CHAPTER 5

LINKED LIST

"The whole is equal to the sum of its parts". **- *Euclid***

KEY FEATURES

- 🕮 Singly Linked List
- 🕮 Doubly Linked List
- 🕮 Circular Linked List
- 🕮 Linked Stack
- 🕮 Linked Queue
- 🕮 Polynomial Representation

Link list, as the name suggests, is a linear list of linked elements. Like arrays, linked list represents another linear data structure. Arrays are very commonly useful data structure in most of the programming languages. Since it has several limitations and drawbacks; therefore, an alternative approach is required. These limitations can be overcome by using Linked List data structure.

Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation developed linked lists as the primary data structure in 1955–1956 for their Information Processing Language.

Limitations of Array

The array is the most common data structure used to store a collection of homogeneous elements. In most languages, arrays are convenient to declare, and fast to access any element in a constant amount of time. The address of an element is computed as an offset from the start of the array which only requires one multiplication and one addition.

In many applications, the array is not suitable as it has some drawback. The drawbacks of Array are listed below:

- The maximum size of the array needs to be predicted beforehand. One cannot change the size of the array after allocating memory, but, many applications require resizing. Most often this size is specified at compile time with a simple declaration. The size of the array can be deferred until the array is created at runtime, but after that it remains fixed. When arrays are allocating dynamically from the heap and then one can dynamically resize it with realloc(), but that requires some real programmer effort.
- Most of the space in the array is wasted when programmer allocates arrays with large size. On the other hand, when program ever needs to process more than the specify size then the code breaks.
- Storage of the array must be available contiguously. Required storage not always immediately available.
- Insertion and deletion operation may be very slow. The worst case occurs when the first element is to be deleted or inserted. Almost all the elements of the array need to be moved. On an average about half the elements of the array need to be moved. Thus, the time complexity depends on the total no of elements rather than the actual operation.
- Joining and splitting of two or more arrays is difficult.

LINKED LIST

In arrays, there is always a fixed relationship between the addresses of two consecutive elements as all the items of an array must be stored contiguously. However, note that this contiguity requirement makes expansion or contraction of the size of the array difficult. In linked list, data items may be scattered arbitrarily all over the memory, but we can achieve fast insertion and deletion in a dynamic situation.

Definition: A linked list is a linear ordered collection of finite homogeneous data elements called node, where the linear order is maintained by means of links or pointers.

A linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list structure is created by the use of pointers to connect all its nodes together like the links in a chain.

In an array if the address of one element is known, addresses of all other elements become automatically known. Since, in a linked list, there is no relationship between the addresses of elements, each element of a linked list must store explicitly the address of the element next to it.

Table 5.1: Difference between Array and Linked list

Property	Array	Linked List
Storage	Storage of the array must be available contiguously.	Storage need not be contiguous.
Memory utilization	The size of the array needs to be predicted beforehand because memory allocation is done in advance.	Memory of linked list is not pre-allocated, memory is allocated whenever it is required.
Memory utilization	Not necessary for storing addresses of any element.	Extra memory space is necessary for storing addresses of the next node
Change of size	The array size is fixed, extend or shrink not possible during the execution of a program	Linked list may extend or shrink during the execution of a program
Insertion/ deletion	Insertion/deletion operations are slow, half of the elements are required to move on an average	Insertion/deletion operations are performed very fast, in a constant amount of time
Searching	Linear searching, binary searching, interpolation searching are possible	Binary searching, interpolation searching not possible, only linear searching is possible
Access element	Fast access to any element in a constant amount of time.	To access any element in a linked list, traversing is required.
Joining/ splitting	Joining and splitting of the two arrays is difficult.	Joining and splitting of two linked list is very easy.

Advantages of linked list

Linked lists have many advantages. Some of the very important advantages are:

i) Linked list are dynamic data structures. That is, they can extend or shrink during the execution of a program.

ii) Storage need not be contiguous.

iii) Efficient memory utilization. Here memory is not pre-allocated. Memory is allocated whenever it is required.

iv) Insertion or deletion is easy and efficient, may be done very fast, in a constant amount of times, independent of the size of the list.

v) The joining of two linked lists can be done by assigning pointer of the second linked list in the last node of the first linked list.

Splitting can be done by assigning a null address in the node from where we want to split one linked list into two parts.

Types of Linked list

There are different types of linked list. We can put linked lists into the following four types:

i) Singly linked list
ii) Circular Linked list
iii) Doubly Linked list
iv) Circular Doubly linked list

Singly linked list

An element in a linked list is specially termed as a node. In a singly linked list, each node consists of two fields:

i) DATA field that contains the actual information of the element.
ii) LINK field, contains the address of the next node in the list.

A "DATA" field to store whatever element type the list holds for the user, and a "LINK" field, which is a pointer, used to link one node to the next node. Each node is allocated in the heap with a call to malloc() function. The node memory becomes free when it is explicitly de-allocated with a call to free() function. The front of the list is a pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look like.

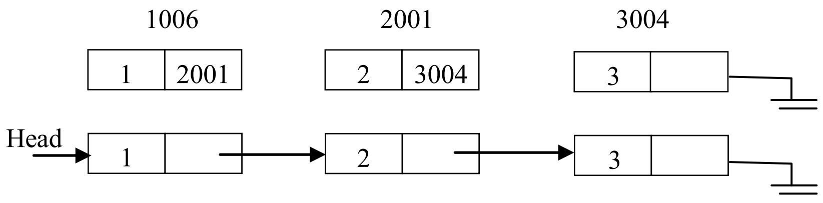


Figure 5.1: Linked list representation

Operations on Singly linked list

Operations supported by a singly linked list are as follows:

Table 5.2: Various Operation on Linked list

Operation	Description
Createlist	This operation creates a linked list.
Traverse	This operation traverse/visit all the elements of the linked list exactly once
Insertion	This operation inserts an element to the linked list
Deletion	This operation removes an element from the linked list
Searching	This operation performs linear searching for a key value in the linked list
Reverse	This operation performs the reverse of the linked list
Merging	This operation performs merging of two linked lists in a single linked list

Before going to the detail operation on singly linked list, we need two data types: Node and Node pointer.

A linked list is constructed by the nodes. These nodes are allocated in the heap. Each node contains a single data element and a pointer to the next node in the list.

```
struct node
{
   int DATA;
   struct node *LINK;
};
```

The Node structure of a linked list is shown here. Where DATA contains value and LINK holds the address of the next node.

DATA	LINK

Figure 5.2: Symbolic representation of a node

The LINK field of a node in a linked list points to the next node of list. LINK pointer is termed as self-referential pointer as it points to the address of a node of the same type.

Create a Singly Linked List

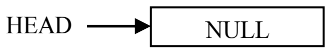
The following algorithm creates a node and appends it at the end of the existing list. _HEAD‘ is a pointer which holds the address of the HEADER of the linked list and _ITEM‘ is the value of the new node. NEW is a pointer which holds the address of the new node and _Temp‘ is a temporary pointer.

Algorithm to create a singly Linked List

```
Algorithm: CREATE (HEAD, ITEM)
1. [Create NEW node]
   a) Allocate memory for NEW node.
   b) IF NEW = NULL then Print: "Memory not Available" and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→LINK = NULL
2. [Whether List is empty, head is the content of HEADER]
   If HEAD = NULL then Set HEAD = NEW
3. Else
   a) Set Temp = HEAD
   b) While Temp→LINK ≠ NULL do
          Set Temp = Temp→LINK
      [End of while]
   c) Set Temp→LINK = NEW
   [End of IF]
4. Return
```

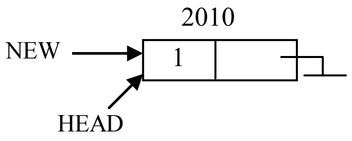
At first creates a NEW node from the heap using dynamic memory allocation and checks whether is NULL or not. If the NEW node is NULL, then memory is not available for creating the linked list. Otherwise, stores the value of ITEM to the DATA part of the NEW node and stores NULL to the LINK part of the NEW node.

The following diagrams explain the creation of a singly linked list.
At first HEAD is assigned with NULL value



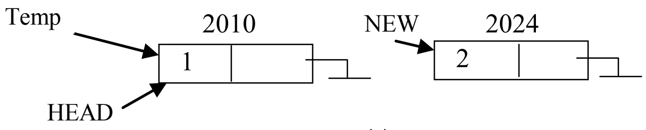
(a)

After that a new node is created and the address of this node is assigned to HEAD,

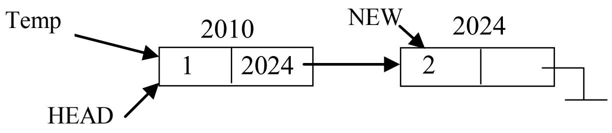


(b)

In the next step another node is created and linked to HEAD node.



(c)



(d)

Figure 5.3 (a-d): Create a Singly Linked List

Traversing / Display a Linked List

This algorithm traverses a linked list and prints the data part of each node of the linked list. The ‗HEAD' is a pointer which points to the starting node of the linked list and ‗Temp' is a temporary pointer to traverse the list.

Algorithm to traverse a Linked List

```
Algorithm: TRAVERSE(HEAD)
1. If HEAD = NULL then
      i) Print: "The linked list is empty"
      ii) Return
2. Temp = HEAD
3. Repeat while Temp ≠ NULL
      i) Print: Temp→DATA
      ii) Set Temp = Temp→LINK
   [End of Loop]
4. Return
```

Insertion in Singly Linked List

Insertion operation in a singly linked list can be done in different ways using position.

- Insertion at beginning.
- Insertion in the middle.
- Insertion at end.

Insertion of a node at first position of a singly linked list

In the following algorithm insertion of node at beginning position is described. The _HEAD' is a pointer which points to the starting node of the linked list. NEW points to the new node.

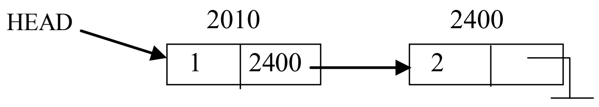
Algorithm to insert a node at the beginning

```
Algorithm:ADD_BEG (HEAD, ITEM)
1. [Create the new node]
   a) Allocate memory for NEW node.
   b) IF NEW = NULL then Print: "Memory not Available" and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→LINK = HEAD
2. [Make the HEADER to point to the NEW node]
   Set HEAD = NEW
3. Return
```

At first creates a NEW node from the heap using dynamic memory allocation and checks whether is NULL or not. If the NEW node is NULL, then memory is not available. Otherwise, stores the value of ITEM of the DATA part of the NEW node.

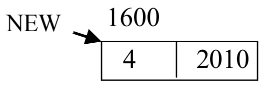
The following diagrams explain the insertion operation at the beginning of a singly linked list.

At first HEAD points to the first node of the list containing two nodes.



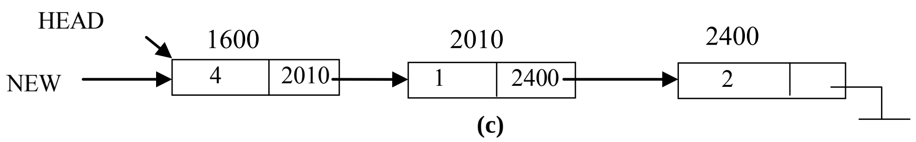
(a)

A new node is created and pointed by pointer NEW. LINK field of new node contains the address of head node.



(b)

Now, HEAD pointer points to the new node



(c)

Figure 5.4 (a-c): Node insertion in a single linked list at beginning

Insertion of a node before a specified node of a singly linked list

This algorithm creates a node and inserts it before the node pointer P. The HEAD is a pointer that points to the first node of the linked list and ITEM is the value of the new node. NEW is a pointer that holds the address of the new node. Temp and PTemp are two temporary pointers to traverse the list.

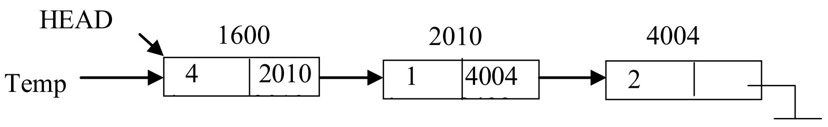
Algorithm to insert a node before a given node pointer

```
Algorithm: ADD_BEFORE (HEAD, ITEM, P)
1. Set Temp = HEAD   [to make temp to point to the first node]
2. Repeat step 3 while Temp ≠ P
3. a) Set PTemp = Temp
   b) Set Temp = Temp→LINK
   c) If Temp = NULL then
      i) Print: "Not Found"
      ii) Return
   [End of loop]
4. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: "Memory not Available" and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→LINK = Temp
   e) Set PTemp→LINK = NEW
5. Return
```

At the beginning traversing the linked list from HEAD to node pointer P to get the location of the previous node. Then, create a NEW node from the heap using dynamic memory allocation and checks whether is NULL or not. If the NEW node is NULL, then memory is not available. Otherwise, stores the value of ITEM of the DATA part of the NEW node.

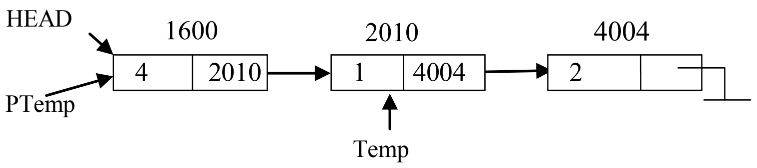
The following diagrams explain the insertion operation before a node pointer of a singly linked list.

At first, Temp is assigned to HEAD.



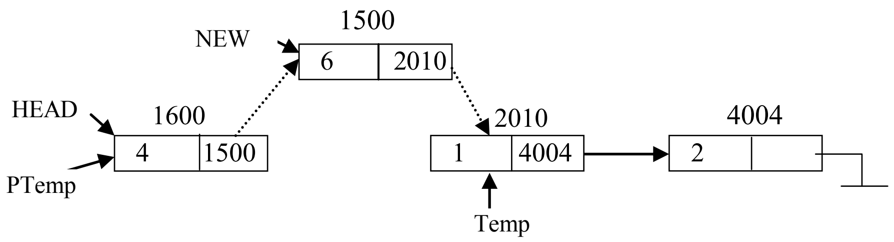
(a)

In the next step, Temp is moved to the next node and PTemp is assigned to the address of the previous node



(b)

After that a NEW node is created and linked with previous and next node.



(c)

Figure 5.5 (a-c): Node insertion before any position in a linked list

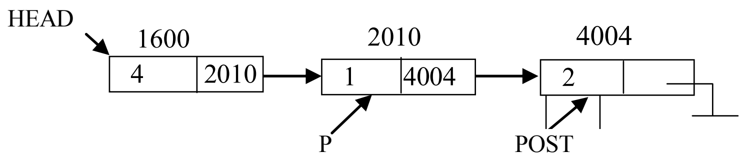
Insertion of a node after a node of a singly linked list

In the following algorithm, it is described how to insert a node after a specific node pointer P in a linked list. The HEAD is a pointer which points to the first node of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node. POST is a temporary pointer that points to the next node of P.

Algorithm to insert a node after a given node pointer

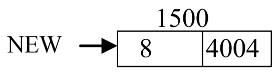
```
Algorithm: ADD_AFTER (HEAD, ITEM, P)
1. Set POST = P→LINK
2. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: "Memory not Available" and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→LINK = POST
   e) Set P→LINK = NEW
3. Return
```

At the beginning, set POST pointer by the link part of P pointer. Now, POST points to the to the next node of P. Then creates a NEW node and stores the value of ITEM of the DATA part of the NEW node. The following diagrams explain the insertion operation after a node pointer of a singly linked list. In the figure 5.6a, POST holds the address of the node after a specific node P.



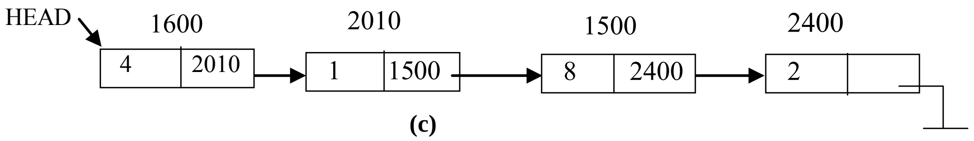
(a)

After that a NEW node is created, ITEM is placed in DATA field and POST is assigned to the LINK field which is shown in figure 5.6b.



(b)

At last, NEW node is assigned to LINK of node P



(c)

Figure 5.6 (a, b, c): Node insertion after any position in a linked list

Insertion of a node at the end of a singly linked list

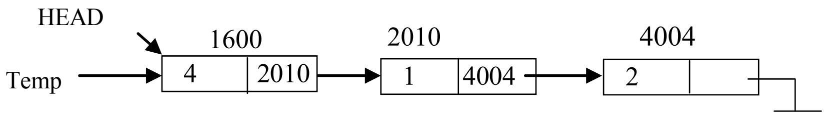
The following algorithm describes how the new node is inserted at the end of a singly linked list. The HEAD is a pointer which points to the first node of the linked list and NEW is a pointer which holds the address of the new node. ITEM is the value of the new node. Temp holds the address of header node.

Algorithm to insert a node at end

```
Algorithm: ADD_END (HEAD, ITEM)
1. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: "Memory not Available"  and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→LINK = NULL
2. Set Temp = HEAD   [to make Temp to point to the first node]
3. Repeat while Temp→LINK ≠ NULL
      Set Temp = Temp→LINK
   [End of loop]
4. Set Temp→LINK = NEW
5. Return
```

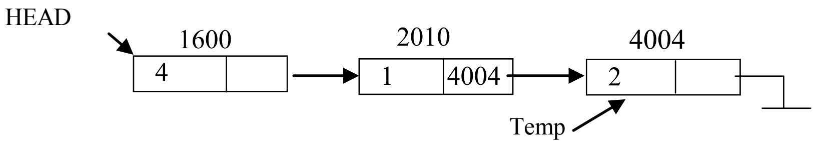
At the beginning creates a NEW node and stores the value of ITEM of the DATA part of the NEW node. Then, traversing the linked list from HEAD to the last node to get the location of the last node. Then The following diagrams explain the insertion operation at the end of a singly linked list.

At first Temp is assigned to the address of the HEAD node of the linked list



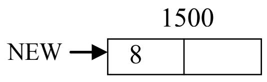
(a)

Now, Temp moves to the end of the linked list.



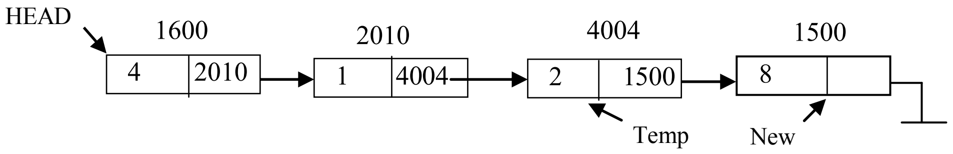
(b)

A NEW node is created ITEM is placed in DATA field and NULL is assigned to LINK field.



(c)

At last, NEW is assigned to the LINK field of the last node of the linked list.



(d)

Figure 5.7 (a, b, c, d): Node insertion at last position in a linked list

Deletion from a singly linked list

Deletion operation in a singly linked list can be done in different ways using position.

- Deletion from beginning.
- Deletion in the middle.
- Deletion from end.

Deleting a node from the beginning

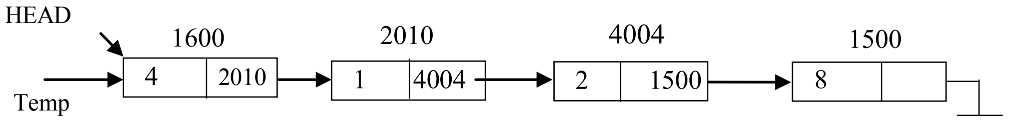
In the following algorithm deletion of head node of a linked list is described. Temp is a temporary pointer holds the address of a header node (HEAD). ITEM variable is used to store the value of the deleted node.

Algorithm to delete a node from the beginning

```
Algorithm: DELETE_BEG (HEAD, ITEM)
1. [Check for empty list]
   IF HEAD = NULL then Print: "The linked list is empty" and Return
2. Set Temp = HEAD      [To make Temp to point the first node]
3. Set HEAD = Temp→LINK
4. Set ITEM = Temp→DATA
5. Set Temp→LINK = NULL
6. Deallocate memory for Temp Node
7. Return
```

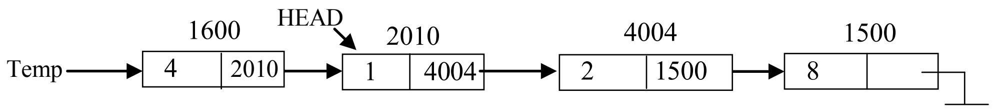
At the beginning checks whether the linked list is empty or not. Temp pointer points to the first node of the linked list, then HEAD moves to the next node and stores the DATA part of the Temp node to ITEM. The following diagrams explain the deletion operation from a singly linked list.

In the first step Temp is assigned to the address of the HEAD.



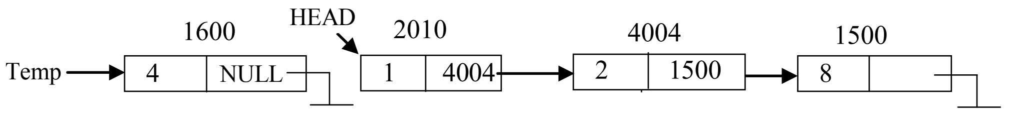
(a)

In the next step HEAD moves to the next node.



(b)

Finally, the first node of the linked list is deleted which is pointed by Temp.



(c)

Figure 5.8 (a, b, c): Node deletion from first position of a linked list

Deletion from a singly linked list from end position

Deletion of a node from the end of a singly linked list is described in the following algorithm. Temp is a temporary pointer points to HEAD node. Temp pointer is used to traverse the linked list to keep HEAD pointer in its position. PTemp is an another temporary pointer that holds the address of the previous node of the node to be deleted.

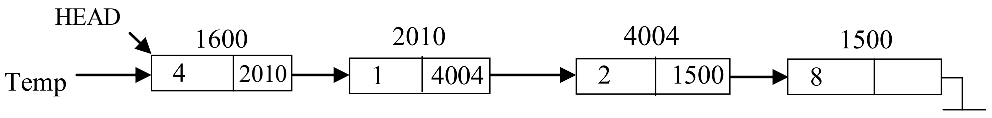
Algorithm to delete a node from the end

```
Algorithm: DELETE_END (HEAD, ITEM)
1. [Check for empty list]
   IF HEAD = NULL then Print: "The linked list is empty" and Return
2. Set Temp = HEAD
3. Repeat while Temp→LINK ≠ NULL
      a) PTemp = Temp
      b) Set Temp = Temp→LINK
   [End of loop]
4. Set ITEM = Temp→DATA
5. Set PTemp→LINK = NULL
6. Deallocate memory for Temp Node
7. Return
```

At the beginning checks whether the linked list is empty or not. Then, traversing the linked list from HEAD to the last node to get the location of the last node and the second last node. Stores the DATA part of the Temp node to ITEM.

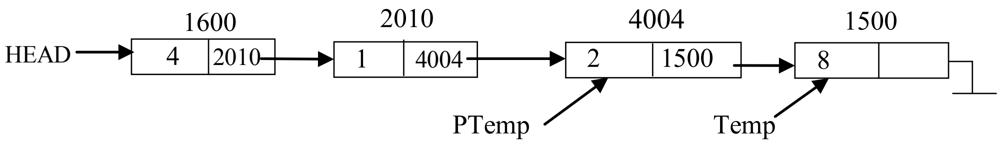
The following diagrams explain the deletion operation from a singly linked list.

At first Temp is assigned with HEAD.



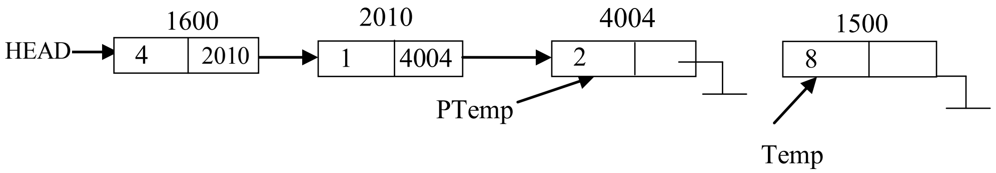
(a)

Temp is moved to the end of the list PTemp holds the address of the previous node.



(b)

At last, PTemp LINK field is assigned to NULL and Temp is deleted.



(c)

Figure 5.9 (a, b, c): Node deletion from end position of a linked list

Deletion from a singly linked list after any intermediate node

In the following algorithm the deletion of a node from a specific position is described. P holds the address of the previous node of Temp node which has to be deleted.

Algorithm to delete a node after a given node pointer

```
Algorithm: DELETE_AFTER (ITEM, P)
1. Set Temp = P→LINK
2. Set P→LINK = Temp→LINK
3. Set ITEM = Temp→DATA
4. Set Temp→LINK = NULL
5. Deallocate memory for Temp Node
6. Return
```

At the beginning stores the DATA part of the Temp node to ITEM. P points to the previous node of the Temp node that has to be deleted. LINK part of P node points to the next node of Temp node. LINK of Temp is set to NULL. Then, Temp is de-allocated.

Deletion from a singly linked list with a given ITEM

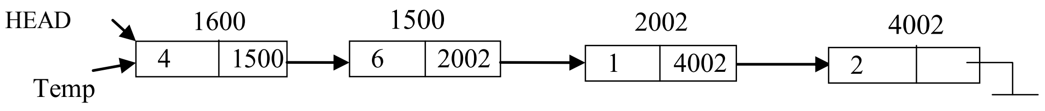
This algorithm finds and deletes a node whose value is taken as ITEM. Temp is a pointer which holds the address of HEADER of the linked list. Temp and old are two temporary pointers to traverse the list.

Algorithm to delete a node by value

```
Algorithm: DELETE_ITEM (HEAD, ITEM)
1. [Make temp to point the first node]
   Set Temp = HEAD
2. [Check for empty list]
```

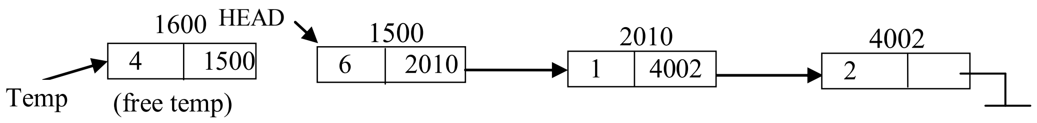
```
    IF HEAD = NULL then
       a) Print: "The linked list is empty"
       b) Return
3. Repeat Step 4 to 5 until Temp is NULL
4. If Temp→DATA = ITEM then
       a) If Temp = HEAD then  // node to be deleted is the first node
             Set HEAD = Temp→LINK
          Else
             Set PTemp→LINK = Temp→LINK
       b) Deallocate memory for Temp Node// de allocate node
       c) Return
 5. Else
       Set PTemp = Temp and Temp = Temp→LINK
    [End of loop]
 6. Print: "Element not found"
 7. Return
```

At first, Temp points to the first node of the list which is shown in the following diagram.



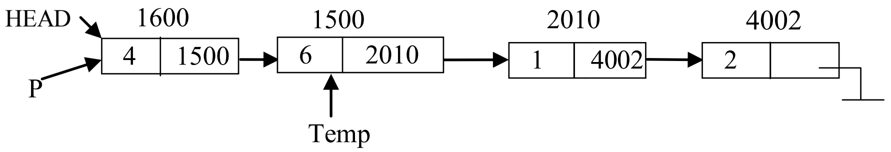
(a)

Then it is checked whether the list is empty or not, if not empty then the first node is deleted as shown in the figure 5.10b.



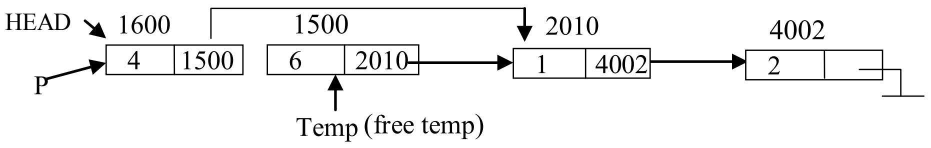
(b)

To delete any other node Temp is assigned to that node and P holds the address of the previous node.

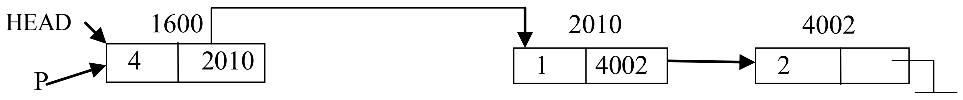


(c)

Node of the node to be deleted. Then, LINK of P is assigned with LINK of Temp. After that Temp is deleted.



(d)



(e)

Figure 5.10 (a-e): Node deletion at any position in a linked list

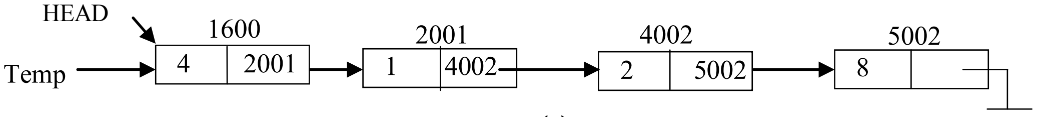
Searching a singly linked list

This algorithm finds the location of a node in a linked list whose value is ITEM. Temp is a pointer which points to the starting node of the linked list. ‗Temp' is a temporary pointer to traverse the list. ‗LOC' is the variable to store the location of the search item.

Algorithm to search Linked List by given value

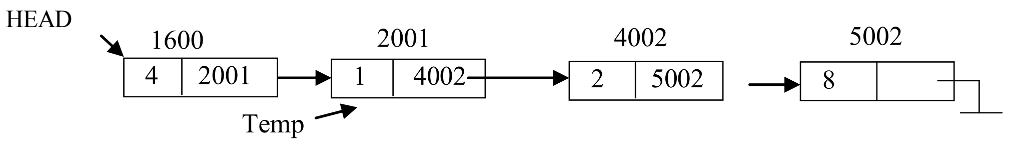
```
Algorithm: SEARCH(HEAD, ITEM, LOC)
1. Set Temp = HEAD, LOC=NULL
2. If Temp = NULL then
      i) "The linked list is empty"
      ii) Return
3. Repeat step 4 while Temp ≠ NULL
4.    If Temp→DATA = ITEM then
            i) Print: "Element found"
            ii) Set LOC = Temp
            iii) Return
      Else
            Temp = Temp→LINK
      [End of If]
   [End of Loop]
5. Print: "Element not found"
6. Return
```

At first Temp is assigned to the address of HEAD



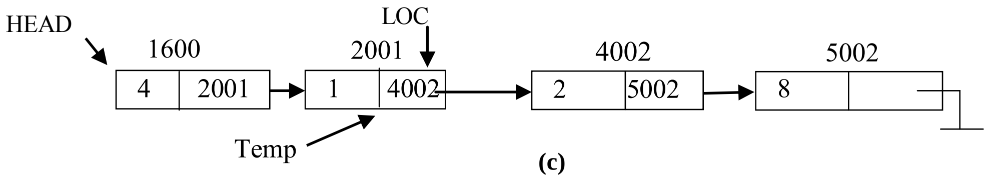
(a)

After that Temp→DATA is compared with ITEM=1 and Temp moves accordingly



(b)

When the ITEM is found LOC points to the index of that node in the list.



(c)

Figure 5.11(a, b, c): Searching of an ITEM in linked list

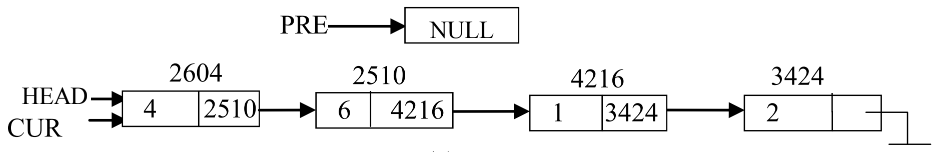
Reverse of a Linked List

Reversing of linked list means that the last node becomes the first node and first becomes the last. HEAD is a pointer which holds the address of HEADER of the linked list. PRE, POST and CUR are temporary pointers.

Algorithm to reverse Linked List

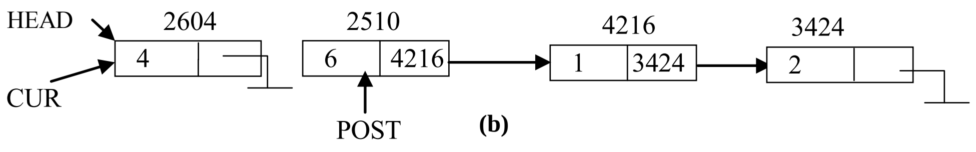
```
Algorithm: REVERSE (HEAD)
1. Set PRE = NULL and CUR = HEAD
2. Repeat step 3 to 6 while CUR ≠ NULL
3.    Set POST =CUR→LINK
4.    Set CUR→LINK=PRE
5.    Set PRE = CUR
6.    Set CUR = POST
      [End of Loop]
7. HEAD = PRE
8. Return
```

In the first step NULL is assigned to PRE pointer and CUR pointer is assigned to the HEAD node of the linked list.



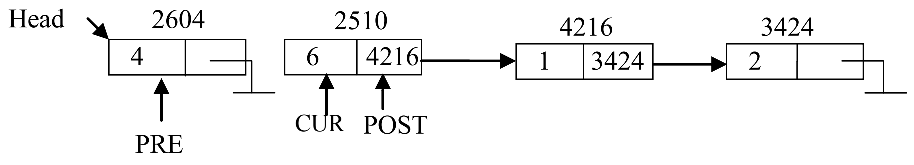
(a)

After first iteration POST points to the next node of the current node



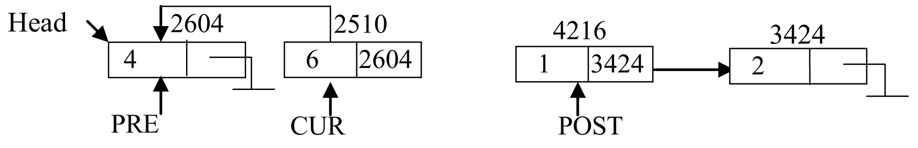
(b)

In the next step CUR is assigned to PRE and POST is assigned to CUR. LINK field of the PRE node is assigned to NULL.



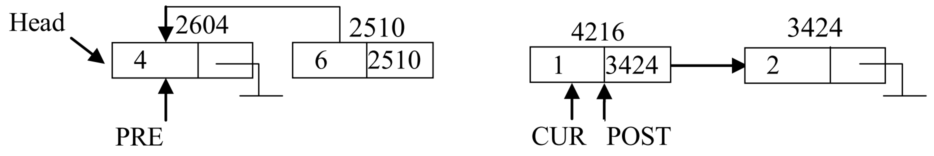
(c)

After a second iteration LINK field of CUR node is linked with PRE and POST moves to the next node.



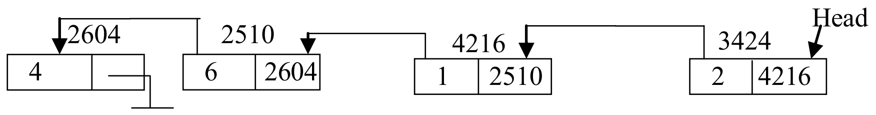
(d)

After that CUR is assigned with POST.



(e)

After 4th iteration the single linked list look like the following figure 5.11f



(f)

Figure 5.12 (a-f): Reverse of a linked list

Reverse Display of a Linked List

In this algorithm the linked list is displayed in reverse order. HEAD points to the first node of the list. P is used to forward the loop to the end of the list. Then R is set to the last node of the list. After that R is used for displaying the list in reverse order.

Algorithm to reverse display the Linked List

```
Algorithm: REVERSE_DISPLAY (HEAD)
1. Set R = NULL and Q = HEAD
2. Repeat step 3 to 7 while Q≠R
3.    Set P = Q
4.    Repeat step 5 while P→ LINK ≠ R
5.      Set P = P→LINK
```

```
      [End of Loop]
6.    Print: P→INFO
7.    Set R =P
   [End of Loop]
8. Return
```

Convert Array to Singly Linked List

We can convert a singly linked list using an existing array. The following algorithm creates a node, stores an array element and appends it at the beginning of the existing list.

This algorithm creates a node and inserts it at the beginning of the list. ‗HEAD‘ is a pointer which holds the address of the HEADER of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node.

Algorithm to convert Linked-List from an existing Array

```
Algorithm: CONVERT (A, N, HEAD)
1. Set HEAD = NULL
2. Repeat steps 3 to 8 while N ≠ -1
3.     Allocated memory for NEW node
4.     IF NEW = NULL then Print: "Memory not Available" and Return
5.     Set NEW→DATA = A[N-1]
6.     Set NEW→LINK = HEAD
7.     Set HEAD = NEW
8.     Set N = N - 1
   [End of loop]
9. Return
```

Representation of polynomials using linked list

In this representation, the polynomial can also be represented by linking list and a node in the linked list represents a term in the polynomial.

At first, we have to store exponent/degree and coefficient of the polynomial f (x). For that, we have to define a structure as follows:

```
typedef struct poly
{
      int exp;
      int coef;
      struct poly *LINK;
 } term;
poly *head;
```

We will always store terms of the polynomial in descending order of degree.

$4x^6 + 2x^4 + 3x + 1$ would be represented by linked list like:

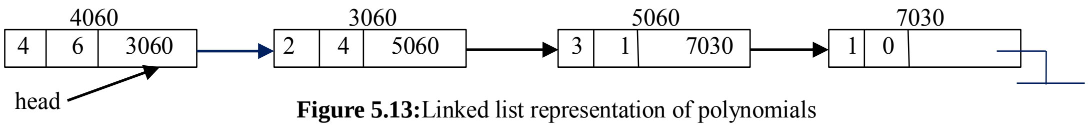


Figure 5.13: Linked list representation of polynomials

The addition of two Polynomial using Linked List

In the following algorithm addition of two polynomials are described. Two linked lists consisting of two polynomials are headed by HEAD1 and HEAD2 respectively. Insert function is used to insert a node in a new linked list which is headed by R.

Algorithm to addition of two polynomials

```
Algorithm: POLYADD(HEAD1, HEAD2)
1. Set P=HEAD1, Q=HEAD2, R=NULL
2. Repeat while P≠NULL and Q ≠NULL
      i) If P→EXP > Q→EXP then
            Call INSERT(R, P→EXP, P→COEF)
            Set P=P→LINK
      ii) Else If P→EXP < Q→EXP then
            Call INSERT(R, Q→EXP,Q→COEF)
            Set Q=Q→LINK
      iii) Else If P→COEF + Q→COEF ≠ 0
            Call INSERT(R, P→EXP, P→COEF + Q→COEF)
            Set P=P→LINK
            Set Q=Q→LINK
           [End of If]
    [End of Loop]
3. Repeat while P≠NULL
      i) Call INSERT(R, P→EXP, P→COEF)
      ii) Set P=P→LINK
    [End of Loop]
4. Repeat while Q≠NULL
      i) Call INSERT(R, Q→EXP, Q→COEF)
      ii) Set Q=Q→LINK
    [End of Loop]
5. Return
Function: INSERT(R, EXP, COEF)
1. Allocate memory for NEW node
2. If NEW=NULL then
      i) Print: Out of Memory
      ii) Return
3. Set NEW→EXP=EXP, NEW→COEF=COEF, NEW→LINK=NULL
4. If R=NULL then
      HEAD3=R=NEW
   Else
      Set R→LINK=NEW and R=NEW
    [End of If]
5. Return
```

Complexity of a singly linked list

The cost to insert or delete an element into a known location in the singly linked list is O(1). Whereas for a linear array the insert or delete operation cost is O(n). This is because, in the case of linked list insert or delete operation do not involve any data movement just by performing pointer exchange among nodes the insertion or deletion take place. In case of array both the operations involve data replacement by n places.

Table 5.3: Comparison of Array and Linked list

Operations	Array	Linked List
Insert/delete at the beginning	O(n)	O(1)
Insert/delete at the end	O(1)	O(1)
Insert/delete in the middle	O(n)	O(1)
Indexing	O(1)	O(n)
Wastage space	O(1)	O(n)

Circular Linked List

A circular linked list is the variation of singly linked list in which the last node points to the first node of the list. The circular linked lists have neither a beginning nor an end.

In a circular linked list, there are two methods to know if a node is the first node or not.

i) Either an external pointer, list, points the first node or

ii) A header node is placed as the first node of the circular list.

The header node can be separated from the others by either having a sentinel value as the info part or having a dedicated flag variable to specify if the node is a header node or not.

A linked list can be implemented either in two ways:

i) A single pointer is used to point to the header node of the linked list and the last node of the list points to the header node of the list.

ii) Two different pointers can be used to point to the first and last node of the circular linked list respectively.

In a linear linked list once we traverse the list, then it is difficult to return back to the first node of the list. A circular linked list provides a solution to this problem. In a circular linked list the last node points to the first node of the linked list so, there is no difficulty to return back to the first node when the list is traversed.

A circular linked list can easily be formed from a linear linked list where the last node of linear linked list points to the first node of the linked list.

The circular linked list can be represented by three nodes as follows

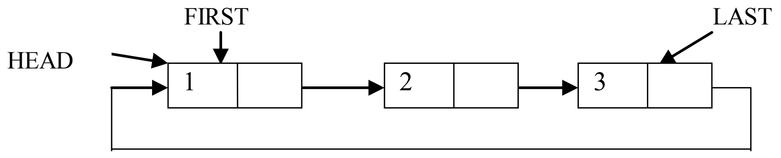


Figure 5.14: Circular linked list

Here, HEAD points to the first node of the list and last node points to the first node forming a circular list.

Operations on Circular linked list

Operations supported by a singly linked list are as follows:

Table 5.4: Operation of Circular Linked list

Operation	Description
Createlist	This operation creates a linked list.
Traverse	This operation traverse/visit all the elements of the linked list exactly once.
Insertion	This operation inserts an element to the linked list.
Deletion	This operation removes an element from the linked list.
Searching	This operation performs linear searching for a key value in the linked list.
Reverse	This operation performs the reverse of the linked list.
Merging	This operation performs merging of two linked lists in a single linked list.

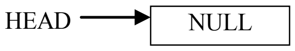
Create a Circular linked list

This algorithm creates a node and appends it after the last node (which points to the first node) of the existing list. HEAD is a pointer which holds the address of the FIRST pointer of the linked list and _DATA' is the value of the new node. _NEW' is a pointer which holds the address of the new node and _temp' is a temporary pointer.

Algorithm to create a Circular Linked List

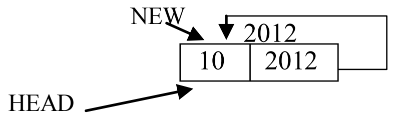
```
Algorithm: CIRCULAR_ADD (HEAD, ITEM)
1. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: "Memory not Available" and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→LINK = NEW
2. [Check whether List is empty or not]
   If HEAD = NULL then
      Set HEAD = NEW
3. Else
   a) Set Temp = HEAD
   b) Repeat while Temp→LINK ≠ HEAD do
         Set Temp = Temp→LINK
      [End of loop]
c) Set Temp→LINK = NEW
4. [Make a new node to point to the first node]
   Set NEW→LINK=HEAD
5. Return
```

Primarily HEAD is assigned with NULL when there is no node in a circular linked list indicates first node of the linked list. Initially it is pointing to NULL.



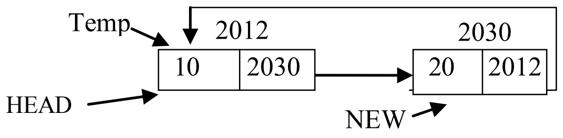
(a)

After that a NEW node is created the address is assigned with HEAD. The LINK field is assigned to the address of the HEAD node.



(b)

In the next step another node is created and linked with first node. The LINK field of the NEW node is linked with first node.



(c)

Figure 5.15 (a, b, c): Circular linked list

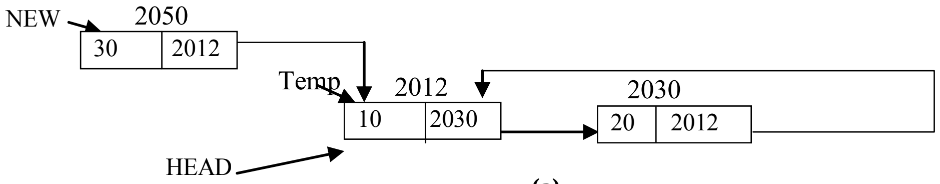
Add a New Node at The Beginning of the Circular Linked list

This algorithm creates a node and inserts it at the beginning of the list. HEAD is a pointer, which holds the address of the first node of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node and ‗Temp' and ‗ln' are temporary pointers.

Algorithm to insert a node at the beginning

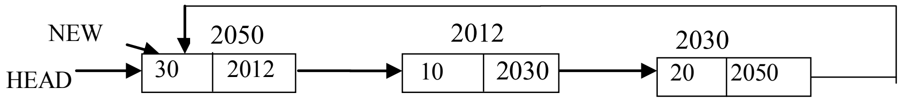
```
Algorithm: CIRADD_BEG (HEAD, ITEM)
1. [Create the new node]
a) Allocate memory for NEW node.
b) IF NEW = NULL then Print: "Memory not Available"  and Return
c) Set NEW→DATA= ITEM
d) Set NEW→LINK= HEAD
2. a) Set Temp=HEAD
   b) Repeat while Temp→LINK ≠ HEAD do
         Set Temp = Temp→LINK
      [End of loop]
3.  [Make FIRST to point to the new node]
       Set HEAD = NEW          // to make new node the first node
       Set Temp→LINK=HEAD
4. Return
```

A NEW node is created and linked with the HEAD node of Circular linked list.



(a)

At last, the address of the NEW node is assigned to the LINK field of the last node of the linked list.



(b)

Figure 5.16(a, b): Insertion in a Circular linked list before a node

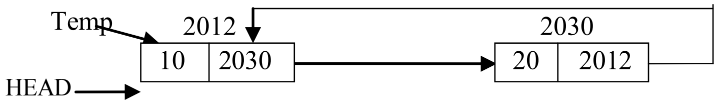
Add the New Node after a specific location of the Circular Linked list

This algorithm creates a node and inserts it after the node at location loc. HEAD is a pointer which points to the first node and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node. _Temp‘ and _PTemp‘ are two temporary pointers to traverse the list.

Algorithm to insert a node after given node pointer

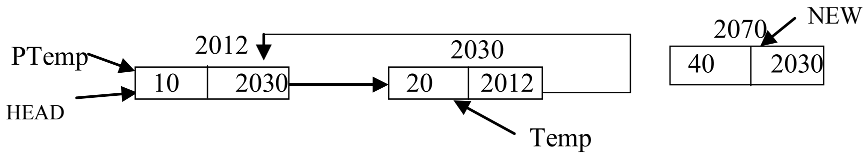
```
Algorithm: CIRADD_AFTER (HEAD, ITEM, LOC)
1. [Create the new node]
    a) Allocate memory for new node.
    b) IF NEW = NULL then Print: "Memory not Available" and Return
    b) Set NEW→DATA = ITEM
    c) Set NEW→LINK= Temp
2. [Make Temp point to the first node]
   Set Temp = HEAD
3. Repeat step 3 for i=1 to LOC
3.  a) Set  PTemp = Temp and   Temp = Temp→LINK
    b) If Temp = HEAD then
       Print: "There are less than loc+1 elements" and Return
4. Set PTemp→LINK = NEW
5. Return
```

In the first step Temp points to the first node of the circular linked list.



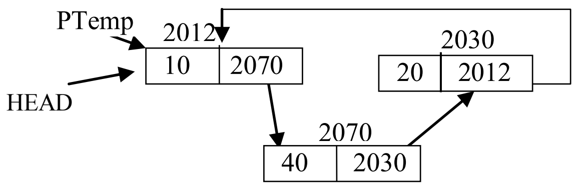
(a)

In the next step Temp is moved until loc is found. Temp points to the node at loc position. PTemp holds the address of the previous node. A NEW node is created whose LINK field is assigned with the address of the HEAD node.



(b)

At last the NEW node is inserted at the position before loc. For this example loc is considered to be 2.That means the NEW node is inserted at 2nd position of the circular linked list.



(c)

Figure 5.17 (a, b, c): Insertion in a Circular linked list

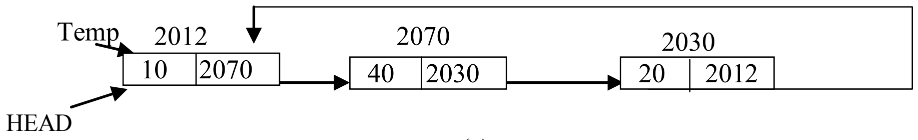
Delete a Node from The Beginning of the Circular Linked list

This algorithm deletes a node from the beginning of a circular linked list. HEAD is a pointer which holds the address of the FIRST pointer of the linked list. Temp is a temporary pointer. ITEM is the variable to store the value of the deleted item.

Algorithm to delete a node from the beginning

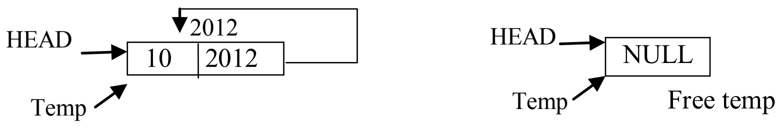
```
ALGORITHM: CIRDEL (HEAD)
1. [Make Temp to point the first node]
   Set Temp = HEAD
2. [Check for empty list]
   If HEAD = NULL then
      Print: "The linked list is empty"
3. Else
         [Linked list contains single element]
      a) If HEAD = Temp→LINK then
            Set HEAD = NULL
         Else
            P=HEAD
            Repeat while P→LINK ≠ HEAD do
              Set P = P→LINK
            [End of loop]
      b)HEAD=HEAD→LINK
      c)P→LINK=HEAD
4. Set ITEM = Temp→DATA
5. Deallocate memory for Temp
6. Return
```

Firstly, Temp is assigned to the HEAD node of the circular linked list.



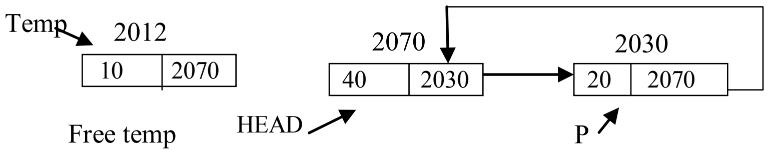
(a)

In the succeeding step it is checked whether the list is empty or have only one node, if the list has only one node then HEAD points to NULL and the node is deleted. This phenomena are depicted in the following figure 5.16b



(b)

If the list contains more than one node, then Temp is assigned to HEAD and HEAD moves to the next node. Then Temp is deleted. The LINK field of the last node contains the address of the new first node.



(c)

Figure 5.18 (a, b, c): Deletion from a Circular linked list

Traverse a circular Linked list

This algorithm traverses a circular linked list and prints the data part of each node of the linked list. _f is a pointer which points to the starting node of the linked list . _Temp‘ is a temporary pointer to traverse the list.

Algorithm to display a Circular Linked List

```
ALGORITHM: DISPLAY (HEAD)
1. [Make Temp to point to the first node]
   Set Temp = HEAD
2. [Check for empty list]
   If HEAD = NULL then
      Print: "The linked list is empty" and Return
3. Print: Temp→DATA
      Set Temp = Temp→LINK
4. Repeat While Temp! = HEAD
      Print: Temp→DATA
      Set Temp = Temp→LINK
5. Return
```

Complexity of a circular linked list

The cost to add or delete an element from a known location in the circular linked list is O(1).

Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS is given a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example, can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.

- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Josephus Problem

The Josephus problem consists of a group of soldiers surrounded by an overwhelming enemy force. There is only a single horse available for escape. To determine which soldier is to escape, they form a circle & a number n is picked from a hat. A name is also picked from the hat. They begin to count clockwise around the circle, beginning with the soldier, whose name is picked. One soldier is removed from the circle on which count reaches n & the count starts again with next soldier, removing another soldier each time the count reaches n & the last remaining soldier is the one to take the horse.

Our problem is to find the order in which soldier will escape. The input to our program is the number n and a list of names. The names are according to the clockwise; ordering of circle beginning with the soldier from whom the count is to start. The program prints the names in the order in which they are eliminated & the name of the soldier who escapes. The data structure used is, a circular linked list in which each node represents one soldier as it is possible to reach any node from any other node by moving around the circle.

The structure of the node in the circular linked list consist of name and link part

```
struct node
{
 char name[30];
 struct node *link;
}
```

Algorithm to solve Josephus problem

```
ALGORITHM: JOSEPHUS(HEAD)
1. Read N and Name
2. Set count= number of soldiers
        Set Temp=HEAD
3. Repeat until Name=Temp→name
        Temp=Temp→LINK
   [End of loop]
4. Repeat until Count=1
       a) Set C=1
       b) Repeat until C= N
          i)   Set Prev=Temp
          ii)  Set Temp=Temp→LINK
          iii) Set C=C+1
          [End of loop]
       c) Prev→LINK=Temp->LINK
             Print Temp→DATA "soldier remove"
       d) Set Count=Count-1
   [End of loop]
5. Print Prev→name "soldier escape"
6. Return
```

Doubly Linked List

A double linked list is defined as a collection of nodes in which each node has three parts: data, llink, rlink. Data contains the data value for the node, llink contains the address of node before it and rlink contains the address of node after it.

A generic doubly linked list node can be designed as:

```
struct node
{
   int data;
   struct node* rlink;
   struct node* llink;
};
```

Each node contains three parts:

i) An information field contains *data.*
ii) A pointer field *rlink,* which contains the location of the next node.
iii) A pointer field *llink,* which contains the location of the preceding node.

The following figure, explain how the doubly linked list looks like:

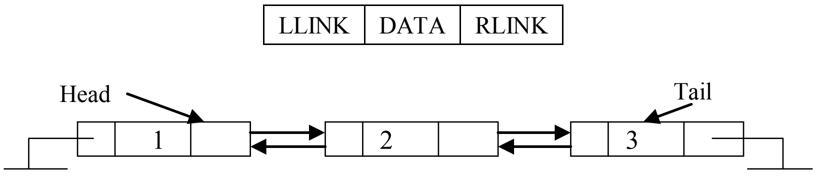


Figure 5.19: Structure of a doubly linked list

In the figure Head and Tail pointer points to the header node and tailor node of the list respectively.

Algorithm for Creation of Doubly Linked list

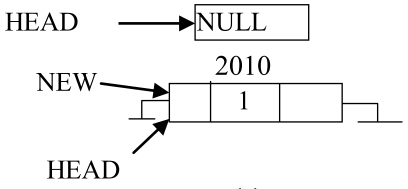
This algorithm creates a node and appends it at the end of existing list. Temp is a pointer which holds the address of the HEADER of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node and _Temp‘ is a temporary pointer.

Algorithm to create Doubly Linked List

```
Algorithm: DLL_CREATE(HEAD, ITEM)
1. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: "Memory not Available" and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→LLINK = NULL
   e) Set NEW→RLINK = NULL
2. [Whether List is empty, head is the content of HEADER]
   If HEAD = NULL then Set HEAD = NEW
3. Else
   a) Set Temp = HEAD
   b) While Temp→RLINK ≠ NULL do
```

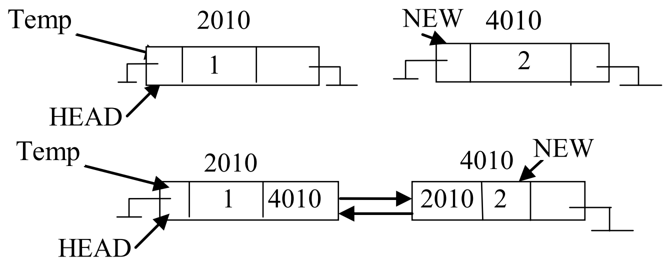
```
          Set Temp = Temp→RLINK
       [End of while]
    c) Set Temp→RLINK = NEW
    d) Set NEW→LLINK = Temp
   [End of IF]
4. Return
```

Initially HEAD is assigned to NULL. Then a NEW node is created and HEAD points to that node.



(a)

After creation of HEAD node, next node is created and linked with first node. HEAD node is pointed by Temp. rlink of HEAD node holds the address of NEW node and llink of NEW node holds the address of HEAD node.



(b)

Figure 5.20 (a, b): Creation of a doubly linked list

Addition at the Beginning of the Doubly Linked list

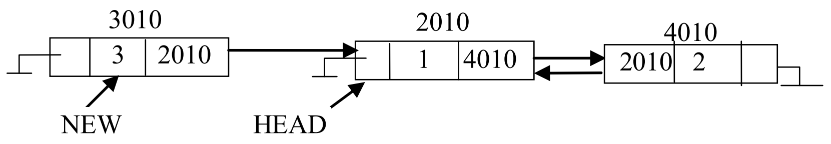
This algorithm creates a node and inserts it at the beginning of the list. ‗head‘ is a pointer which holds the address of the HEAD of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node.

Algorithm to insert a node at the beginning

```
Algorithm: DLL_ADD_BEG (HEAD, ITEM)
1. [Create the new node]
   a) Allocate memory for NEW node.
   b) IF NEW = NULL then Print: "Memory not Available"  and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→RLINK = HEAD
   e) Set NEW→LLINK = NULL
2. Set HEAD→LLINK =NEW
3. [Make the HEADER to point to the NEW node]
```

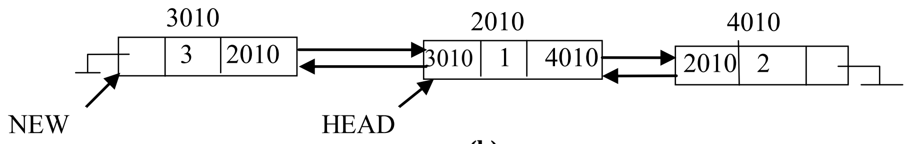
```
   Set HEAD = NEW
4. Return
```

Initially a NEW node is created where rlink of NEW node contains the address of the HEAD node.



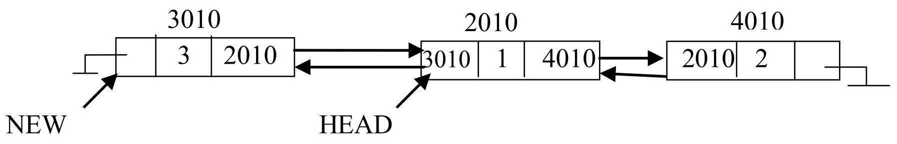
(a)

After that llink of HEAD node is assigned to the address of NEW node.



(b)

At last the HEAD pointer moves to the NEW node which is depicted in the following figure 5.19c



(c)

Figure 5.21 (a, b, c): Addition of node at the beginning of doubly linked list

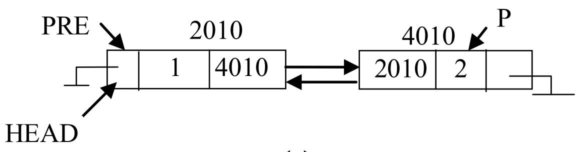
Addition at the Beginning of the Doubly Linked list

Following algorithm describes the addition of a node before any specific node in a doubly linked list. PRE holds the address of the previous node of P before which NEW node to be inserted.

Algorithm to insert a node before a node pointer

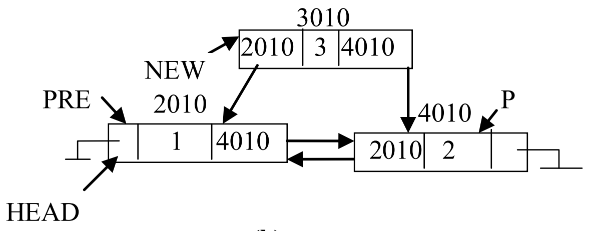
```
Algorithm: DLL_ADD_BEFORE (HEAD, ITEM, P)
1. Set PRE = P→LLINK
2. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: Overflow and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→LLINK = PRE
   e) Set NEW→RLINK = P
3. Set PRE→RLINK = NEW
4. Set P→LLINK = NEW
5. Return
```

Initially PRE holds the address of the previous node where NEW node to be inserted and P holds the address of the next node.



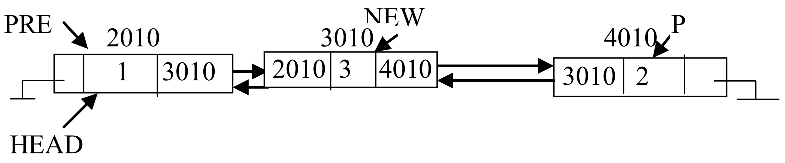
(a)

A NEW node is created whose rlink holds the address of the P node and llink holds the address of the PRE node.



(b)

At last, rlink of PRE is linked with NEW node and llink of P is linked with NEW node.



(c)

Figure 5.22 (a, b, c): Insertion in a doubly linked list

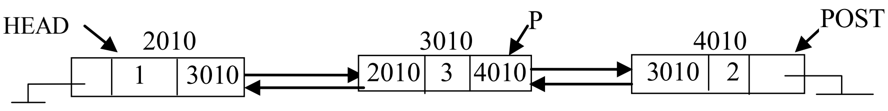
Addition after any position of the Doubly Linked list

This algorithm creates a node and inserts it after the node at location P. HEAD is a pointer which points to the first node, i.e. it holds the content of HEADER of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node. POST holds the address of the next node where a new node to be inserted.

Algorithm to insert a node after a given node pointer

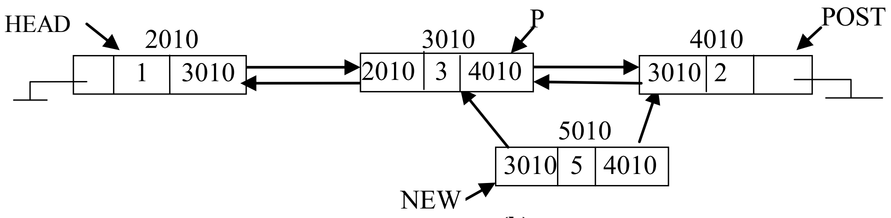
```
Algorithm: DLL_ADD_AFTER (HEAD, ITEM, P)
1. Set POST = P→RLINK
2. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: "Memory not Available" and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→RLINK = POST
   e) Set NEW→LLINK = P
3. Set P→RLINK = NEW
4. Set POST→LLINK = NEW
5. Return
```

At first POST is assigned with the RLINK of the node P after which a new node to be inserted.



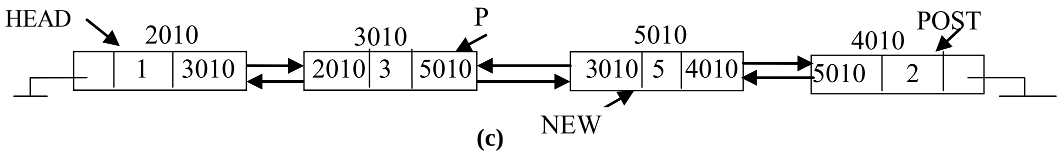
(a)

After that a NEW is created where ITEM is placed in DATA field and RLINK holds the address of the POST where LLINK contains the address of the P node.



(b)

At last, RLINK of node P is linked with NEW node and LLINK link of POST node is linked with NEW node



(c)

Figure 5.23 (a, b, c): Addition of a node after any position in a doubly linked list

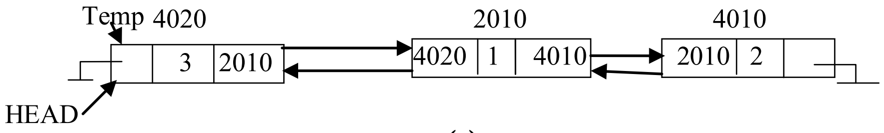
Addition at last position of the Doubly Linked list

In the following algorithm addition of node at the end of the doubly linked list is described. Here Temp points to the HEAD node. Then Temp moves to the end of the linked list. At last NEW node is linked with RLINK of Temp node.

Algorithm to insert a node at the end

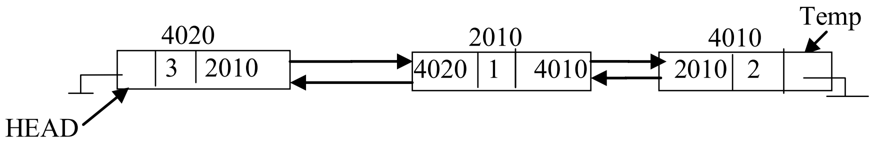
```
Algorithm: ADD_END (HEAD, ITEM)
1. [Make temp to point to the first node]
   Set Temp = HEAD
2. Repeat while Temp→RLINK ≠ NULL
      Set Temp = Temp→RLINK
   [End of loop]
3. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: Overflow and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→RLINK = NULL
   e) Set NEW→LLINK = Temp
4. Set Temp→RLINK = NEW
5. Return
```

Firstly, Temp is assigned to the HEAD node.



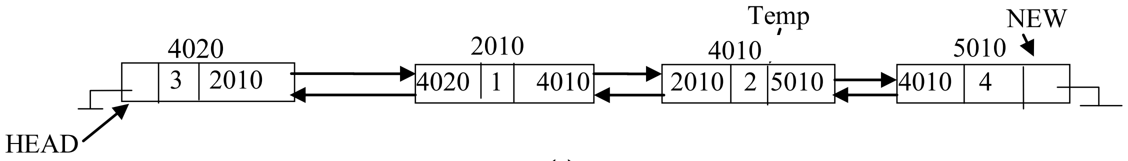
(a)

Now, move Temp to the end of the doubly linked list.



(b)

At last a NEW node is created and linked with Temp



(c)

Figure 5.24 (a, b, c): Insertion of a new node at the end of a doubly linked list

Traverse a Doubly Linked list in Forward Direction

This algorithm traverses a linked list and prints the DATA part of each node of the linked list in the forward direction (from the first node to last). HEAD is a pointer which points to the starting node of the linked list. Temp is a temporary pointer to traverse the list.

Algorithm to traverse in forward direction

```
Algorithm: DLL_FTRAVERSE (HEAD)
1. [Check for empty list]
   If HEAD = NULL then
      i) Print: "The linked list is empty"
      ii) Return
2. Set Temp = HEAD
3. Repeat while Temp ≠ NULL
   i) Print: Temp→DATA
   ii) Set Temp = Temp→RLINK
   [End Of Loop]
4. Return
```

Traverse a Doubly Linked list in Backward Direction

This algorithm traverses a linked list and prints the data part of each node of the linked list in backward direction (from the last node to first). HEAD is a pointer which points to the starting node of the linked list. Temp is a temporary pointer to traverse the list. At first Temp moves in forward direction using RLINK and then Temp is used for backward traverse of the doubly linked list using LLINK.

Algorithm to traverse in backward direction

```
Algorithm: DLL_BTRAVERSE (HEAD)
1. [Check for empty list]
   If HEAD = NULL then
      i) Print: “The linked list is empty”
      ii) Return
2. [Make temp to point to the first node]
   Set Temp = HEAD
3. Repeat while Temp→RLINK ≠ NULL
      Set Temp = Temp →RLINK
   [End of loop]
4. Repeat while Temp ≠ NULL
      i) Print: Temp→DATA
      ii) Set Temp = Temp →LLINK
   [End of loop]
5. Return
```

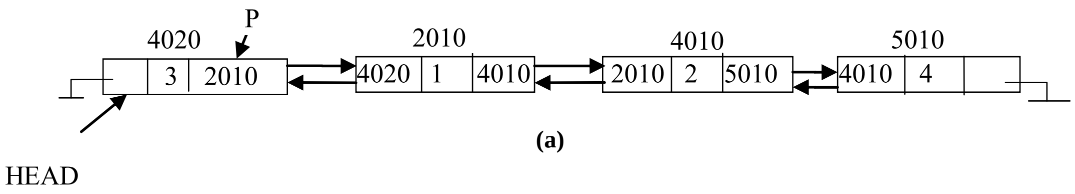
Search an Item in a doubly linked list

This algorithm describes searching of an ITEM in a doubly linked list. P is a pointer that assigned the Head node address of the doubly linked list. P moves from left to right until the ITEM is found or the list end.

Algorithm to search an item in doubly linked list

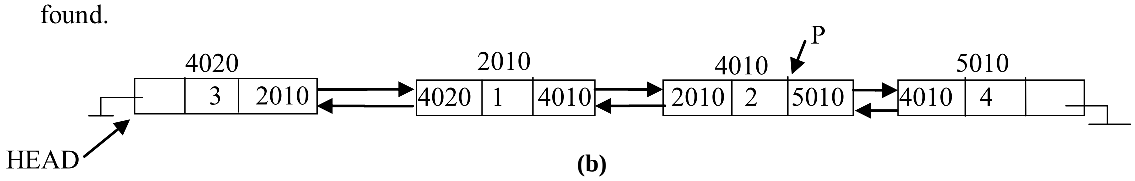
```
Algorithm: DLL_SEARCH (HEAD, ITEM, LOC)
1. Set P = HEAD, LOC=NULL
2. [Check for empty list]
   If P = NULL then
      i) “The linked list is empty”
      ii) Return
3. Repeat step 4 while P ≠ NULL
4.    If P→DATA = ITEM then
            i) Print: “Element found”
            ii) Set LOC = P
            iii) Return
      Else
            P = P→RLINK
      [End of If]
   [End of Loop]
5. Print: “Element not found”
6. Return
```

In the first step P is assigned with the address of the first node of the doubly linked list.



(a)

Let us consider the ITEM to be searched is 2 which is at 3rd position. P moves until ITEM is found.



(b)

Figure 5.25 (a, b): Searching of an ITEM in doubly linked list

Delete a Node from the Beginning of a doubly linked list

In the following algorithm the process of deleting a node from the beginning of a doubly linked list is described. HEAD denotes the first node of the list. In the figure 5.26b this phenomena are depicted.

Algorithm to delete a node from the beginning

```
Algorithm: DLL_DELETE_BEG (HEAD)
1. [Check for empty list]
   IF HEAD = NULL then
      a) Print: "The linked list is empty"
      b) Return
2. [Make temp to point the first node]
   Set Temp = HEAD
3. Set HEAD = Temp→RLINK
4. Set HEAD→LLINK = NULL
5. Set ITEM = Temp→DATA
6. Set Temp→RLINK = NULL
7. Set Temp→LLINK = NULL
8. Deallocate memory for Temp Node
9. Return
```

Delete a Node from the End of a doubly linked list

The following algorithm describes the procedure of removing a node from the end of a doubly linked list. Here, HEAD denotes the first node of the list. Temp is used to move the pointer from the first node to last node. PTemp is used to hold the address of the preceding node of the node to be deleted. After reaching at last position the Temp is de-allocated. Figure 5.24c depicts this phenomenon pictorially.

Algorithm to delete a node from the end

```
Algorithm: DLL_DELETE_END (HEAD, ITEM)
1. [Check for empty list]
```

```
   IF HEAD = NULL then
      a) Print: "The linked list is empty"
      b) Return
2. Repeat step while Temp→RLINK ≠ NULL
      a) Set PTemp = Temp
      b) Set Temp = Temp→RLINK
   [End of loop]
3. Set ITEM = Temp→DATA
4. Set Temp→LLINK = NULL
5. Set PTemp→RLINK = NULL
6. Deallocate memory for Temp Node
7. Return
```

Delete a Node after a specified position (P) of a doubly linked list

Here, the algorithm specifies the procedure of deleting a node which is after a node P. Temp holds the address of the next node of the node P and POST holds the address of the next node of the node Temp which is to be deleted. Figure 5.24d describes this process.

Algorithm to delete a node after a given node pointer

```
Algorithm: DLL_DELETE_AFTER (HEAD, ITEM, P)
1. Set Temp = P→RLINK
2. Set POST = Temp→RLINK
3. Set P→RLINK = POST
4. IF POST ≠ NULL then Set POST→LLINK = P
5. Set ITEM = Temp→DATA
6. Set Temp→RLINK = NULL
7. Set Temp→LLINK = NULL
8. Deallocate memory for Temp Node
9. Return
```

Delete a Node before a specified position (P) of a doubly linked list

Here, the algorithm specifies the procedure of deleting a node which is before a node P. Temp holds the address of the preceding node of the node P and PRE holds the address of the preceding node of the node Temp which is to be deleted. Figure 5.24e describes this process.

Algorithm to delete a node before given node pointer

```
Algorithm: DLL_DELETE_BEFORE (HEAD, ITEM, P)
1. Set Temp = P→LLINK
2. Set PRE= Temp→LLINK
3. IF PRE ≠ NULL then Set PRE→RLINK = P
4. Set P→LLINK = PRE
5. Set ITEM = Temp→DATA
6. Set Temp→RLINK = N ULL
7. Set Temp→LLINK = NULL
```

```
8. Deallocate memory for Temp Node
9. Return
```

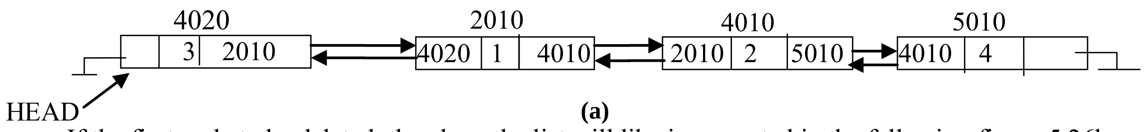
Delete a Node from the doubly linked list having a specific DATA value

This algorithm finds and deletes a node whose value is ‗NO'. Temp is a pointer which holds the address of HEADER of the linked list .

Algorithm to delete a node by value

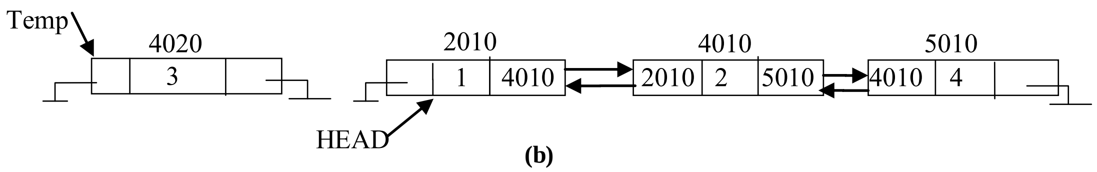
```
ALGORITHM: DDEL (HEAD, NO)
1. Set Temp = HEAD //to make temp to point the first node
2. [Check for empty list]
   If HEAD = NULL then
      Print: "The linked list is empty" and Return
3. Repeat Step 4 to 5 until temp is NULL
  a) If Temp→DATA = NO then
     i) If TEMP = HEAD then// node to be deleted is the first node
            Set HEAD = Temp→RLINK
            Set Temp→RLINK→LLINK = NULL
        Else if Temp→RLINK = NULL   //Check for last node
            Set P=Temp→LLINK
            Set POST= Temp→RLINK
            Set P→RLINK=NULL
        Else
            Set POST→LLINK = Temp→LLINK
            Set P→RLINK=Temp→RLINK
     ii) Deallocate memory for Temp Node        // de allocate node
     iii)Return
    Else
        Set Temp = Temp→RLINK
4. Print: "Element not found"
5. Return
```

Following figure 5.26a denotes a doubly linked list where HEAD points to the first node of the list.



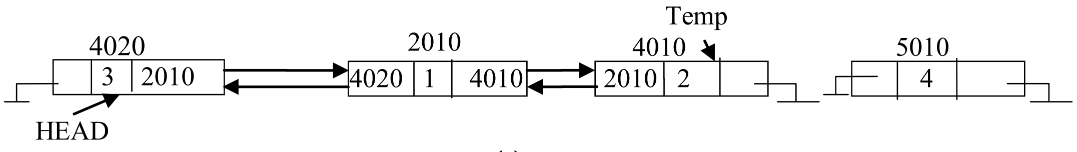
(a)

If the first node to be deleted, then how the list will like is presented in the following figure 5.26b.



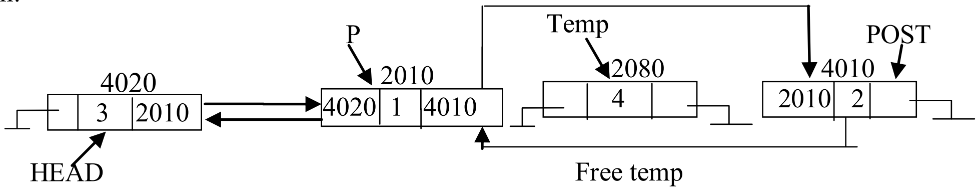
(b)

In the following figure 5.26c deletion of a node from last position of doubly linked list is described.



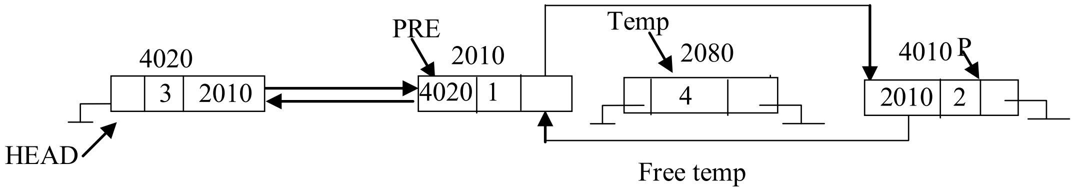
(c)

In the following figure 5.26d, the deletion of a node Temp which is after a specific node P is shown.



(d)

In the succeeding figure it is pictorially presents that how a node to be deleted which is before a specific node of a doubly linked list.



(e)

Figure 5.26 (a-e): Node deletion at any position in a doubly linked list

Time Complexity of Doubly Linked list

The time complexity for insertion and deletion from a doubly linked list is O (1), as there is no movement of nodes just by exchanging some pointers the a new node can be inserted or an existing node can be deleted. Whereas for traversal, the complexity is O(n).

Circular Doubly Linked List:

The advantages of both double linked list and circular linked list are incorporated into another list structure that is called circular doubly linked list and it is known to be the best of its kind.

The following is a schematic representation of a circular doubly linked list.

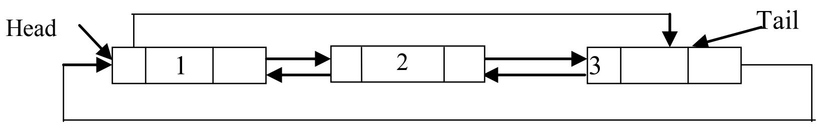


Figure 5.27: Circular Doubly linked list

As per above shown illustrations, following are the important points to be considered.

- Last Link's rlink points to the first node of the list in both cases of singly as well as doubly linked list.
- First Link's llink points to the last node of the list in case of doubly linked list.

Disadvantages of Linked List

- Linked list consumes more memory space for storing addresses of the next node.
- Searching will be slow as only linear searching will be possible.
- Direct access to any node not possible, therefore, if a node required to access it is required to traverse from the first node onwards until the desired node is found.

Therefore, arrays are suitable when a large number of searching operations are necessary, but insertion/deletion is very small in number. Whereas linked-list are suitable, when there will be frequent insertion/deletion, but very few search operations are required.

Summary

- A linked list is a linear ordered collection of finite homogeneous data elements called node.
- The linked list overcomes the demerits of array in terms of memory allocation.
- Linked list can be classified into 4 types single linked list, circular linked list, doubly linked list and circular doubly linked list.
- Polynomial addition can be done using linked list.
- Addition, Deletion and Insertion of a node in a linked list is faster than array.

Exercises

1. What are the advantages and disadvantages of linked list data structure over an array?
2. Write an algorithm to insert a data X after a specific data item Y in the linked list.
3. Write a function to delete the n^{th} node of a singly linked list. The error conditions are to be handled properly.
4. Write an algorithm to delete any node from the singly liked list where the key value of the node is known.
5. Write an algorithm to delete a node from a doubly linked list.
6. Write an algorithm to delete all nodes having greater than a given value, from a given singly linked list.
7. Write a function to reverse the direction of all links of a singly linked list.
8. Write an algorithm to add two polynomials using linked list.
9. How the polynomial $4x^3-10x^2+3$ can be represented using linked list?
10. Write a C program to create a doubly linked list in ascending order of information.
11. Write a C program to join two sorted linked list so that the third list is in sorted order.
12. Write a C program to show the information of a doubly linked list in reverse order.
13. Write a C program to INSERT and DELETE node in a circularly doubly linked list. Display the elements of the list.
14. Write C program to insert an element in a sorted doubly linked list, so that the list remain in sorted order after insertion.
15. Write a program in C language to find the predecessor of a node in linked list.
16. Choose the correct alternatives for the following:

i) The situation when in a linked list START = NULL is

a) Underflow b) Overflow c) Saturated d) None

ii) A linked list follows

a) random access mechanism b) sequential access mechanism
c) no access mechanism d) none of these.

iii) A linear collection of data elements where the linear node is given by means of pointer is called

a) Linked list b) Node list c) Primitive list d) None of these.

iv) In linked list representation a node contains at least

a) node address field, data field b) node number field, data field
c) next address field, information field d) none of these.

v) Inserting a new node after a given node in a doubly linked list requires

a) four pointer exchanges b) two pointer exchanges
c) one pointer exchanges d) *np*pointer exchange.

vi) The nth node in a singly linked list can be accessed via

a) The head node b) The tail node c) (n-1)th node d) None

vii) Linear order in linked list is provided through

a) Index number b) The implied position of the node c) Pointer d) None

viiii) Null pointer is used to tell

a) End of a linked list b) Empty pointer field of a structure
c) The linked list is empty d) All the above

ix) In linked list the successive elements

a) Must occupy contiguous space in memory
b) Need not occupy contiguous space in memory
c) Must not occupy contiguous space in memory
d) None of the above

x) Searching in a linked list requires linked list be created

a) In sorted order b) In any order c) Without underflow condition d) None of the above

xi) Deletion of a node in linked list involves keeping track of the address of the node

a) Which immediately follows the node that is to be deleted
b) Which immediately precedes the node that is to be deleted
c) The node to be deleted
d) None of the above

xii) Header in a linked list is a special node at the

a) End of the linked list b) At middle of a linked list
c) Beginning of the linked list d) None of the above

xiii) Header linked list in which last node points to the header node is called

a) Grounded header list b) Circular header list c) General header list d) None of the above

xiv) Representing polynomial using linked list requires each node having

a) Two fields b) Three fields c) More than three fields d) None of the above

xv) Inserting a node after a given node in a doubly linked list requires

a) One pointer change b) Two pointer change c) Four pointer change d) None of the above

*****