START.**
3. **Repeat while POS != NULL.**

**If ROLL[POS] = AFTER\_ROLL, then:**

**Goto step 3.**

**[End of step (a) if structure]**

1. **Set POS := N[POS].**

**[End of while loop of step (a)]**

1. **Set LOC := N[POS].**
2. **N[POS] := N[LOC].**
3. **Free memory occupied by LOC.**

**[End of step 1 if structure]**

1. **Exit.**

**Start**

**NULL**

**Algorithm On Concept Of Delete Before A Given Node In A Linked List.**

**DELETE\_BEFORE()**

**‘BEFORE\_ROLL’ is the value before which node will be deleted. The node containing value of ‘BEFORE\_ROLL’ will be pointed by pointer ‘POS’.**

1. **Check if START = NULL, then: [If linked list is absent]**

**Print "THE LIST IS EMPTY.” and return.**

1. **Else [If linked list is present]**
2. **Set PLOC := START, LOC := START and POS := START.**
3. **Repeat while POS != NULL.**

**If ROLL[POS] = BEFORE\_ROLL, then:**

**Goto step 3.**

**[End of step (a) if structure]**

1. **Set PLOC := LOC.**
2. **Set LOC := POS.**
3. **Set POS := N[POS].**

**[End of while loop of step (a)]**

1. **Set N[PLOC] := POS.**
2. **Free memory occupied by LOC.**

**[End of step 1 if structure]**

1. **Exit.**

**Start**

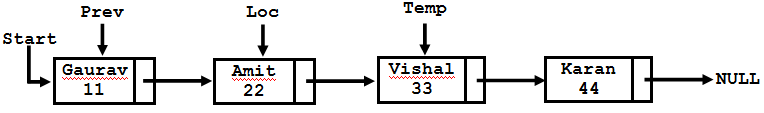
**NULL**

**Algorithm On Concept Of Reversal Of A Linked List.**

1. **Set PREV := NULL.**
2. **Repeat steps (a) to (d) while(LOC)**
3. **Set** **TEMP := N[LOC].**
4. **Set N[LOC] := PREV.**
5. **Set PREV := LOC.**
6. **Set** **LOC := TEMP**

**[End of while loop of step 2]**

1. **Set LOC := PREV.**
2. **Exit.**



**Given Only A Pointer To The Node To Be Deleted In A Singly Linked, Discuss Function On How To Remove That Node.**

**DeleteNode(NODE LOC)**

1. **Check if(LOC), then:**
2. **Set N[LOC] := N[N[LOC]].**
3. **Free memory pointed by LOC.**
4. **Exit.**

**Start**

**NULL**

**Discuss Logic To Detect A Loop In A Singly Linked List.**

**STEPS:**

1. **Start two pointers at the head of the list.**
2. **Loop infinitely.**
3. **If the fast pointer reaches a NULL pointer return that the list is NULL terminated.**
4. **If the fast pointer moves onto or over the slow pointer return that there is a cycle.**
5. **Advances the slow pointer one node.**
6. **Advances the fast pointer two node.**

**Algorithm:**

**Two pointers are used in this algorithm “fast” and “slow”. “fast” pointer will move twice faster than “slow” pointer.**

**int DetermineTermination(NODE \*head)**

1. **Set fast := slow := START;**
2. **Repeat steps (a) to (c) while(1). [Infinite Loop]**
3. **Check** **if(!fast or !N[fast]), then: return 0.**
4. **Else Check if (fast==slow or N[fast]==slow), then:**

**return 1. [Loop Detected]**

1. **Else set slow := N[slow] and fast := N[N[fast]].**

**[End of if structure of step (a)]**

**[End of while loop of step 2]**

1. **Exit.**