Nobody can go back and start a new beginning, but anybody can start today and make a new ending.
— Unknown

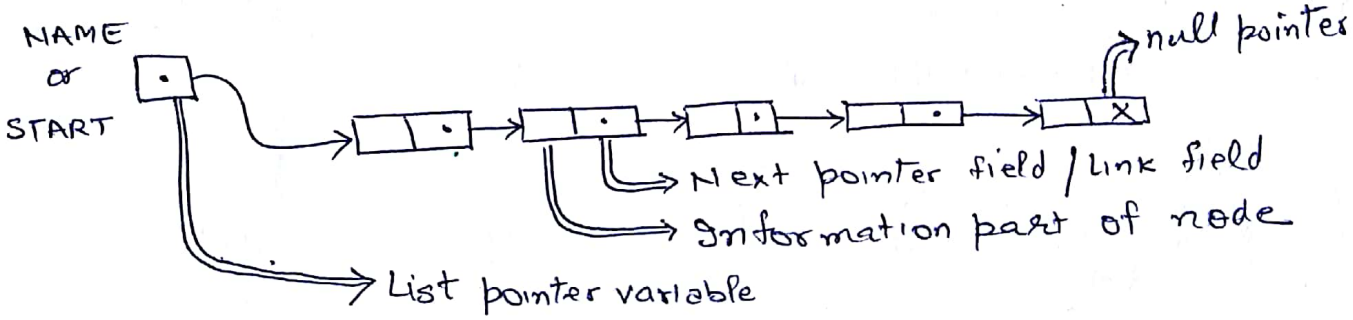
LINK LIST

Array

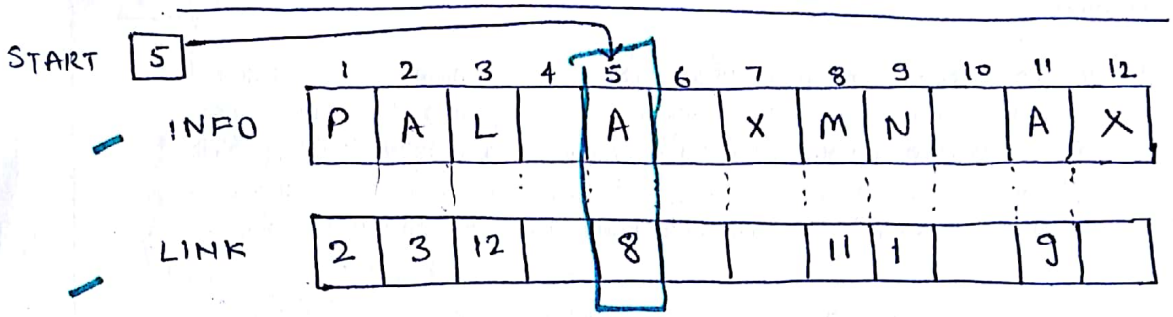
Advantage	Disadvantage
• Easy to compute the address of an element in an array	• Relatively expensive to insert and delete elements in an array • one can not simply double or triple the size of an array when additional space is required. (that why its called static data structure)

In case of link list, the successive elements in the list need not occupy adjecent space in memory. (as in case of array)
– This will make it easier to insert & delete elements in the list.

LINKED LIST



REPRESENTATION OF LL IN MEMORY



In this way, more (I love U mom)

More than one list may be maintained in the same linear arrays INFO & LINK. However, each list must have its own pointer variable giving the location of its first node.

TRAVERSING A LINK LIST

Traversing algorithm uses a pointer variable PTR, which points to the node that is currently being processed. Accordingly, LINK[PTR] points to the next node to be processed.

Thus

PTR := LINK[PTR]

moves the pointer to the next node in the list.

Algorithm LinkList_Traversal

Let LIST be linked list. This algo traverses LIST, applying PROCESS to each element of LIST. The variable PTR points to node currently being processed.

1. Set PTR := START
2. Repeat step 3 & 4 while PTR ≠ NULL
3. Apply PROCESS to INFO[PTR]
4. Set PTR := LINK[PTR]
   [END of step 2 loop]
5. EXIT

Procedure: COUNT (INFO, LINK, START, NUM)

1. Set NUM := 0
2. Set PTR := START
3. Repeat step 4 & 5 while PTR ≠ NULL
4. Set NUM := NUM + 1
5. Set PTR := LINK[PTR]
   [END of step 3 loop]
6. Return

SEARCHING A LINKED LIST

Let LIST be a linked list in memory, a specific ITEM of information is given.
There are two searching algorithm for finding the LOC location of NODE.

(a) LIST is Unsorted: Using a pointer variable PTR & comparing ITEM with the contents — INFO[PTR] of each node, one by one, of LIST.

— We update pointer PTR by PTR := LINK[PTR]

— We require two tests

(i) whether PTR = NULL

If not, (ii) check whether INFO[PTR] = ITEM

Accordingly, we use first test to control the execution of a loop, second test place inside the loop.

Algorithm SEARCH (INFO, LINK, START, ITEM, LOC)

1. Set PTR := START
2. Repeat Step 3 while PTR ≠ NULL:
3. If ITEM = INFO[PTR], then
   set LOC := PTR and exit.
   Else
   set PTR := LINK[PTR]
   [End of if structure]
   [End of step 2 loop]
4. Set LOC := NULL [unsuccessful]
5. EXIT

The complexity of this algorithm is same as that of the linear search algorithm.

(b) LIST is sorted

Now, however, we can stop once ITEM exceeds INFO[PTR].

Algorithm SEARCH_SOR (INFO, LINK, START, ITEM, LOC)

1. Set PTR := START
2. Repeat Step 3 while PTR ≠ NULL
3. If ITEM < INFO[PTR], then:
   set PTR := LINK[PTR]
   Else if ITEM = INFO[PTR], then:
   set LOC := PTR, & Exit
   Else
   set LOC := NULL, & Exit
   [End of If structure]
   [End of step 2 loop]
4. Set LOC := NULL
5. EXIT

Binary search algorithm cannot be applied to a sorted link list, since there is no way of indexing the middle element in the list.

GARBAGE COLLECTION

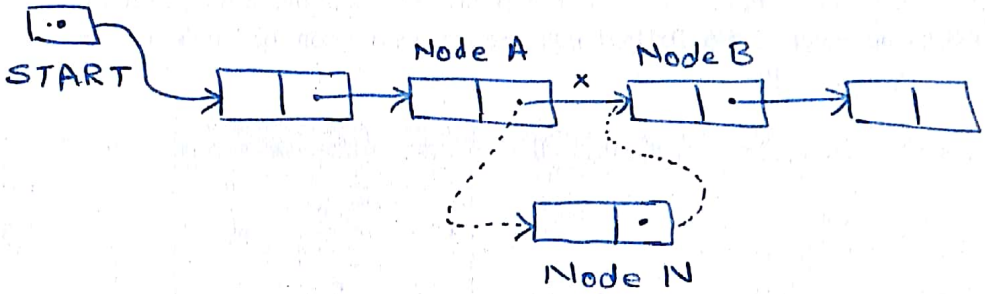
Suppose a memory space become reusable because a node is deleted, we want space to be available for future use. (one way to bring this into free-storage list <time consuming>)

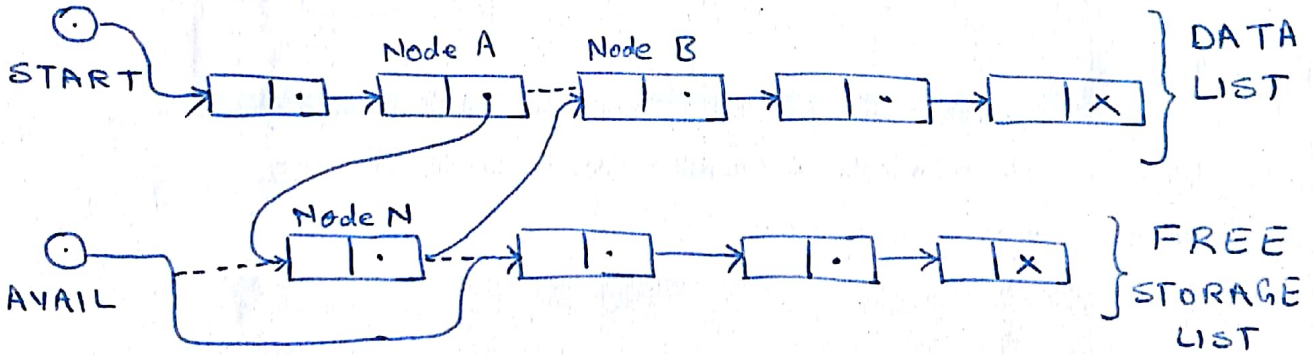
The operating system of computer may periodically collect all the deleted space into the free-storage list.

OVERFLOW & UNDERFLOW

* Overflow: New data are to be inserted into a data structure but there is no available space.
* Underflow: situation where one wants to delete data from data ~~sto~~ structure that is empty.

INSERTION INTO LINKED LIST





The three pointer fields are going to be change

(i) Next PTR of Node A points to Node N

(ii) AVAIL now points to NEXT AVAIL

(iii) Next PTR of Node N points to Node B

Two special case

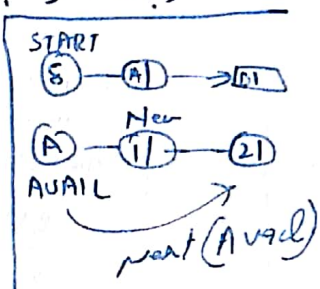
(i) If new node N is the first node, then START points to N

(ii) If new node N is the last node, then N will contain null pointer.

Insertion Algorithms : The main three situations

(i) Inserting a node at the beginning of the list,
(ii) Inserting after the node with given location
(iii) Inserting a node into a sorted list.

(i) Algorithm: Insert_first(INFO, LINK, START, AVIL, ITEM)



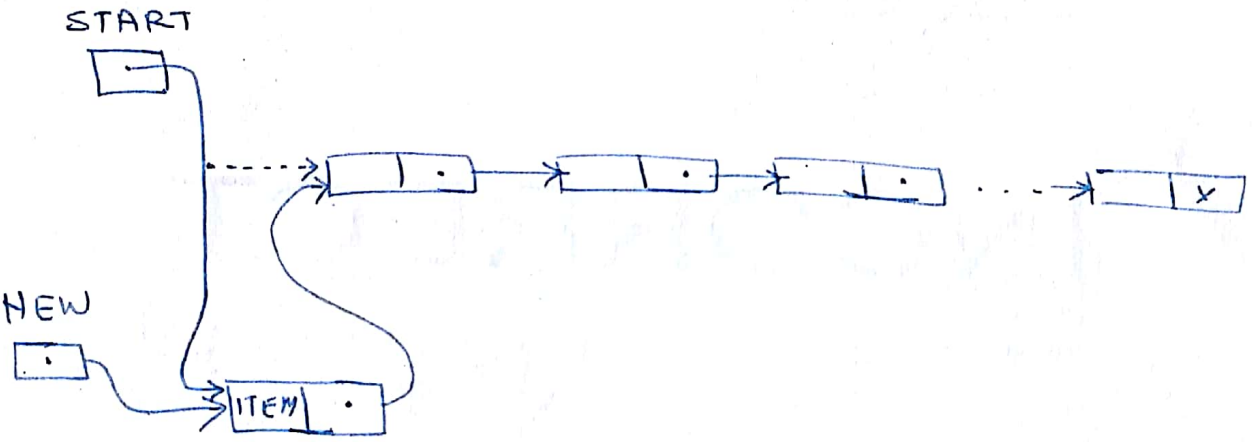
Overflow?
1. If AVAIL = NULL, then:
   Write: OVERFLOW and EXIT

Remove first node from AVAIL list
2. Set NEW := AVAIL
   & AVAIL := LINK[AVAIL]

Copy new data into node
3. Set INFO[NEW] := ITEM

New node points to original 1st node
4. Set LINK[NEW] := START

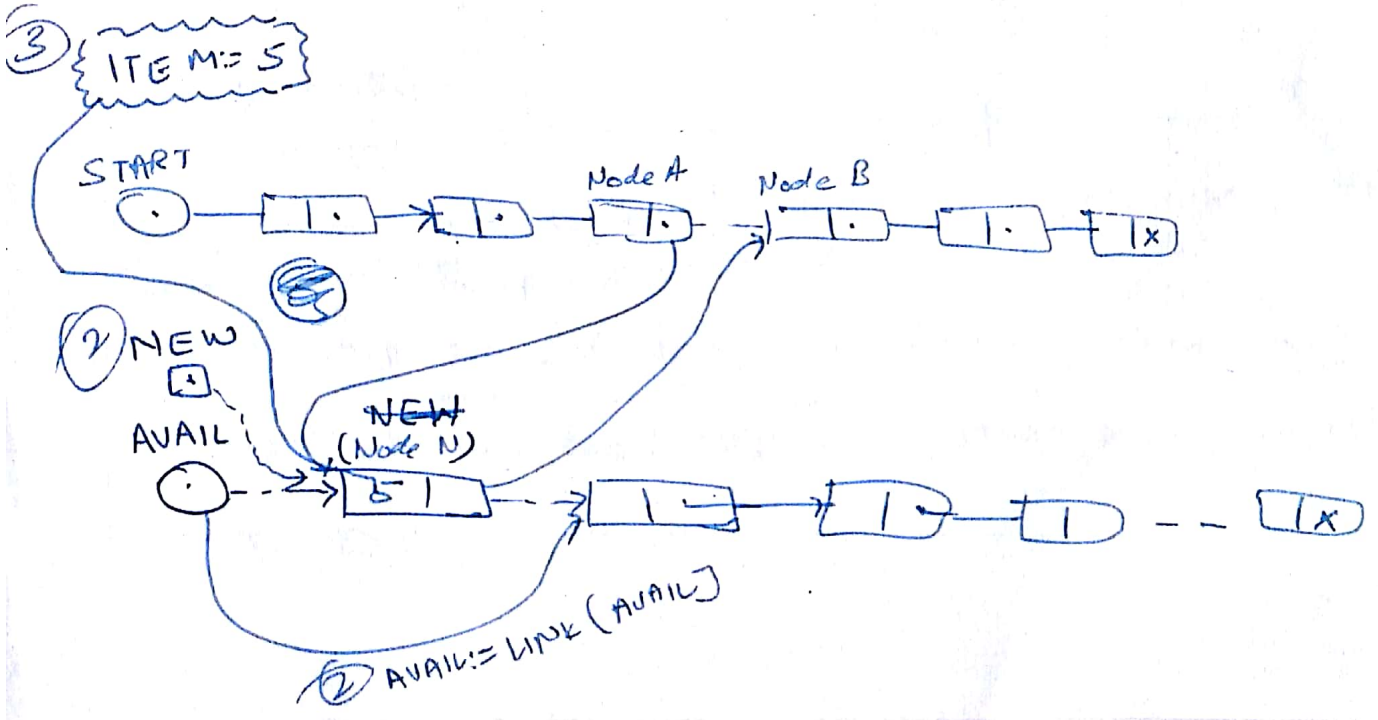
START points to first node
5. Set START := NEW
6. EXIT



(ii)

Algorithm : Ins_Loc (INFO, LINK, START, AVAIL, LOC, ITEM)

1. If AVAIL = NULL, then :
   WRITE : OVERFLOW, and EXIT
2. Set NEW := AVAIL
   & AVAIL := LINK [AVAIL]
3. Set INFO[NEW] := ITEM
4. If LOC = NULL, then :
   Set LINK[NEW] := START
   & START := NEW
   Else
   set LINK [NEW] := LINK [LOC]
   & LINK[LOC] := NEW
   [End of if structure of step 4]
5. EXIT



Inserting into a Sorted Linked List

Procedure: FIND (INFO, LINK, START, ITEM, LOC)

This procedure find the location LOC of the node in a sorted list such that

INFO[LOC] < ITEM or set LOC = NULL

1). If START = NULL, then: Set LOC := NULL, & Return

2). If ITEM < INFO[START], then Set LOC := NULL & Return

3). Set SAVE := START & PTR := LINK[START]

4) Repeat Step 5 & 6 while PTR ≠ NULL

5) If ITEM < INFO[PTR], then
Set LOC := SAVE & Return
[End of If structure]

6) Set SAVE := PTR & PTR := LINK[PTR].

[End of Step 4 loop]

7) Set LOC := SAVE

8) Return

Algorithm: Insert_sort_ll (INFO, LINK, START, AVAIL, ITEM)

This algo inserts ITEM into a sorted linked list.

1) Call FIND (INFO, LINK, START, ITEM, LOC)

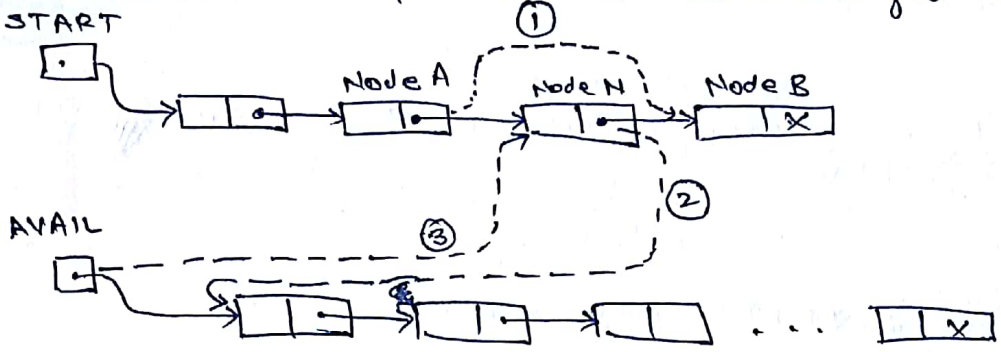
2) Call Ins_loc (INFO, LINK, START, AVAIL, LOC, ITEM)

3) EXIT

DELETION FROM A LINK LIST

→ (a) Deleting node following a given node
→ (b) Deleting with given ITEM of info.

Observe three pointer fields are changed as follows:



(1) The next pointer field of node A now points to node B, where node N previously pointed.

(2) The next pointer field of N now points to the original first node in the free pool, where AVAIL previously pointed

(3) AVAIL now points to the deleted node N.

(a) DELETING: FOLLOWING A GIVEN NODE

Algorithm : DEL_LL (INFO, LINK, START, AVAIL, LOC, LOCP)

// LOCP is the location of the node which precedes N (when N is first node : LOCP = NULL)

<1> If LOCP = NULL, then:

Set START := LINK[START] // Delete first node.

Else

Set LINK[LOCP] := LINK[LOC] // Delete node N

[End of If structure]

<2> Set LINK[LOC] := AVAIL & AVAIL := LOC

//Return deleted node to the AVAIL list]

<3> EXIT

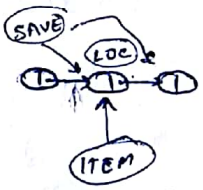
3) DELETING THE NODE WITH A GIVEN ITEM of information.

Procedure : FIND_N (INFO, LINK, START, ITEM, LOC, LOCP)

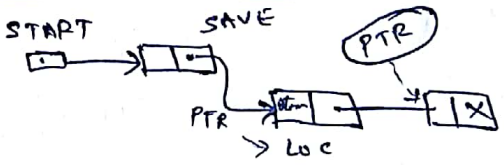
If list is empty
1. If START = NULL, then : Set LOC := NULL, & LOP := NULL & Return.
[End of if]

Node is 1st node
2. If INFO [START] = ITEM, then :
Set LOC := START & LOCP = NULL, & Return
[End of if structure of step 2]

3. Set SAVE := START & PTR := LINK [START]



4. Repeat step 5 & 6 while PTR ≠ NULL
5. If INFO [PTR] = ITEM, then :
set LOC := PTR & LOCP := SAVE & Return
[End of if structure]
6. Set SAVE := PTR & PTR := LINK [PTR]
[end of loop 4]
7. Set LOC := NULL
8. Return



Algorithm : DELETE (INFO, LINK, START, AVAIL, ITEM)

Find node
1. Call FIND_N (INFO, LINK, START, ITEM, LOC, LOCP)

node not in list
2. If LOC = NULL, then : Write : ITEM not in list, & Exit

node is 1st node
3. If LOCP = NULL, then : Set START := LINK [START]
Else Set LINK [LOCP] := LINK [LOC]
[End of if structure]

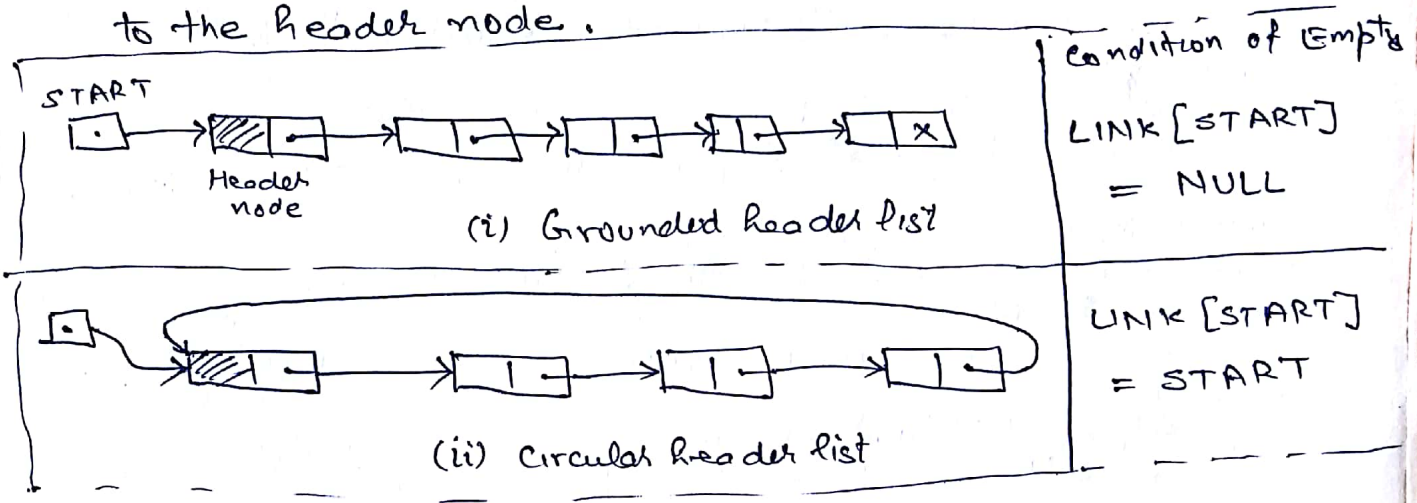
Return del node to AVAIL list
4. Set LINK [LOC] := AVAIL & AVAIL := LOC
5. EXIT

HEADER LINK LIST

A header linked list is a linked list which always contains a special node, called header node, at the begning of the list. Two kind of header link list:

(i) A grounded header list: is the header link list where the last node contains the null pointer

(ii) A circular header list: where last node points back to the header node.



Although our data may be maintained by header list in memory, the AVAIL list will always be maintained as ordinary linked list.

The first node in a header list is the node following the header node, and the location of the first node is LINK [START] not START.

→ circular header list are ~~frequently~~ used ~~instead of~~ or property

(i) The null pointer is not used, and hense all pointers contain valid pointer addresses

(ii) Every (ordinary) node has a predecessor, so first node may not require a special case

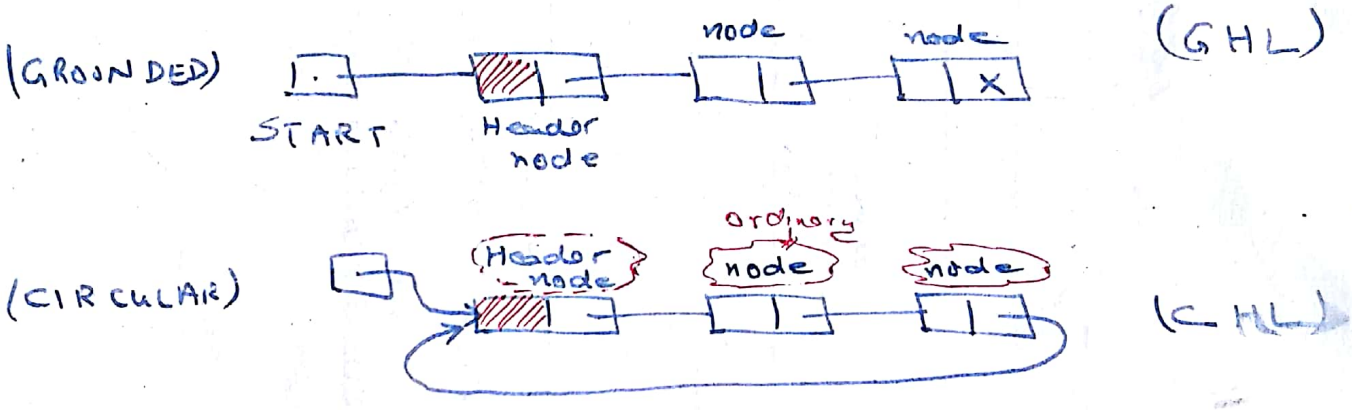
HEADER LINKED LISTS

It is the link list which always contains a special node, called the Header node, at the beginning of the list.

Two type of header link lists are:

(1) Grounded Header list: where last node contains the null pointer

(2) Circular Header list: where last node points back to the header node.



START always points to the header node.

LINK[START] = NULL; GHL is empty

LINK[START] = START; CHL is empty

Data may be maintained in memory by header list but the AVAIL list will always be maintained as ordinary linked list.

location of first node is LINK[START] not START.

Algorithm : TCHL (Traversed circular header LL)

conditions (i) begins with PTR = LINK[START] not START

(ii) end PTR = START not PTR = NULL

Let LIST be circular header link list in memory

The algorithm traverses LIST, applying an operation PROCESS to each node in LIST

1. Set PTR := LINK[START]
2. Repeat step 3 & 4 while PTR ≠ START
3. Apply process to INFO[PTR]
4. Set PTR := LINK[PTR]

[End of step 2 loop]

5. EXIT

PROPERTIES

The NULL pointer is not used, & hence all pointer contain valid addresses

Every (ordinary) node has a predecessor, so the first node may not require a special case

Procedure : Find_loc_CHL (INFO, LINK, START, ITEM, LOC, LOCP)

1. Set SAVE := START & PTR := LINK [START]

2. Repeat while INFO[PTR] ≠ ITEM & PTR ≠ START
   Set SAVE := PTR & PTR := LINK [PTR]
   [END of loop 2]

3. If INFO[PTR] = ITEM, then:
   Set LOC := PTR & LOCP := SAVE
   Else
   Set LOC := NULL & LOCP := SAVE
   [END of IF structure]

4. EXIT

Algorithm : Del_loc_CHL (INFO, LINK, START, AVAIL, ITEM)

1. [Use above procedure to find the location of N & it preceding node] Call Find_loc_CHL(....)

2. If LOC = NULL, then : Write: ITEM not in list, & EXIT

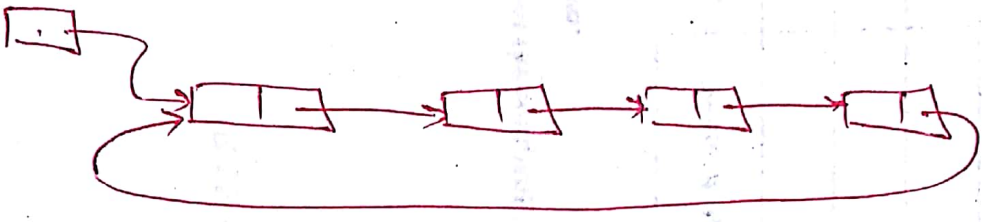
3. Set LINK[LOCP] := LINK[LOC] // Delete node

4. Set LINK[LOC] := AVAIL & AVAIL := LOC

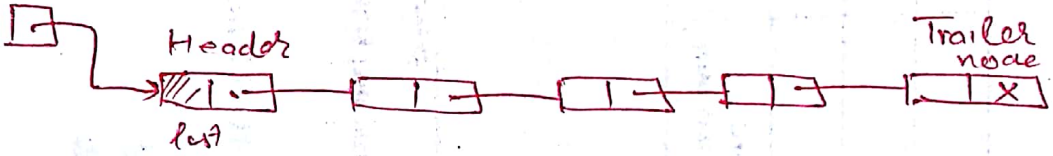
5. EXIT

Two other varient of link lists :

(1) Simple link list, whose last node points to first node



(2) List contain both header & trailer node



TWO-WAY LIST

Two-way list can be traversed into two direction.

This mean, one is able to delete N from the list without traversing any part of list.

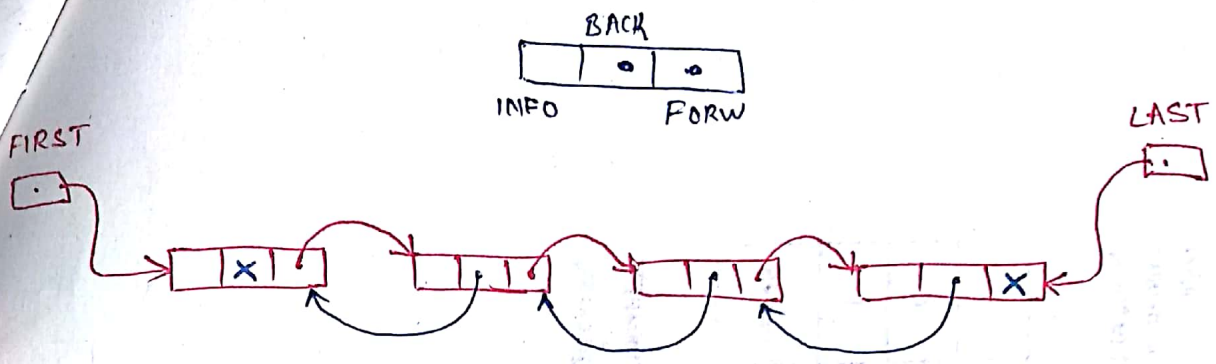
Provide imimideate access to the next & preceding node.

Each node is divided into three parts :

(i) An information field INFO which contains the data,

(ii) A pointer field FORW, contain location of next node.

& (iii) A pointer field BACK, contain location of preceding node.



: Two-way list :

It also requires two list pointer variables:

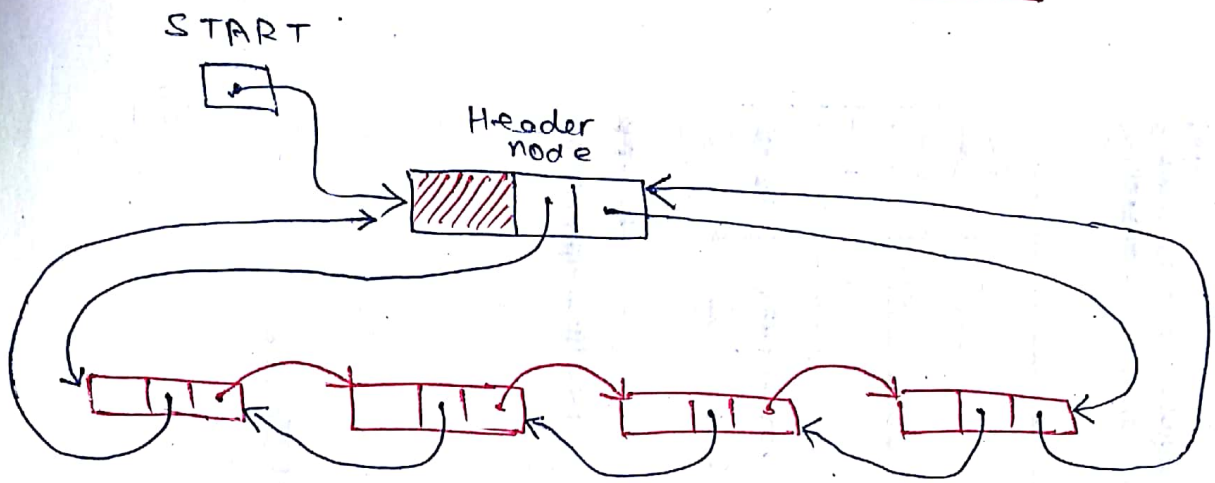
FIRST → points to the first node in the list.

LAST → " last node " " ".

Observe that null pointer appears in FORW field of the last node in the list ≠
Also in BACK field of the first node in the list.

Beside so many changes, the AVAIL list of available space in the arrays will still be maintained as one-way list - using FORW as pointer field - since we delete and insert nodes only at the beginning of the AVAIL list.

TWO-WAY HEADER LISTS



Two way Circular Header List

This list is circular because the two end nodes point back to the header node.

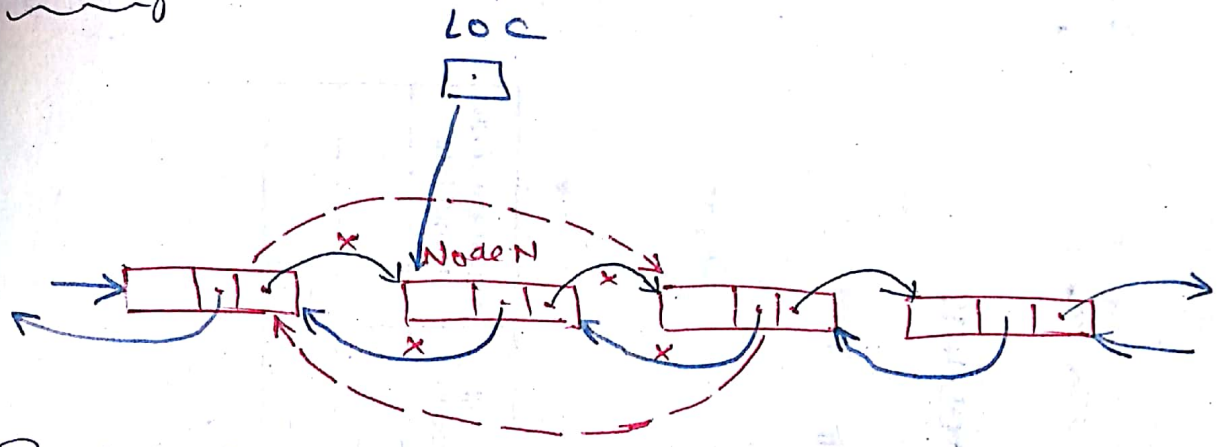
OPERATIONS ON TWO-WAY LIST

(i) Traversing : Same procedure as in ordinary / Header link list. (not any specific advange of two way)

(ii) Searching : Same as previous algorithms.

Advantage : We can search for ITEM in the backward direction if we have reason to suspect that ITEM appears near the end of the list.

ii) Deleting:



Pointer changes as:

FORW[BACK[LOC]] := FORW[LOC] &

BACK[FORW[LOC]] := BACK[LOC]

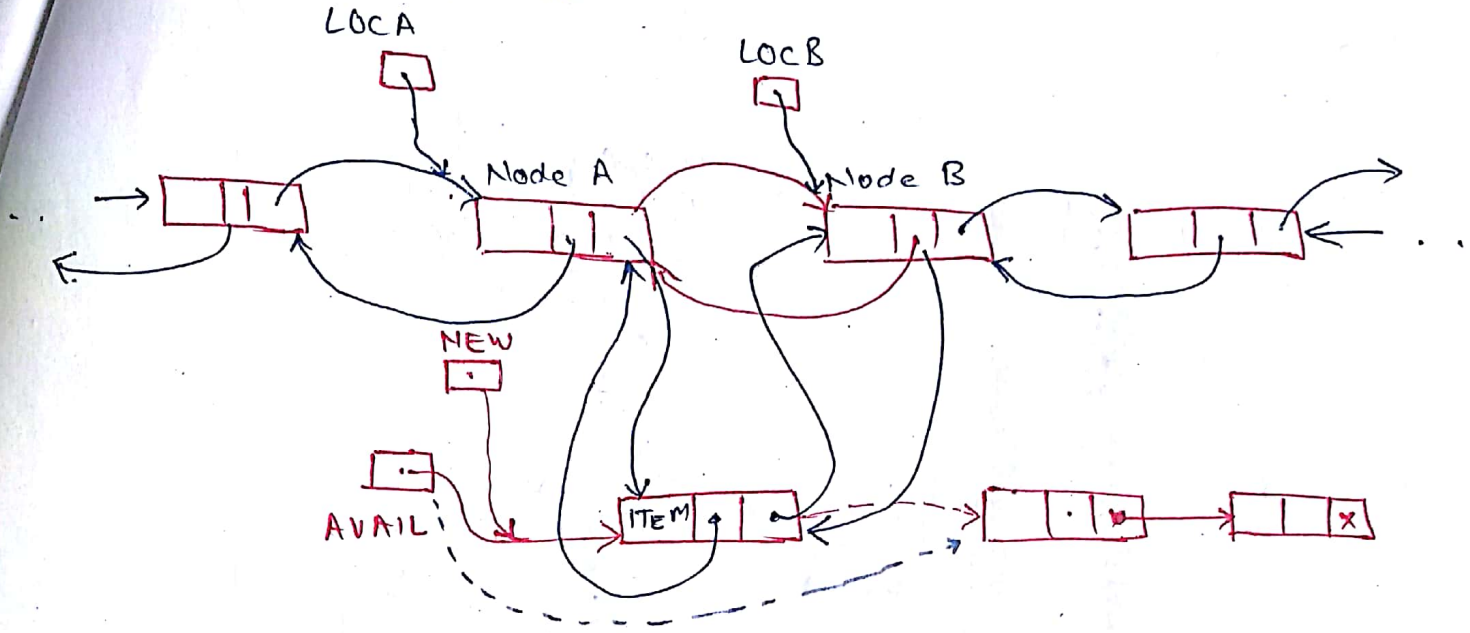
Algorithm: Del_TWL (INFO, FORW, BACK, START, AVAIL, LOC)

1. Set FORW[BACK[LOC]] := FORW[LOC] & { Deleting node in TWL }
   BACK[FORW[LOC]] := BACK[LOC]

2. Set FORW[LOC] := AVAIL & { Return node to AVAIL list }
   AVAIL := LOC

3 EXIT

(iv) Inserting : Inserting node N between node A & B



Suppose we are given the locations LOCA & LOCB of adjacent nodes A & B in LIST.

We set : (for removing first node from AVAIL)

NEW := AVAIL , AVAIL := FORW [AVAIL]

INFO [NEW] := ITEM

Insert into LIST by changing following four pointers :—

FORW [LOCA] := NEW ; FORW [NEW] := LOCB

BACK [LOCB] := NEW ; BACK [NEW] := LOCA

Algorithm : INS_TWL (INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)

1. If AVAIL = NULL then : Write : OVERFLOW, & EXIT

2. Set NEW := AVAIL
   AVAIL := FORW[AVAIL]
   INFO[NEW] := ITEM

3. Set FORW[LOCA] := NEW
   FORW[NEW] := LOC B
   BACK[LOCB] := NEW
   BACK[NEW] := LOC A

4. EXIT