

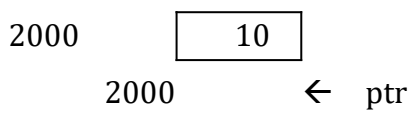
LINKED LISTS

Pointers and Linked Allocation

- **Pointer:** A pointer is a variable which contains an address of another variable in memory. It is declared by * indicator.
- We can create a pointer variable in C using following syntax:
- For example:

```
int *ptr;
```

Here, ptr is a pointer to integer data type.

- Suppose one variable called X having value 10 is stored at address (memory location) 2000
- 
- 2000 10
 ← ptr

- If we want a pointer ptr to point to the address 2000 then we have to use following syntax:

`ptr = &X;`

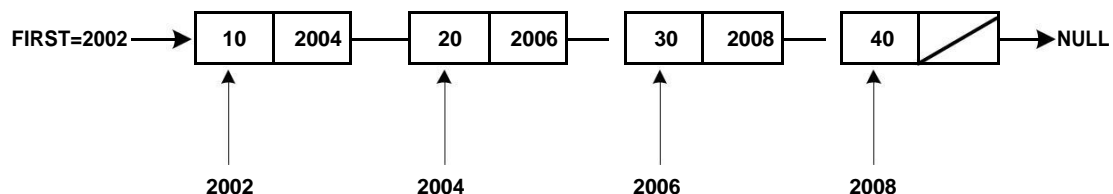
- If we want to access the value of variable X using pointer variable then we have to use the following syntax:

`Y = *ptr;` give 10 to variable Y.

- Thus the use of a pointer (link) to refer to the element of data structure implies that, *Elements which are logically adjacent need not be physically adjacent in Memory* is known as **linked allocation**.

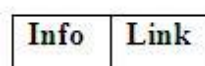
Linked lists & Sequential list

- A singly linked list is to expand each node to contain a link or pointer to the next node. This is also called as a one way chain.
- For example:



- First contain the address of the first node of the list.
- Each node in the list consists of two parts:
 1. Information (INFO)
 2. Address or pointer to next node (LINK).

Node



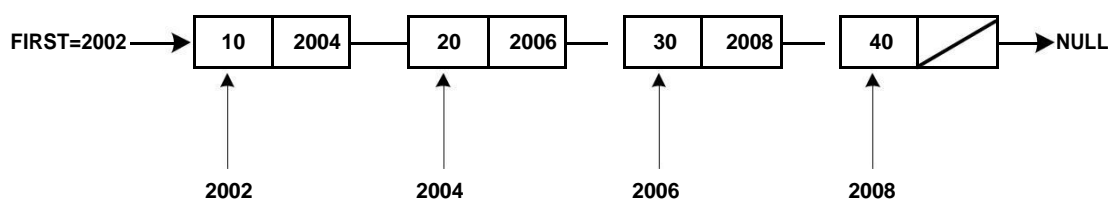
- In link list each node contains a **Link** and **INFO**. **INFO** part contains the actual element of list. **Link** or pointer contains the address of next node in the list. Last node indicates end of list contains a special value, known as a “**NULL**”. **Link(Next)=NULL**.

Difference between Linked list and Sequential list

Linked List	Sequential List
1. In linked list number of elements in the list is not fixed.	1. in sequential list number of elements in list is fixed.
2. In linked list allocation of memory is done at runtime.	2. In sequential list allocation of memory is done at compile time.
3. Insertion and deletion operation are very easy and less time consuming in linked list.	3. Insertion and deletion operation are very lengthy and time consuming in sequential list
4. In linked list we require pointer variable which occupies extra memory space.	4. In sequential list there is no need to use pointer variable so it does not occupies extra memory space.
5. Searching in a linked list is very time consuming because we have to traverse entire list even if all the elements in the list are sorted.	5. Searching is less time consuming in sequential list because we can use binary search method to search an element which is very efficient.
6. In linked list the elements which are logically adjacent need not be physically adjacent.	6. In sequential list elements are logically and physically adjacent to each other.
7. It is very easy to join or split two lists.	7. It is very difficult to split or join two lists.

Singly linked list

- A list in which each node contains a link or pointer to next node in the list is known as singly linked list or one way chain.
- Representation of Singly linked list is shown below:



- Here, the variable FIRST contains an address of the first node in the list.

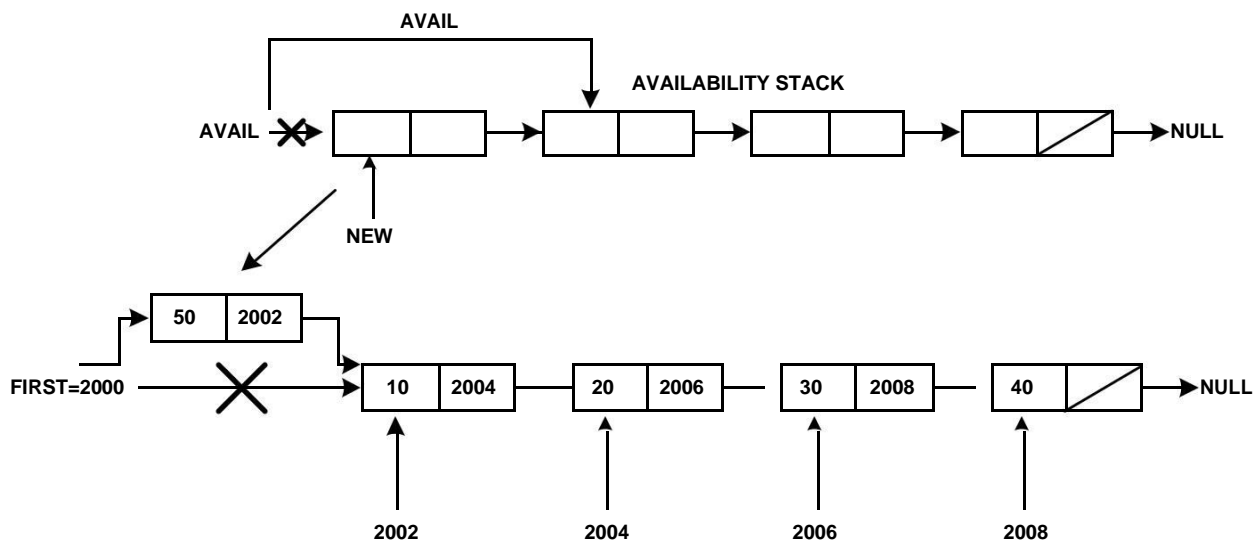
- The last node of the list does not have any successor node, so it does not contain any address in its link field. In such case a NULL value is stored as an address. NULL indicates end of the list.

Operations on Linked list

- Followings are the operations that can be performed on linked list.
 1. Insert new node at beginning of the list
 2. Insert new node at end of the list
 3. Insert new node at specific location
 4. Insert new node in ordered link list.
 5. Delete node from any specific position in the list.
 6. Copy of the list.

Algorithms for Singly linked list.

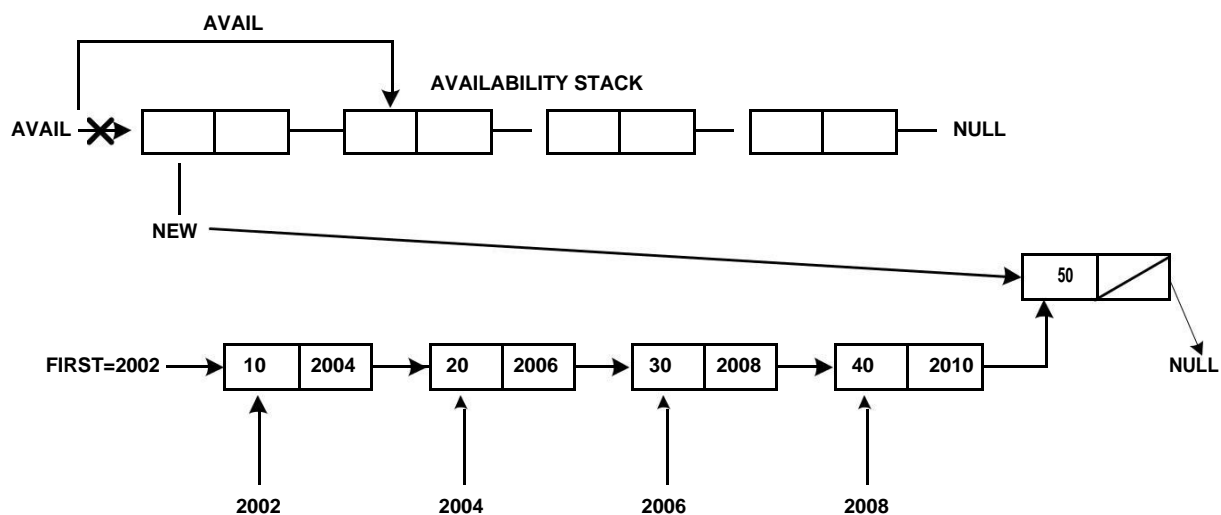
1. Algorithm to insert new node at beginning of the linked list.



INSERTBEG (VAL,FIRST)

- This function inserts a new element VAL at the beginning of the linked list.
- FIRST is a pointer which contains address of first node in the list.
 1. [Check for availability stack underflow]
If AVAIL = NULL then
Write "Availability stack underflow"
Return
 2. [Obtain address of next free node]
 $NEW \leftarrow AVAIL$
 3. [Remove free node from availability stack]
 $AVAIL \leftarrow LINK(AVAIL)$
 4. [Initialize node to the linked list]
 $INFO(NEW) \leftarrow VAL$
 $LINK(NEW) \leftarrow FIRST$
 5. [Assign the address of the Temporary node to the First Node]
 $FIRST \leftarrow NEW$
 6. [Finished]
Return (FIRST)

2. Algorithm to insert new node at end of the linked list.



INSERTEND (VAL,FIRST)

- This function inserts a new element VAL at the end of the linked list.
- FIRST is a pointer which contains address of first node in the list.
 1. [Check for availability stack underflow]
If AVAIL = NULL then
Write "Availability stack underflow"
Return
 2. [Obtain address of next free node]
 $NEW \leftarrow AVAIL$
 3. [Remove free node from availability stack]
 $AVAIL \leftarrow LINK (AVAIL)$
 4. [initialize field of new node]
 $INFO (NEW) \leftarrow VAL$
 $LINK (NEW) \leftarrow NULL$
 5. [If list is empty?]
If FIRST = NULL then
 $FIRST \leftarrow NEW$
 6. [initialize search for last node]
 $SAVE \leftarrow FIRST$
 7. [Search end of the list]
Repeat while $LINK (SAVE) \neq NULL$
 $SAVE \leftarrow LINK (SAVE)$
 8. [Set LINK field of last node to NEW]
 $LINK (SAVE) \leftarrow NEW$
 9. [Finished]

3. Algorithm to insert new node at specific location.

INSPOS (VAL, FIRST, N)

- This function inserts a new element VAL into the linked list specified by address N.
 - FIRST is a pointer which contains address of first node in the list.
1. [Check for availability stack underflow]
If AVAIL = NULL then
Write "Availability stack underflow"
Return
 2. [Obtain address of next free node]
NEW ← AVAIL
 3. [Remove free node from availability stack]
AVAIL ← LINK (AVAIL)
 4. [initialize field of new node]
INFO (NEW) ← VAL
 5. [If list is empty?]
If FIRST = NULL then
LINK (NEW) ← NULL
FIRST ← NEW
 6. [Search the list until desired address found]
SAVE ← FIRST
Repeat while LINK (SAVE) ≠ NULL and SAVE ≠ N
PRED ← SAVE
SAVE ← LINK (SAVE)
 7. [Node found]
LINK (PRED) ← NEW
 8. [Finished]
Return (FIRST)

4. Algorithm to insert new node in ordered linked list.

INSORD (VAL, FIRST)

- This function inserts a new element VAL into the ordered linked list.
- FIRST is a pointer which contains address of first node in the list.
 1. [Check for availability stack underflow]
If AVAIL = NULL then
Write "Availability stack underflow" Return
 2. [Obtain address of next free node]
 $NEW \leftarrow AVAIL$
 3. [Remove free node from availability stack]
 4. [initialize field of new node]
 $INFO (NEW) \leftarrow VAL$
 5. [If list is empty?]
If FIRST = NULL then
 $LINK (NEW) \leftarrow NULL$
 $FIRST \leftarrow NEW$
 6. [Does the new node precede all nodes in the list?]
If $INFO (NEW) \leq INFO (FIRST)$ then
 $LINK (NEW) \leftarrow FIRST$
 $FIRST \leftarrow NEW$
 7. [Initialize search pointer]
 $SAVE \leftarrow FIRST$
 8. [Search for predecessor of new node]
Repeat while $LINK (SAVE) \neq NULL$ and $INFO (LINK (SAVE)) \leq INFO (NEW)$
 $SAVE \leftarrow LINK (SAVE)$
 9. [Set LINK field of new node and its predecessor]
 $LINK (NEW) \leftarrow LINK (SAVE)$
 $LINK (SAVE) \leftarrow NEW$
 10. [Finished]
Return (FIRST)

7. Algorithm to delete specific node from linked list.

DELETE(X,FIRST)

This function deletes a node from specific location from the list.
FIRST is a pointer which contains address of first node in the list.
TEMP is used to find the desired node
PRED keeps track of the predecessor of TEMP

1. [Check for empty list]
If FIRST = NULL then
Write "List is empty"
Return
2. [Initialize Search for X]
$$\text{TEMP} \leftarrow \text{FIRST}$$
3. [Find X]
Repeat thru step 5 while LINK (TEMP) \neq X and
LINK (TEMP) \neq NULL
4. [Update Predecessor marker]
$$\text{PRED} \leftarrow \text{TEMP}$$
5. [Move to next node]
$$\text{TEMP} \leftarrow \text{LINK (TEMP)}$$
6. [End of the list?]
If TEMP \neq X then
Write "Node not found"
Return
7. [Delete X]
If X=FIRST(Is X the first node?)
then
$$\text{FIRST} \leftarrow \text{LINK (FIRST)}$$

else
$$\text{LINK (PRED)} \leftarrow \text{LINK (X)}$$
8. [Return node to availability area]
$$\text{LINK (X)} \leftarrow \text{AVAIL}$$

$$\text{AVAIL} \leftarrow \text{X}$$

Return

9. Algorithm to copy linked list

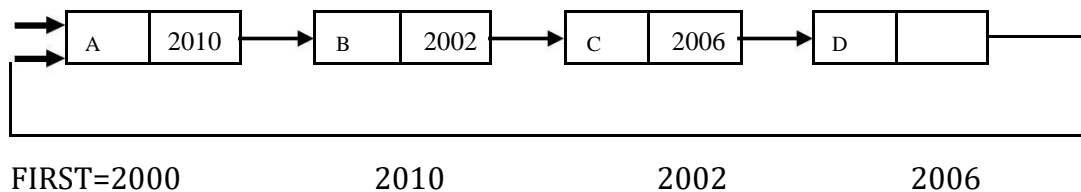
COPY(FIRST)

- This function copy one list into another list.
- FIRST is a pointer which contains address of first node in the list.
- BEGIN is a pointer which points to the address of first node in the New list.

```
1. [Empty List?]
   If FIRST = NULL then
       BEGIN ← NULL
2. [Copy First Node]
   If AVAIL = NULL then
       Write "Availability stack underflow"
       Return(0)
   Else
       NEW ← AVAIL
       AVAIL ← LINK(AVAIL)
       FIELD(NEW) ← INFO(FIRST)
       BEGIN ← NEW
3. [Initialize traversal]
   SAVE ← FIRST
4. [Move to next node if not at end of the list]
   Repeat thru step 6 while LINK(SAVE) ≠ NULL
5. [Update predecessor and save pointers]
   PRED ← NEW
   SAVE ← LINK(SAVE)
6. [Copy node]
   If AVAIL = NULL then
       Write "availability stack underflow"
       Return(0)
   Else
       NEW ← AVAIL
       AVAIL ← LINK(AVAIL)
       FIELD(NEW) ← INFO(SAVE)
       PTR(PRED) ← NEW
7. [Set link of last node and return]
   PTR(NEW) ← NULL
   Return(Begin)
```

Circular Linked list

- A list in which last node contains a link or pointer to the first node in the list is known as circular linked list.
- Representation of circular linked list is shown below:



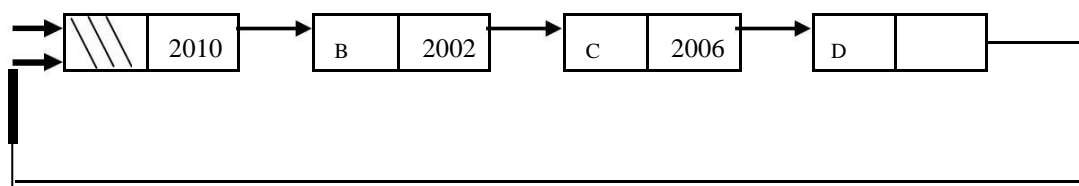
➤ **Advantage of circular linked list over singly linked list:**

1. In singly linked list the last node contains NULL address. So if we are at middle of the list and want to access the first node we can not go backward in the list. Thus every node in the list is not accessible from given node. While in Circular linked list last node contains an address of the first node so every node in the list is accessible from given node.
2. In singly linked list if we want to delete node at location X, first we have to find out the predecessor of the node. To find out the predecessor of the node we have to search entire list from starting from FIRST node in the list. So in order to delete the node at Location X we also have to give address of the FIRST node in the list. While in Circular linked list every node is accessible from given node so there is no need to give address of the first node in the list.
3. Concatenation and splitting operations are also efficient in circular linked list as compared to the singly linked list.

➤ **Disadvantage:**

1. Without some care in processing it is possible to get into the infinite loop. So we must be able to detect end of the list.
2. To detect the end of the list in circular linked list we use one special node called the HEAD node.
3. Such a circular linked list with HEAD node is shown below:

HEAD



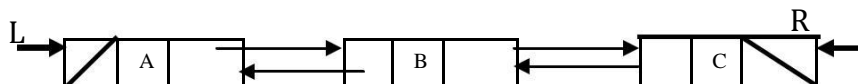
Doubly Linked list

- In Singly linked list we are able to traverse the list in only one direction. However in some cases it is necessary to traverse the list in both directions.
- This property of linked list implies that each node must contain two link fields instead of one link field.
- One link is used to denote the predecessor of the node and another link is used to denote the successor of the node.
- Thus each node in the list consist of three fields:
 1. Information
 2. LPTR
 3. RPTR

NODE

LPTR	INFO	RPTR
-------------	-------------	-------------

- A list in which each node contains two links one to the predecessor of the node and one to the successor of the node is known as doubly linked linier list or two way chain.
- Representation of doubly linked linier list is shown below:



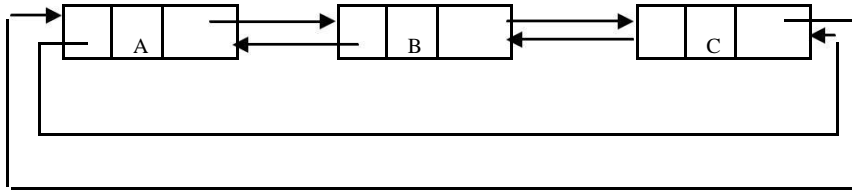
- L is a pointer denotes the left most node in the list.
- R is a pointer denotes the right most node in the list.
- The left link of the left most node and right link of right most node are set to NULL to indicate end of the list in each direction.
- **Advantage:**
 1. In singly linked list we can traverse only in one direction while in doubly linked list we can traverse the list in both directions.
 2. Deletion operation is faster in doubly linked list as compared to the singly linked list.

Operations on Doubly linked list

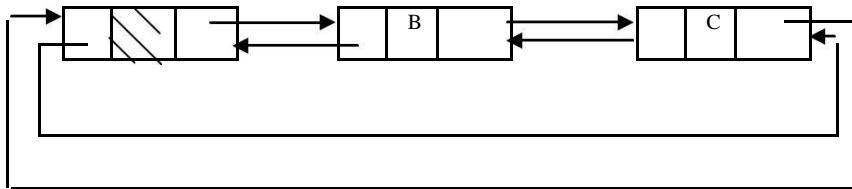
- Followings are the operations that can be performed on doubly linked list.
 1. Insert new node in doubly linked list
 2. Delete node from doubly linked list
 3. Traverse doubly linked list.

Doubly linked circular list

- A doubly linked list in which left link of the left most node points to the right most node and right link of the right most node points to the left most node is known as Doubly circular linked list.
- Representation of doubly circular linked list shown below:



- **Advantage:**
 1. Every node is accessible from given node.
- **Disadvantage:**
 1. Without some care in processing it is possible to get into the infinite loop. So we must be able to detect end of the list.
 2. To detect the end of the list in doubly circular linked list we use one special node called the HEAD node.
 3. Such a doubly circular linked list with HEAD node is shown below:



Algorithms for Doubly linked list

1. Algorithm to insert new node in doubly linked list

DOUBLEINS (L, R, M, X)

- This function inserts an element into double linked list.
- L is a pointer which contains address of left most node in the list.
- R is a pointer which contains address of right most node in the list.
- X contains the value to be inserted in to the list
 1. [Create new node] If
AVAIL = NULL then
Write "Availability stack underflow"
Else
NEW ← AVAIL
AVAIL ← LINK (AVAIL)
 2. [Copy information field]

INFO (NEW) \leftarrow X

3. [insertion into an empty list]

If R = NULL then

LPTR (NEW) \leftarrow RPTR (NEW) \leftarrow NULL

L \leftarrow R \leftarrow NEW

Return

4. [left most insertion]

If M = L then

LPTR (NEW) \leftarrow NULL

RPTR (NEW) \leftarrow M

LPTR (M) \leftarrow NEW

L \leftarrow NEW

Return

5. [insert in middle]

LPTR (NEW) \leftarrow LPTR (M)

RPTR (NEW) \leftarrow M

LPTR (M) \leftarrow NEW

RPTR (LPTR (NEW)) \leftarrow NEW

6. [Finished]

2. Algorithm to delete node from doubly linked list.

➤ **DOUBLEDEL(L, R, OLD)**

➤ This function inserts an element into double linked list.

➤ L is a pointer which contains address of left most node in the list.

➤ R is a pointer which contains address of right most node in the list.

➤ OLD contains address of node which we want to delete.

1. [Underflow]

If R = NULL then

Write "Underflow"

2. [Delete Node]

If L = R then

L \leftarrow R \leftarrow NULL

Else if OLD = L

```

    then  $L \leftarrow \text{RPTR}(L)$ 
     $\text{LPTR}(L) \leftarrow \text{NULL}$ 
    Else if  $\text{OLD} = R$ 
    then  $R \leftarrow \text{LPTR}(R)$ 
     $\text{RPTR}(R) \leftarrow \text{NULL}$ 
    Else
     $\text{RPTR}(\text{LPTR}(\text{OLD})) \leftarrow \text{RPTR}(\text{OLD})$ 
     $\text{LPTR}(\text{RPTR}(\text{OLD})) \leftarrow \text{LPTR}(\text{OLD})$ 
3. [Return Deleted
   node]
   Restore (OLD)
   Return

```

Application of Linked list

- Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations.
- The "data" field of a node can be another linked list. By this device, one can construct many linked data structures with lists; this practice originated in the Lisp programming language, where linked lists are a primary (though by no means the only) data structure, and is now a common feature of the functional programming style.
- Sometimes, linked lists are used to implement associative arrays, and are in this context called **association lists**. There is very little good to be said about this use of linked lists; they are easily outperformed by other data structures such as self-balancing binary search trees even on small data sets. However, sometimes a linked list is dynamically created out of a subset of nodes in such a tree, and used to more efficiently traverse that set.
- Polynomial manipulation
- Linked dictionary
- Multiple precision arithmetic