LINKED LISTS

List :- It is a term used to refer to a linear collection of data items.
A list can be implemented either by using arrays or linked lists.

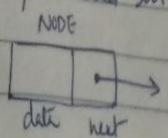
Drawbacks of arrays

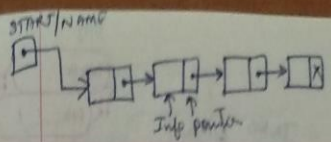
- A large block of memory is occupied by an array which may not be in use and it is difficult to increase the size of an array.
- A contigous block of memory is requird

Linked lists :- Linked lists are a linear collection of data elements that stores a group of values of the same data type. They are also known as dynamic data structures because successive elements are not stored at contigous memory allocation.

- NODE :- The data elements in a linked list are called nodes, each of which contains a pointer field pointing to the next node.

It is DS that contains data and link field encapsulated in a node





A pointer to the head of the list is used to gain access to the list itself and the end of the list is denoted by a NULL pointer.

The structure defined for single linked list is implemented as follows:

```
struct node                          // stores the element
{
    int data;                        // stores the address of next node
    struct node *next;
};
```

BASIC LINKED LIST OPERATIONS

NOTE :- A list that has no nodes is called a null list or empty list and is denoted by the null pointer in the variable START

Representation of Linked list in Memory

Let LIST → linked list

Two linear arrays are required in a LIST

INFO[K], LINK[K]

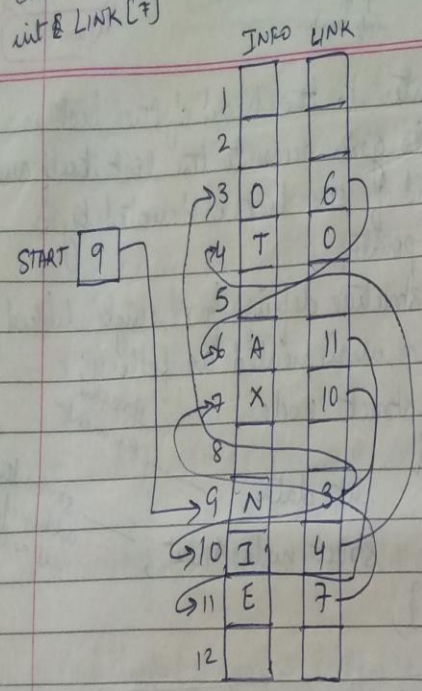
info part, link part

START → contains the location of the beginning of the list

NULL → indicates End of list

NULL = 0 → end of list

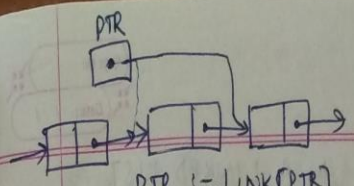
char INFO[7]
int & LINK[7]



BASIC LINKED LIST OPERATIONS

① TRAVERSING A LINKED LIST

Processing each node of the list exactly once is known as traversing a linked list.
Let LIST be a linked list in memory stored in a linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST.
PTR → pointer variable which points to the node i.e. currently being processed.
LINK[PTR] → points to the next node to be process



PTR := LINK[PTR]

LL-insert.c

Algorithm:

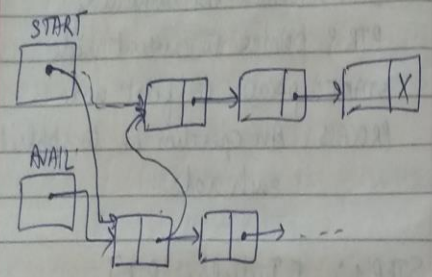
LIST: Linked list in memory
PTR: pointer to current node
START: pointer to first node
PROCESS: An operation i.e. to applied on each node.

STEP 1: [Initialize PTR]
Set PTR := START
STEP 2: [Repeat Step 3 and 4]
while PTR ≠ NULL
STEP 3: [Apply operation on node]
Apply PROCESS to INFO[PTR]
STEP 4: [Set PTR to point to next node]
Set PTR := LINK[PTR]
[End of Step 2 loop]
STEP 5: [Finished]
exit

② INSERTION IN A LINKED LIST

ⓐ Inserting at the Beginning of a list

Add beg. c



NOTE: *Sometimes new data are to be inserted into a DS but there is no available space. This situation is called overflow
i.e AVAIL = NULL

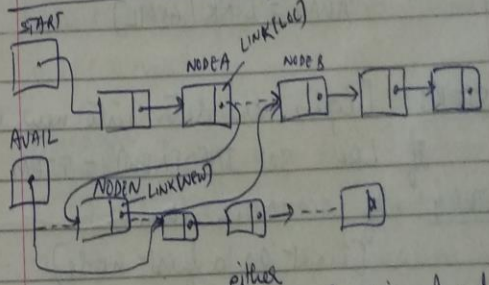
* The term underflow refers to a situation where one wants to delete a data item from DS that is empty. i.e START = NULL

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

STEP 1: [Checking for Overflow condition]
If AVAIL = NULL then
Write ("OVERFLOW")
Exit

STEP 2: [Remove first node from AVAIL LIST]
Set NEW := AVAIL
AVAIL := LINK[AVAIL]
[Copies new data into new node]
STEP 3: Set INFO[NEW] := ITEM
STEP 4: [New node now points to original first node]
Set LINK[NEW] := START
STEP 5: [Changes START so it points to the new node]
START := NEW
[Finished]
STEP 6: Exit.

ⓑ Inserting after a given node



Suppose LOC → A is the location of node A in a linked list or LOC = NULL (either)

Algorit

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

This algo. inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC = NULL

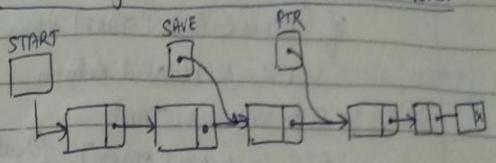
STEP 1: [Checking for overflow condition]
If AVAIL = NULL
Write ("OVERFLOW")
Exit.

STEP 2: [Remove first node from AVAIL List]
Set NEW := AVAIL
AVAIL := LINK [AVAIL]

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

STEP 4: If LOC = NULL then
[Insert as a first node]
Set LINK [NEW] := START
START := NEW
Else [Insert after new node with location LOC]
Set LINK [NEW] := LINK[LOC] and
LINK[LOC] := NEW
[End of If structure]
STEP [Finished] Exit

© Inserting into a Sorted linked list



ITEM is to be inserted between node A & B
i.e INFO(A) < ITEM ≤ INFO(B)
First find the LOC of node A.
Traverse the list using a ptr var. PTR and comparing INFO[PTR] with ITEM at each node. While traversing, keep track of the location of the preceding node by using a ptr var. SAVE ; SAVE := PTR & PTR := LINK[PTR]
When list is empty where ITEM ≤ INFO(START)
so LOC = NULL

Procedure for finding the Location of node A

Procedure: FINDA (INFO, LINK, START, ITEM, LOC)
This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ITEM or sets LOC = NULL

STEP 1: [List empty?]
If START := NULL then
Set LOC := NULL and Return.

STEP 2: [Special Case?]
If ITEM < INFO[START] then
LOC := NULL and Return
STEP 3: Set SAVE := START
PTR := LINK[START] [Initializes pointers]
STEP 4: [Repeat Step 5 and 6 while PTR ≠ NULL]
STEP 5: If ITEM < INFO[PTR] then
Set LOC := SAVE & Return
[End of if structure]
STEP 6: Set SAVE := PTR
PTR := LINK[PTR]
[End of Step 4 loop]
STEP 7: Set LOC := SAVE
STEP 8: Return

Algorithm: INSERT (INFO, LINK, START, AVAIL, ITEM)
This algo. inserts ITEM into a sorted linked list.

STEP 1: [Use procedure to find location of the node preceding ITEM]
Call FINDA (INFO, LINK, START, ITEM, LOC)

STEP 2: [Use Algo to insert ITEM after node with location LOC]
Call INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM).
STEP 3: Exit.

Example: Consider the alphabetized list of patients in the following fig. Determine the changes if Jones is added to the list of patients.

Sol:

Here ITEM = Jones & INFO = BED

(a) FINDA (BED, LINK, START, ITEM, LOC)

STEP 1: Since START ≠ NULL, control is transferred to step 2.

STEP 2: Since BED[5] = Adams < Jones, control is transferred to step 3.

STEP 3: SAVE = 5 & PTR = LINK[5] = 3

STEP 4: steps 5 and 6 are repeated as follows:

(a) BED[3] = Dean < Jones so SAVE = 3 & PTR = LINK[3] = 11

(b) BED[11] = Fields < Jones so SAVE = 11 & PTR = LINK[11] = 8

(c) BED[8] = Green < Jones so SAVE = 8 & PTR = LINK[8] = 10

(d) Since BED[10] = Kirk > Jones we have: LOC = SAVE = 8 and Return.

(b) INSLOC (BED, LINK, START, AVAIL, LOC, ITEM)

STEP 1: Since AVAIL ≠ NULL control is transferred to step 2

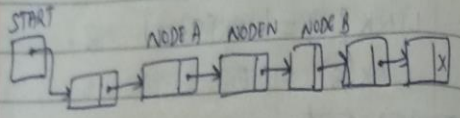
STEP 2: NEW = 10 and AVAIL = LINK[10] = 2

STEP 3: BED[10] = Jones

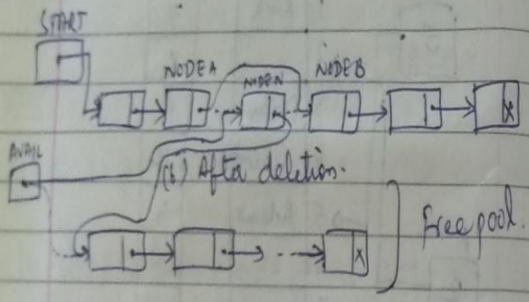
STEP 4: LOC ≠ NULL we have LINK[10] = LINK[8] = 1 & LINK[8] = NEW = 10

STEP 5: Exit

(3) DELETION FROM A LINKED LIST



(a) Before deletion of NODE N



(b) After deletion.

Two special cases

1. If the deleted node N is the first node in the list then START point to node B.
2. If deleted node N is the last node in the list, then node A will contain the NULL pointer.

(a) Deleting the Node following a Given node

Let LIST be a list in memory. Suppose LOC be the location of Node N in LIST. Let LOCP be the location of the node preceding node N. When N is the first node given LOCP = NULL.

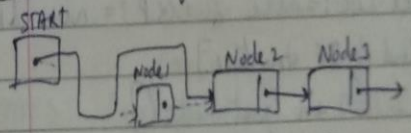
Algorithm: DEL (INFO, LINK, START, AVAIL, LOC, LOCP)

STEP 1: If LOCP = NULL then
  Set START := LINK[START] (deletes first node)
  Else
  Set LINK[LOCP] := LINK[LOC] (Deletes node)
  [End of If structure]

STEP 2: [Return deleted node to the AVAIL LIST]
  Set LINK[LOC] := AVAIL AVAIL := LOC

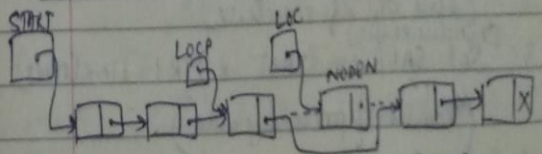
STEP 3: Exit

To delete the first node



START := LINK[START]

To delete the node N when N is not the first node



LINK[LOCP] := LINK[LOC]

⑥ Deleting the Node with a Given ITEM of Information

Note: Find the LOC of the node N containing ITEM & LOCP of the node preceding node N.

Procedure FINDB(INFO, LINK, START, ITEM, LOC, LOCP)

STEP 1: [List empty?]
If START = NULL then
Set LOC := NULL & LOCP := NULL & Return
[End of If structure]

STEP 2: [ITEM in first node?]
If INFO[START] = ITEM then
Set LOC := START & LOCP = NULL & Return.
[End of If structure]

STEP 3: Set SAVE := START & PTR := LINK[START] (Initialize pointers) (Location of preceding node)

STEP 4: Repeat Steps 5 and 6 while PTR ≠ NULL

STEP 5: If INFO[PTR] = ITEM then
Set LOC := PTR & LOCP := SAVE & Return
[End of If structure]

[Update pointers]

STEP 6: Set SAVE := PTR and PTR := LINK[PTR]
[End of step 4 loop]

STEP 7: Set LOC := NULL [Search Unsuccessful]

STEP 8: Return

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

STEP 1: [Use Procedure 5.9 to find the location of N & its preceding node]
Call FINDB(INFO, LINK, START, ITEM, LOC, LOCP)

STEP 2: If LOC = NULL then
Write "Item not in list"
Exit

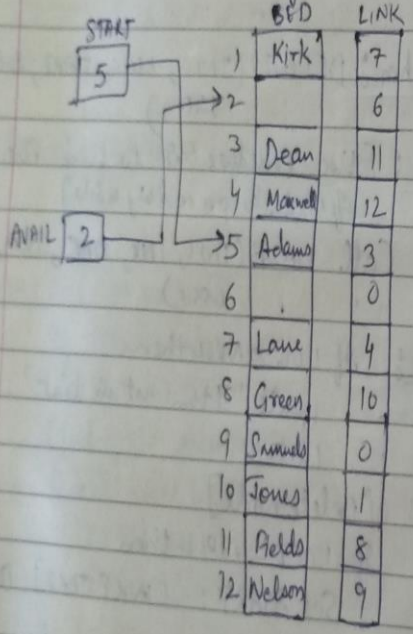
STEP 3: [Delete node]
If LOCP = NULL then
Set START := LINK[START]. (Deletes first node)
Else
Set LINK[LOCP] := LINK[LOC]
[End of If structure]

STEP 4: [Return deleted node to the AVAIL List]
Set LINK[LOC] := AVAIL and AVAIL := LOC

STEP 5 exit.

Example

Consider the list of patients in the given Fig. Delete the patient Green when he is discharged.



Here ITEM = Green INFO = BED,
START = 5, AVAIL = 2

(a) FINDB (BED, LINK, START, ITEM, LOC, LOCP)

1. Since START ≠ NULL control is transferred to Step2
2. Since BED[5] = Adams ≠ Green control is transferred to Step3.
3. Set Save = 5 and PTR = LINK[5] = 3
4. Step 5 and 6 are repeated as follows:

(a) BED[3] = Dean ≠ Green so
SAVE = 3 & PTR := LINK[3] = 11

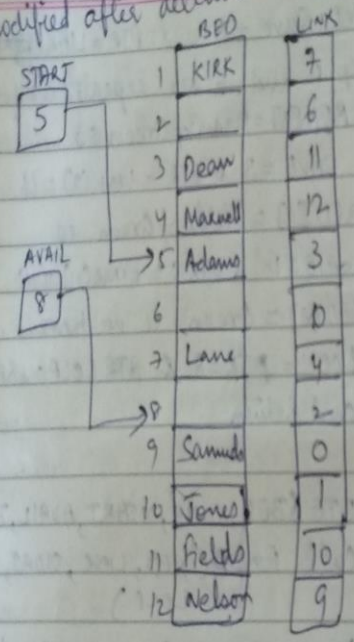
(b) BED[11] = Fields ≠ Green so
SAVE = 11 & PTR := LINK[11] = 8

(c) BED[8] = Green so we have
LOC := PTR = 8 & LOCP := SAVE = 11
and Return.

(b) DELETE (BED, LINK, START, AVAIL, ITEM)

1. Call FINDB(BED, LINK, START, ITEM, LOC, LOCP)
Hence LOC = 8 & LOCP = 11
2. Since LOC ≠ NULL control is transferred to Step3.
3. Since LOCP ≠ NULL then
LINK[11] := LINK[8] = 10
4. LINK[8] = 2 & AVAIL = 8
5 Exit

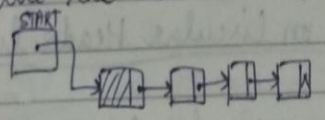
Modified after deletion



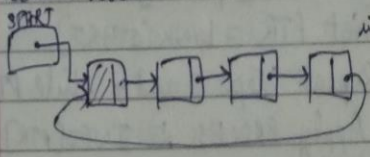
HEADER LINKED LISTS

A header linked list is a linked list which always contains a special node called the header node, at the beginning of the list. Types of widely used header lists.

① A grounded header list : is a list where the last node contains the null pointer.



② A circular header list : is a header list where the last node points back to the header node.



NOTE : The term node by itself refers to an ordinary node, not the header node when used with header list. Thus the first node in a header list is the node following the header node & location of the first node is LINK[START] ; not START.

Properties of Circular header lists

(1) The null pointer is not used & hence all pointers contain valid addresses.

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

Operations on Circular Header list

(1) Traversing a circular header list

Algorithm:

[Initializes pointer]
STEP 1: Set PTR := LINK[START]
STEP 2: Repeat Steps 3 and 4 while PTR ≠ START
STEP 3: Apply PROCESS to INFO[PTR]
STEP 4: [Update pointer]
Set PTR := LINK[PTR]
[End of Step 2 loop]
STEP 5: [Finished]
Exit

(2) Searching in Circular header list

Algorithm: SRCHHL(INFO, LINK, START, ITEM, LOC)

LIST is a circular header list in memory. This algo. finds the location LOC of the node where ITEM first appears in LIST or sets LOC = NULL

STEP 1: Set PTR := LINK[START]
STEP 2: Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START
Set PTR := LINK[PTR]
[End of loop]
STEP 3: If INFO[PTR] = ITEM, then
Set LOC := PTR
Else
Set LOC := NULL
[End of If Structure]
STEP 5: [Finished]
Exit

(3) Deletion in circular header list

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

This finds the location LOC of the first node N which contains ITEM & also the location LOCP of the node preceding N

[Initialize Pointers]

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

STEP 2: Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START

Set SAVE := PTR and PTR := LINK[PTR] [Update pointers]

[End of Loop]

STEP 3: If INFO[PTR] = ITEM then

Set LOC := PTR & LOCP := SAVE

Else

Set LOC := NULL & LOCP := SAVE

[End of If structure]

STEP 4: [Finished]

Exit

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

STEP 1: [Use procedure to find the location of N & its preceding node]

Call FINDBHL (INFO, LINK, START, ITEM, LOC, LOCP)

STEP 2: If LOC = NULL then

Write "ITEM not in list"

Exit

STEP 3: Set LINK[LOCP] := LINK[LOC] [Delete node]

STEP 4: [Return deleted node to the AVAIL list]

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

STEP 5: [Finished]

End

TWO-WAY LISTS

It is a linear collection of data elements called nodes where each node N is divided into three parts:

(1) An information field INFO which contains the data of N.

(2) A pointer field FORW which contains the location of the next node in the list.

(3) A pointer field BACK which contains the location of the preceding node in the list.

The list also requires two list pointers variable FIRST which points to the first node in the list & LAST which points to the last node in the list

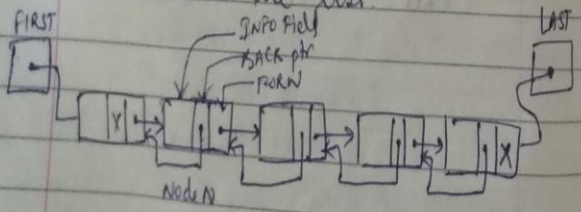


Fig: TWO WAY LIST

NOTE:

(1) Using var. FIRST & pointer field FORW, we can traverse a two-way list in forward direction as before

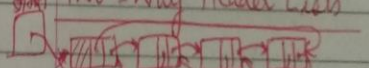
(2) Using the var LAST and ptr. field BACK we can also traverse the list in the backward direction.

(3) Two pointer arrays FORW and BACK are required instead of one LINK array pointer as in one way list

(4) Two list pointer var FIRST & LAST instead of one list pointer START.

(5) The List AVAIL of available spaces in the arrays will still be maintained as a one-way list using FORW as the ptr. field

Two-Way Header Lists



The advantages of a two way list and a circular header list may be combined into a two-way circular header list

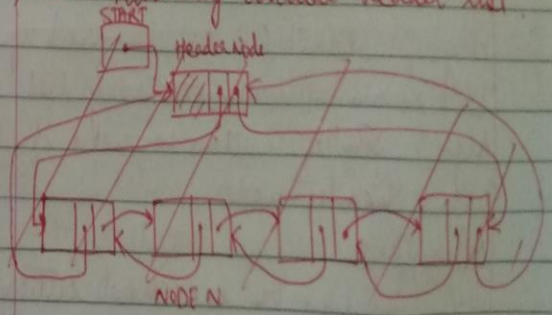


Fig: Two-Way Circular Header List

Operations on Two-Way Lists

① Traversing

Algorithm 1 can be used if LIST is an ordinary two way list

Algorithm: Traversing a Circular header list can be used if LIST contains a header node.

NOTE: There is no disadvantage that the data are organized as a two-way list rather than as a one-way list.

② Searching

Suppose we are given an ITEM of info and we want to find the location LOC of ITEM in LIST.

Algo: SEARCH( ) can be used if LIST is an ordinary two-way list.

Algo: SRCHHL ( ) can be used if LIST has a header node.

Advantage

The list can be searched either in forward or backward direction from depending upon the ITEM.

③ Deletion

Suppose we want to delete NODE N or at a location LOC in LIST.

BACK[LOC] & FORW[LOC] are the locations of the nodes which precede and follow node N

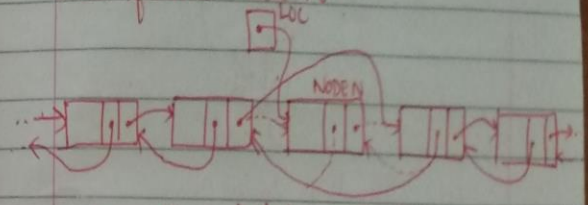


Fig: Deleting Node N

Following pointers updated

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

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

Algo: DELTWL (INFO, FORW, BACK, START, AVAIL, LOC)

STEP 1: [Delete node]
Set FORW[BACK[LOC]] := FORW[LOC] and
BACK[FORW[LOC]] := BACK[LOC]

STEP 2: [Return node to AVAIL list]
Set FORW[LOC] := AVAIL and
AVAIL := LOC

STEP 3: [Finished]
Exit

(4) Insertion

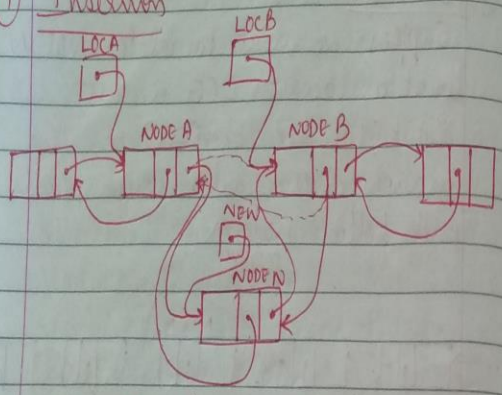


Fig: Inserting NODE N

Algo: INSTWL(INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)

STEP 1:- [OVERFLOW?]
If AVAIL = NULL then
Write "OVERFLOW"
Exit

STEP 2:- [Remove node from AVAIL list and copy new data into node]
Set NEW := AVAIL
AVAIL := FORW[AVAIL]
INFO[NEW] := ITEM

STEP 3:- [Insert node into list]
Set FORW[LOCA] := NEW
FORW[NEW] := LOCB
BACK[LOCB] := NEW
BACK[NEW] := LOCA

STEP 4:- [Finished]
Exit