Linked Lists(Unit 1.2)

1.DEFINITION

A *linked list* is an ordered collection of finite, homogeneous data elements called *nodes* where the linear order is maintained by means of links or pointers. The linked list can be classified into three major groups: single linked list, circular linked list, and double linked list.

2. SINGLE LINKED LIST

In a single linked list each node contains only one link which points to the subsequent node in the list. Figure shows a linked list with six nodes. Here, Nl, N2, ... , N6 are the constituent nodes in the list. HEADER is an empty node (having data content NULL) and only used to store a pointer to the first node Nl. Thus, if one knows the address of the HEADER node from the link field of this node, the next node can be traced, and so on. This means that starting from the first node one can reach to the last node whose link field does not contain any address but has a null value.

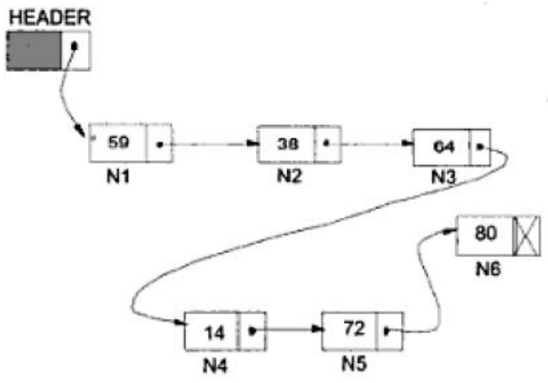


Figure 3.2 A single linked list with six nodes.

Representation of a Linked List in Memory

There are two ways to represent a linked list in memory:

Static representation using array and Dynamic representation using free pool of storage

Static representation: In static representation of a single linked list, two arrays are maintained: one array for data and the other for links. The static representation of the linked list in above Figure is shown in below Figure.

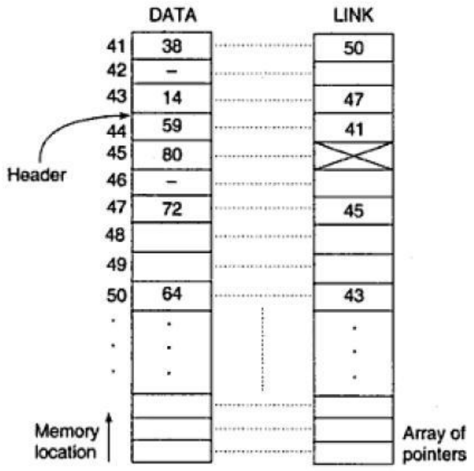


Figure 3.3 Static representation using arrays of the single linked list of Figure 3.20.

Two parallel arrays of equal size are allocated which should be sufficient to store the entire linked list. Nevertheless this contradicts the idea of the linked list (that is non-contagious location of elements). But in some programming languages, for example, ALGOL, FORTRAN, BASIC, etc. such a representation is the only representation to manage a linked list.

Dynamic representation

The efficient way of representing a linked list is using the free pool of storage. In this method, there is a *memory bank* (which "is nothing but a collection of free memory spaces) and a *memory manager* (a program, in fact). During the creation of a linked list, whenever a node is required the request is placed to the memory manager; the memory manager will then search the memory bank for the block requested and, if found, grants the desired block to the caller. Again, there is also another program called the *garbage collector;* it plays whenever a node is no more in use; it returns the unused node to the memory bank. It may be noted that memory bank is basically a list of memory' spaces which is available to a programmer. Such a memory management is known as *dynamic* memory management. The dynamic representation of linked list uses the dynamic memory management policy.

The mechanism of dynamic representation of single linked list is illustrated in Figures (a) and (b). A list of available memory spaces is there whose pointer is stored in AVAIL. For a request of a node, the list AVAIL is searched for the block of right size. If AVAIL is null or if the block of desired size is not found, the memory manager will return a message accordingly. Suppose the block is found and let it be XY. Then the memory manager will return the pointer of XY to the caller in a temporary buffer, say NEW. The newly availed node XY then can be inserted at any position in the linked list by changing the pointers of the concerned nodes. In Figure (a), the node XY is inserted at the end and change of pointers is shown by the dotted arrows. Figure (b) explains the mechanism of how a node can be returned from a linked list to the memory bank.

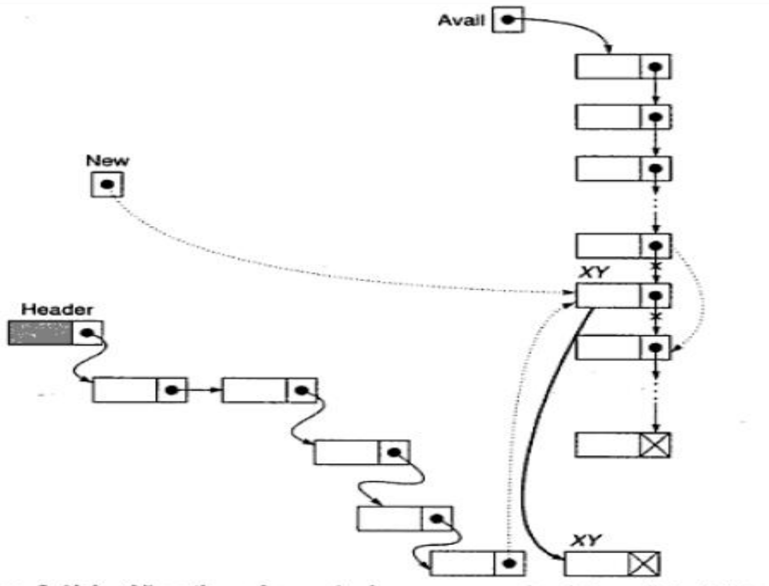


Figure 3.4(a) Allocation of a node from memory bank to a linked list.

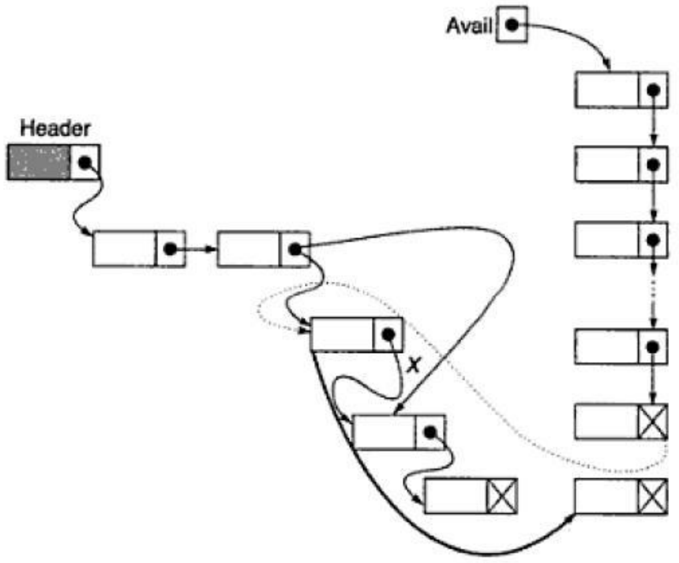


Figure 3.4(b) Returning a node from a linked list to memory bank.

The pointers which are required to be manipulated while returning a node are shown with dotted arrows. Note that such allocations or deallocations are carried out by changing the pointers only.

Operations on a Single Linked List

The operations possible on a single linked list are listed below: *Traversing* the list, *Inserting* a node into the list, *Deleting* a node from the list, *Copying* the list to make a duplicate of it, *Merging* the linked list with another one to make a larger list, *Searching* for an element in the list.

suppose X is a pointer to a node. The values in the DATA field and LINK field will be denoted by XDDATA and XDLINK, respectively, We will write NULL to imply nil value in the DATA and LINK fields.

Traversing a single linked list

In traversing a single linked list, we visit every node in the list starting from the first node to the last node. The following is the algorithm *Traverse_SL* for the same.

Algorithm Traverse_SL

Input: HEADER is the pointer to the header node.

Output: According to the *Process( )*

Data structures: A single linked list :-vhose address of the starting node is known from the *HEADER*.

```
Steps:
 1. ptr = HEADER.LINK              //ptr is to store the pointer to a current node
 2. While (ptr ≠ NULL) do          //Continue till the last node
     1. PROCESS(ptr)               //Perform PROCESS( ) on the current node
     2. ptr = ptr.LINK             //Move to the next node
 3. EndWhile
 4. Stop
```

Note: A simple operation of PROCESS( ) may be thought as to print the data content of a node.

Inserting a node into a single linked list

There are various positions where a node can be inserted:

(i) Inserting at the front (as a first element)

(ii) Inserting at the end (as a last element)

(iii) Inserting at any other position.

Let us assume a procedure *GetNode(NODE)* to get a pointer of a memory block which suits the type NODE. The procedure may be defined as follows:

```
Procedure GETNODE(NODE)
1. If (AVAIL = NULL)              //AVAIL is the pointer to the pool of free storage
    1. Return (NULL)
    2. Print "Insufficient memory: Unable to allocate memory"
2. Else                           //Sufficient memory is available
    1. ptr = AVAIL
    2. While (SIZEOF(ptr) ≠ SIZEOF(NODE)) and (ptr ≠ NULL)
                                  //Till the desired block is not found or search reaches
                                  //at the end of the pool
        1. ptr1 = ptr             //To keep the track of the previous block
        2. ptr = ptr.LINK         //Move to the next block
    3. EndWhile
    4. If (SIZEOF(ptr) = SIZEOF(NODE)) //Memory block of right size is found
        1. ptr1.LINK = ptr.LINK   //Update the AVAIL List
        2. Return(ptr)
    5. Else
        1. Print "The memory block is too large to fit"
        2. Return(NULL)
    6. EndIf
3. EndIf
4. Stop
```

Inserting a node at the front of a single linked list

The algorithm *lnsertliront Sl:* is used to insert a node at the front of a single linked list.

```
Algorithm INSERT_SL_FRONT(HEADER, X)
    Input: HEADER is the pointer to the header node and X is the data of the node to be
inserted.
    Output: A single linked list with newly inserted node in the front of the list.
    Data structures: A single linked list whose address of the starting node is known from
HEADER.

    Steps:
     1. new = GETNODE(NODE)             //Get a memory block of type NODE and store
                                        //its pointer in new
     2. If (new = NULL) then            //Memory manager returns NULL on searching
                                        //the memory bank
        1. Print "Memory underflow: No insertion"
        2. Exit                         //Quit the program
     3. Else                            //Memory is available and get a node from
                                        //memory bank
        1. new.LINK = HEADER.LINK       //Change of pointer 1 as shown in Figure 3.5(a)
        2. new.DATA = X                 //Copy the data X to newly availed node
        3. HEADER.LINK = new            //Change of pointer 2 as shown in Figure 3.5(a)
     4. EndIf
     5. Stop
```

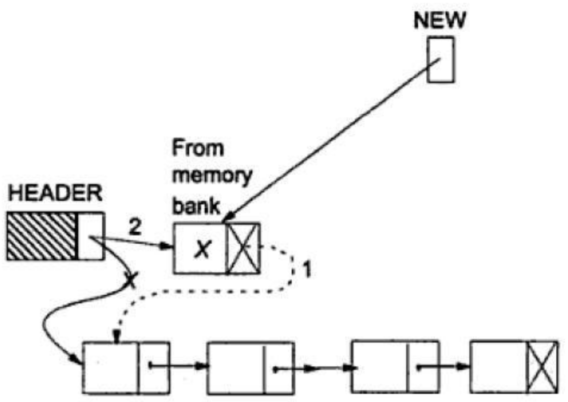


Fig. 3.5(a) Insertion of a node at the front of a single linked list.

Inserting a node at the end of a single linked list

The algorithm *InsertEnd_SL* is used to insert a node at the end of a single linked list

```
Algorithm INSERT_SL_END (HEADER, X)
    Input: HEADER is the pointer to the header node and X is the data of the node to be
inserted.
    Output: A single linked list with newly inserted node having data X at end.
    Data structures: A single linked list whose address of the starting node is known from
HEADER.

    Steps:
    1. new = GETNODE(NODE)             //Get a memory block of type NODE and store
                                       //its pointer in new
    2. If (new = NULL) then            //Unable to allocate memory for a node
       1. Print "Memory is insufficient: Insertion is not possible"
       2. Exit                         //Quit the program
    3. Else                            //Move to the end of the given list and then insert
       1. ptr = HEADER                 //Start from the HEADER node
       2. While (ptr.LINK ≠ NULL) do   //Move to the end
          1. ptr = ptr.LINK            //Change pointer to the next node
       3. EndWhile
       4. ptr.LINK = new               //Change the link field of last node: Pointer 1 as
                                       //in Figure 3.5(b)
       5. new.DATA = X                 //Copy the content X into new node
    4. EndIf
    5. Stop
```

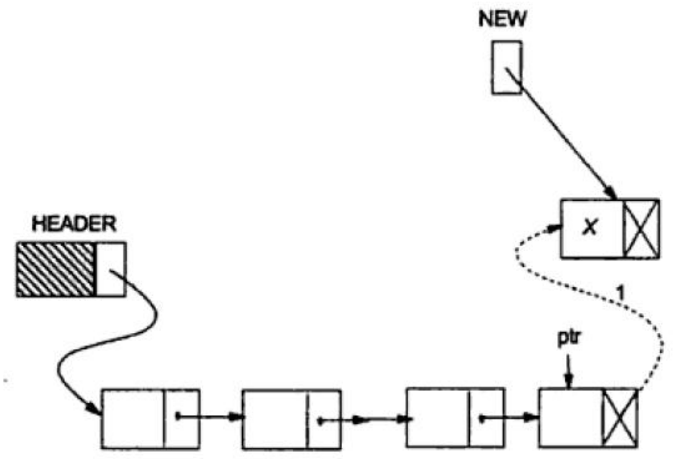


Fig. 3.5(b) Insertion at the end of a single linked list.

Inserting a node into a single linked list at any position in the list

The algorithm *InsertAny _SL* is used to insert a node into a single linked list at any position in the list.

```
Algorithm INSERT_SL_ANY(HEADER, X, KEY)
    Input:  HEADER is the pointer to the header node, X is the data of the node to be inserted,
and KEY being the data of the key node after which the node has to be inserted.
    Output:  A single linked list enriched with newly inserted node having data X after the node
with data KEY.
    Data structures:  A single linked list whose address of the starting node is known from
HEADER.

    Steps:
    1. new = GETNODE(NODE)                 //Get a memory block of type NODE and store
                                           //its pointer in new
    2. If (new = NULL) then                //Unable to allocate memory for a node
        1. Print "Memory is insufficient: Insertion is not possible"
        2. Exit                            //Quit the program
    3. Else
        1. ptr = HEADER                    //Start from the HEADER node
        2. While (ptr.DATA ≠ KEY) and (ptr.LINK ≠ NULL) do
                                           //Move to the node having data as KEY or at
                                           //the end if KEY is not in the list
            1. ptr = ptr.LINK
        3. EndWhile
        4. If (ptr.LINK = NULL) then       //Search fails to find the KEY
            1. Print "KEY is not available in the list"
            2. Exit
        5. Else
            1. new.LINK = ptr.LINK         //Change the pointer 1 as shown in Figure 3.5(c)
            2. new.DATA = X                //Copy the content into the new node
            3. ptr.LINK = new              //Change the pointer 2 as shown in Figure 3.5(c)
        6. EndIf
    4. EndIf
    5. Stop
```

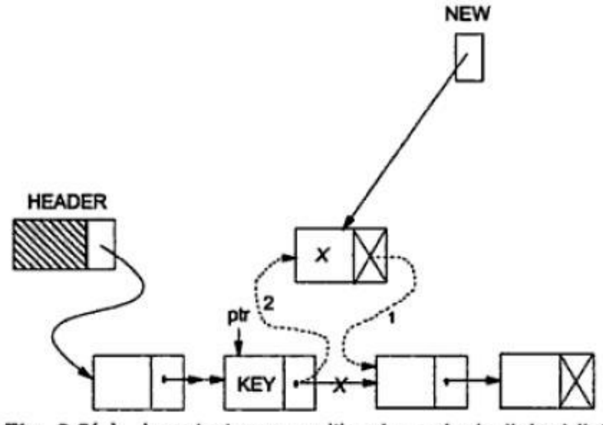


Fig. 3.5(c) Insert at any position in a single linked list.

Deleting a *node from* a *single linked list*
Like insertions, there are also three cases of deletions:
(i) Deleting from the front of the list
(ii) Deleting from the end of the list
(iii) Deleting from any position in the list
Assume a procedur namely *ReturnNode(ptr)* which returns a node having pointer *ptr* to the free pool of storage. The procedure *ReturnNode(ptr)* is defined as fallows.

```
Procedure RETURNNODE(PTR)          //PTR is the pointer of a node to be returned
1. ptr1 = AVAIL                    //Start from the beginning of the free pool
2. While (ptr1.LINK ≠ NULL) do
   1. ptr1 = ptr1.LINK
3. EndWhile
4. ptr1.LINK = PTR                 //Insert the node at the end
5. PTR.LINK = NULL                 //Node inserted is the last node
6. Stop
```

Note: The procedure RETURNNODE( ) inserts the free node at the end of the pool of free storage whose header address is AVAIL. Alternatively, we can insert the free node at the front of the AVAIL list which is left as an exercise.

Deleting the node at the front of a *single linked list*
The algorithm *Deletelirontt Sl:* is used to delete the node at the front of a single linked list. Such a deletion operation is explained in Figure 3.6(a).

Algorithm DELETE_SL_FRONT(HEADER)
Input: HEADER is the pointer to the header node.
Output: A single linked list eliminating the node at the front.
Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

```
1. ptr = HEADER.LINK               //Pointer to the first node
2. If (ptr = NULL)                 //If the list is empty
   1. Print "The list is empty: No deletion"
   2. Exit                         //Quit the program
3. Else                            //The list is not empty
   1. ptr1 = ptr.LINK              //ptr1 is the pointer to the second node, if any
   2. HEADER.LINK = ptr1           //Next node becomes the first node as in Figure 3.6(a)
   3. RETURNNODE(ptr)              //Deleted node is freed to the memory bank for
                                   //future use
4. EndIf
5. Stop
```

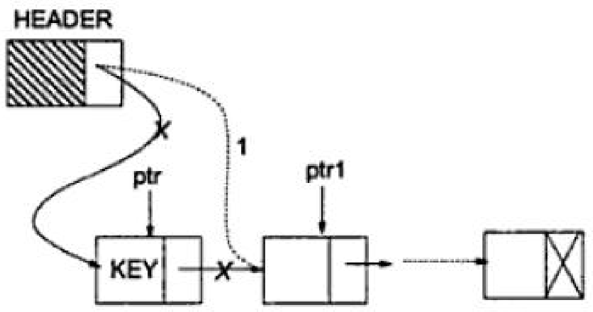


Fig. 3.6(a) Deletion of a node from a single linked list at the front.

Deleting the node at the end of a *single linked list*
The algorithm *DeleteEnd_SL* is used to delete the node at the end of a single linked list. This is shown in Figure.

Algorithm DELETE_SL_END(HEADER)
Input: HEADER is the pointer to the header node.
Output: A single linked list eliminating the node at the end.
Data structures: A single linked list whose address of the starting node is known from HEADER.

```
Steps:
 1. ptr = HEADER                      //Move from the header node
 2. If (ptr.LINK = NULL) then
    1. Print "The list is empty: No deletion possible"
    2. Exit                           //Quit the program
 3. Else
    1. While (ptr.LINK ≠ NULL) do     //Go to the last node
        1. ptr1 = ptr                 //To store the previous pointer
        2. ptr = ptr.LINK             //Move to the next
    2. EndWhile
    3. ptr1.LINK = NULL               //Last but one node become the last node as in
                                      //Figure 3.6(b)
    4. RETURNNODE(ptr)                //Deleted node is returned to the memory bank
                                      //for future use
 4. EndIf
 5. Stop
```

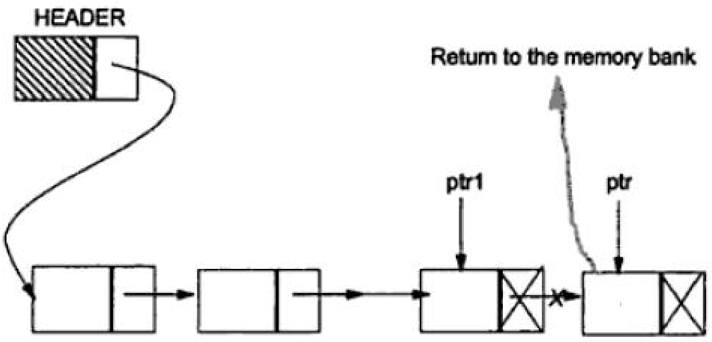


Fig. 3.6(b) Deletion of a node from a single linked list at the end.

<u>DeletIng the node from any position</u> <u>of a</u> *<u>single linked list</u>*
The algorithm *DeleteAny _SL* is used to delete a node from any position in a single linked list. This is illustrated in Figure.

Algorithm DELETE_SL_ANY
Input: HEADER is the pointer to the header node, KEY is the data content of the node to be deleted.
Output: A single linked list except the node with data content as KEY.
Data structures: A single linked list whose address of the starting node is known from HEADER.

```
Steps:
 1. ptr1 = HEADER                //Start from the header node
 2. ptr = ptr1.LINK              //This points to the first node, if any
 3. While (ptr ≠ NULL) do
    1. If (ptr.DATA ≠ KEY) then  //If not found the key
        1. ptr1 = ptr            //Keep a track of the pointer of the previous node
        2. ptr = ptr.LINK        //Shift to the next
    2. Else                      //The node is found
        1. ptr1.LINK = ptr.LINK  //Link field of the predecessor is to point
                                 //the successor of node under deletion, see Figure 3.6(c)
        2. RETURNNODE(ptr)       //Return the deleted node to the memory bank
        3. Exit                  //Exit the program
    3. EndIf
 4. EndWhile
 5. If (ptr = NULL) then         //When the desired node is not available in the list
    1. Print "Node with KEY does not exist: No deletion"
 6. EndIf
 7. Stop
```

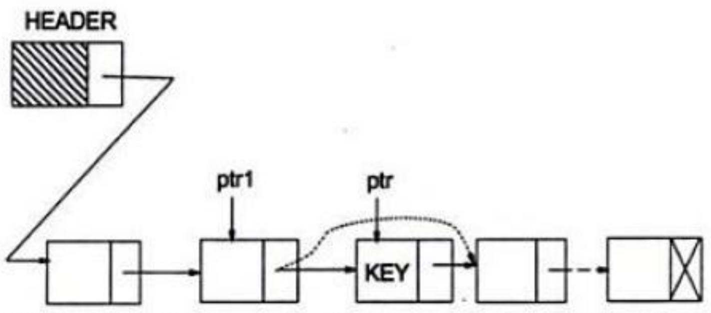


Fig. 3.6(c) Deletion of a node at any position in a single linked list.

Copying a single linked list

For a given list we can copy it into another list by duplicating the content of each node into the newly allocated node. The following is an algorithm to copy an entire single linked list.

```
Algorithm Copy _SL
Input: HEADER is the pointer to the header node of the list.
Output: HEADERl· is the pointer to the duplicate list.
Data structures: Single linked list structure.

Steps:
 1. ptr = HEADER                    // Current position in the master list
 2. HEADERl = GetNode(NODE)         // Get a node for the header of the duplicate list
 3. ptrl = HEADERl                  // ptrl is the current position in the duplicated list
 4. ptrl→DATA = NULL                // Header node does not contain any data
 5. While (ptr != NULL) do          // Till the traversal of master node is finished
 6.      new = GetNode(NODE)        // Get a new node from memory bank
 7.      new→DATA = ptr~DATA        // Copy the content
 8.      ptrl →LINK = new           // Insert the node at the end of the duplicate list
 9.      new→LINK = NULL
10.      ptrl = new
11.      ptr = ptr→LINK                     // Move to the next node in the master list
12. EndWhile
13. Stop
```

Merging two single linked lists into one list

Two single linked lists, namely Ll and L2 are available and we want to merge the list L2 after Ll. Also assume that, *HEADER1* and *HEADER2* are the header nodes of the lists L1 and L2, respectively. Merging can be done by setting the pointer of the link field of the last node in the list L1 with the pointer of the first node in L2. The header node in the list L2 should be returned to the pool of free storage. Merging two single linked lists into one list is illustrated in Figure.

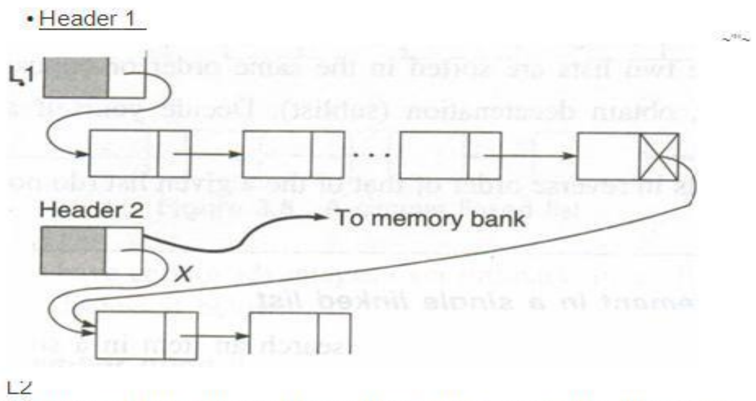


Figure 3.7 Merging two single linked lists into one single linked list.

```
Algorithm MERGE_SL(HEADER1, HEADER2; HEADER)
   Input:  HEADER1 and HEADER2 are pointers to header nodes of lists (L1 and L2,
respectively) to be merged.
   Output:  HEADER is the pointer to the resultant list.
   Data structures:  Single linked list structure.

   Steps:
    1. ptr = HEADER1
    2. While (ptr.LINK ≠ NULL) do   //Move to the last node in the list L1
       1. ptr = ptr.LINK
    3. EndWhile
    4. ptr.LINK = HEADER2.LINK      //Last node in L1 points to the first node in L2
    5. RETURNNODE(HEADER2)          //Return the header node to the memory bank
    6. HEADER = HEADER1             //HEADER becomes the header node of the merged list
    7. Stop
```

Searching for an *element in* a *single linked list*

The algorithm *Search_SL()* is given below to search an item in a single linked list.

```
Algorithm SEARCH_SL(Key; LOCATION)
Input:   KEY, the item to be searched.
Output:   LOCATION, the pointer to a node where the KEY belongs to or an error messages.
Data structures:   A single linked list whose address of the starting node is known from HEADER.

Step:
 1. ptr = HEADER.LINK                 //Start from the first node
 2. flag = 0, LOCATION = NULL
 3. While (ptr ≠ NULL) and (flag = 0) do
    1. If (ptr.DATA = KEY) then
       1. flag = 1                    //Search is finished
       2. LOCATION = ptr
       3. Return (LOCATION)
    2. Else
       1. ptr = ptr.LINK              //Move to the next node
    3. EndIf
 4. EndWhile
 5. If (ptr = NULL) then
    1. Print "Search is unsuccessful"
 6. EndIf
 7. Stop
```

3. CIRCULAR LINKED LIST

In a single linked list, the link field of the last node is null (hereafter a single linked list may be read as ordinary linked list), but a number of advantages can be gained if we utilize this link field to store the pointer of the header node. A linked list where the last node points the header node is called the *circular* linked list. Figure shows a pictorial representation of a circular linked list.

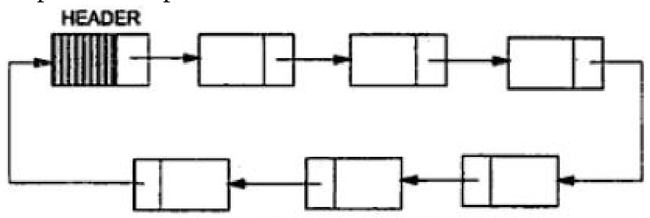


Fig. 3.8 A circular linked list.

Circular linked lists have certain advantages over ordinary linked lists. Some advantages of circular linked lists are discussed below:

Accessibility* of a *member node in the list

In an ordinary list, a member node is accessible from a particular node, that is, from the header node only. But in a circular linked list, every member node is accessible from any node by merely chaining through the list.

Example: Suppose, we are manipulating some information which is stored in a list. Also, think of a case where for a given data, we want to find the earlier occurrence(s) as well as post occurrence(s). Post occurrence(s) can be traced out by chaining through the list from the current node irrespective of whether the list is maintained as a circular linked or an ordinary linked list. In order to find all the earlier occurrences, in case of ordinary linked lists, we have to start our traversing from the header node at the cost of maintaining the pointer for the header in addition to the pointers for the current node and another for chaining. But in the case of a circular linked list, one can trace out the same without maintaining the header information, rather maintaining only two pointers. Note that in ordinary linked lists, one can chain through left to right only whereas it is virtually in both the directions for circular linked lists.

Null link problem

The null value in the link field may create some problem during the execution of programs if proper care is not taken. This is illustrated below by mentioning two algorithms to perform search on ordinary linked lists and circular linked lists.

Algorithm SEARCH_SL(KEY)

Input: KEY the item to be searched.

Output: Location, the pointer to a node where KEY belongs or an error message.

Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

```
1. ptr = HEADER.LINK
2. While ($ptr \neq NULL$) do
    1. If ($ptr.DATA \neq KEY$) then
        1. ptr = ptr.LINK
    2. Else
        1. Return(ptr)
        2. Exit
        3. EndIf
3. EndWhile
4. If (ptr = NULL) then
    1. Print "The entire list has traversed but KEY is not found"
5. EndIf
6. Stop
```

Note that, here two tests in step 2 (which control the loop of searching) cannot be placed together as `While (ptr≠NULL) AND (ptr.DATA≠KEY)` do because in that case there will be an execution error for ptr.DATA since it is not defined when ptr = NULL. But with circular linked list very simple implementation is possible without any special care for NULL pointer. As an illustration the searching of an element in a circular linked list is given below:

Algorithm SEARCH_CL(KEY)

Input: KEY the item of search.

Output: Location, the pointer to a node where KEY belongs or an error message.

Data structures: A circular linked list whose address to the starting node is known from HEADER.

Steps:

```
1. ptr = HEADER.LINK
2. While ($ptr.DATA \neq KEY$) and ($ptr \neq HEADER$) do
    1. ptr = ptr.LINK
3. EndWhile
4. If (ptr.DATA = KEY)
    1. Return (ptr)
5. Else
    1. Print "Entire list is searched: KEY node is not found"
6. EndIf
7. Stop
```

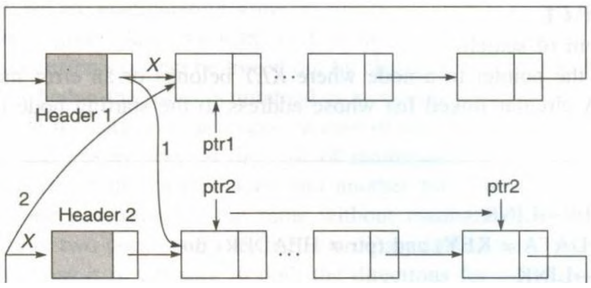
Some easy-to-implement operations

Some operations can more easily be implemented with a circular linked list than with an ordinary linked list. Operations like merging (concatenation), splitting (decatenation), deleting, disposing of an entire list, etc.

can easily be performed on circular linked list. The merging operation, as in Figure 3.9, is explained in the algorithm *Merge_CL* as follows:

```
Algorithm Merge_CL
Input: HEADER1 and HEADER2 are the two pointers to header nodes.
Output: A larger circular linked list containing all the nodes from lists HEADER1 and
HEADER2.
Data structures: Circular linked list structure.

Steps:
1. ptrl = HEADERl→LINK
2. ptr2 = HEADER2→LINK
3. HEADERl→LINK = ptr2                         // Pointer assignment 1 (Figure 3.9)
4. While (ptr2→LINK != HEADER2) do             // Move to the node just preceding the node
          HEADER2
5.    ptr2 = ptr2→LINK
6. EndWhile
7. ptr2→LINK = ptrl                            // Pointer assignment 2 (Figure 3.9)
8. ReturnNode(HEADER2)                         // Return the HEADER2 to the free storage pool
9. Stop
```



One can easily compare the algorithm *Merge_CL* with the algorithm *Merge_SL*. In the algorithm *Merge_SL*, the entire list is needed to be traversed in order to locate the last node, which is not required in the algorithm *Merge_CL*. This implies that *Meger _CL* works faster than *Merge_SL*.

Circular linked lists have some disadvantages as well. One main disadvantage is that without adequate care in processing, it is possible to get trapped into an infinite loop! This problem occurs when we are unable to detect the end of the list while moving from one node to the next. To get rid of this problem, we have to maintain a special node whose data content is possibly NULL, as such a node does not contain any valid information, so it is nothing but just a wastage of memory space.

4. DOUBLE LINKED LISTS

In a single linked list one can move beginning from the header node to any node in one direction only (from left to right). This is why a single linked list is also termed a *one-way* list. On the other hand, a double linked list is a *two-way* list because one can move in either direction, either from left to right or from right to left. This is accomplished by maintaining two link fields instead of one as in a single linked list. A structure of a node for a double linked list is represented as in Figure.

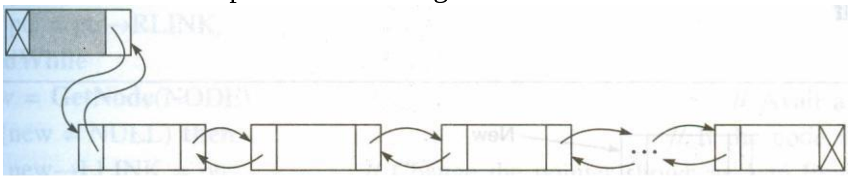


Figure Structure of a node and a double linked list.

From the figure, it can be noticed that two links, viz. RLINK and LLINK, point to the nodes on the right side and left side of the node, respectively. Thus, every node, except the header node and the last node, points to its immediate predecessor and immediate successor.

Operations on a Double Linked List

In this section, only the insertion and deletion operations are discussed.

Inserting a *node into* a *double linked list*

Figure shows a schematic representation of various cases of inserting a node into a double linked list. Let us consider the algorithms of various cases of insertion.

Inserting a node in the front

The algorithm *Insertlironl Dl:* is used to define the insertion operation in a double linked list.

Algorithm INSERT_DL_FRONT(*X*)

Input: *X* the data content of the node to be inserted.

Output: A double linked list enriched with a node containing data as *X* at the front.

Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

```
1. ptr = HEADER.RLINK             //Points to the first node
2. new = GETNODE(NODE)            //Avail a new node from the memory bank
3. If (new ≠ NULL) then           //If new node is available
   1. new.LLINK = HEADER          //Newly inserted node points the header as 1 in Figure
                                  //3.11(a)
   2. HEADER.RLINK = new          //Header now points to the new node as 2 in Figure
                                  //3.11(a)

   3. new.RLINK = ptr             //See the change in pointer shown as 3 in Figure 3.11(a)
   4. ptr.LLINK = new             //See the change in pointer shown as 4 in Figure 3.11(a)
   5. new.DATA = X                //Copy the data into newly inserted node
4. Else
   1. Print "Unable to allocate memory: Insertion is not possible"
5. EndIf
6. Stop
```

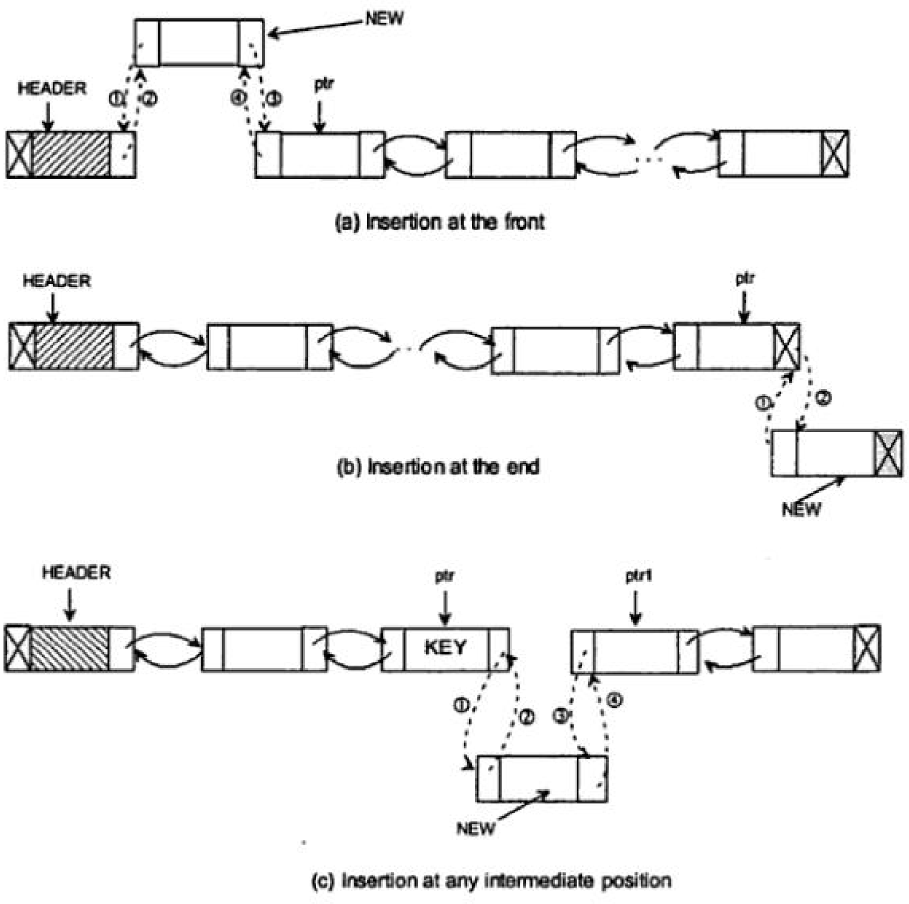


Fig. 3.11 Insertion of a node at various positions in a double linked list.

Inserting a node at the end

The algorithm *InsertEnd_DL* is to insert a node at the end into a double linked list.

Algorithm INSERT_DL_END(*X*)

Input: *X* the data content of the node to be inserted.

Output: A double linked list enriched with a node containing data as *X* at the end of the list.

Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

```
1. ptr = HEADER
2. While (ptr.RLINK ≠ NULL) do          //Move to the last node
   1. ptr = ptr.RLINK
3. EndWhile
4. new = GETNODE(NODE)                   //Avail a new node
5. If (new ≠ NULL) then                  //If the node is available
   1. new.LLINK = ptr                    //Change the pointer shown as 1 in Figure 3.11(b)
   2. ptr.RLINK = new                    //Change the pointer shown as 2 in Figure 3.11(b)
   3. new.RLINK = NULL                   //Make the new node as the last node
   4. new.DATA = X                       //Copy the data into the new node
6. Else
   1. print "Unable to allocate memory: Insertion is not possible"
7. EndIf
8. Stop
```

Inserting a node at any position in the list

The algorithm *InsertAny_DL* is used to insert a node at any position into a double linked list.

Algorithm INSERT_DL_ANY(*X*, KEY)

Input: *X* be the data content of the node to be inserted, and KEY the data content of the node after which the new node to be inserted.

Output: A double linked list enriched with a node containing data as *X* after the node with data KEY, if any.

Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

```
1. ptr = HEADER
2. While (ptr.DATA ≠ KEY) and (ptr.RLINK ≠ NULL) do   // Move to the key node if the current
                                                       //node is not the KEY node or list reaches at the end
   1. ptr = ptr.RLINK
3. EndWhile
4. new = GETNODE(NODE)                 //Get a new node from the pool of free storage
5. If (new = NULL) then                //When the memory is not available
   1. Print "Memory is not available"
   2. Exit                             //Quit the program
6. EndIf
7. If (ptr.RLINK = NULL)               //If the KEY node is at the end or not found in the list
   1. new.LLINK = ptr
   2. ptr.RLINK = new                  //Insert at the end
   3. new.RLINK = NULL
   4. new.DATA = X                     //Copy the information to the newly inserted node
8. Else                                //The KEY is available
   1. ptr1 = ptr.RLINK                 //Next node after the key node
   2. new.LLINK = ptr                  //Change the pointer shown as 2 in Figure 3.11(c)
   3. new.RLINK = ptr1                 //Change the pointer shown as 4 in Figure 3.11(c)
   4. ptr.RLINK = new                  //Change the pointer shown as 1 in Figure 3.11(c)
   5. ptr1.LLINK = new                 //Change the pointer shown as 3 in Figure 3.11(c)
   6. ptr = new                        //This becomes the current node
   7. new.DATA = X                     //Copy the content to the newly inserted node
9. EndIf
10. Stop
```

Note: Observe that the algorithm INSERT_DL_ANY will insert a node even the key node does not exist. In that case the node will be inserted at the end of the list. Also, note that the algorithm will work even if the list is empty.

<u>Deleting a node from a double linked list</u>

Deleting a node from a double linked list may take place from any position in the list, as shown in Figure. Let us consider each of those cases separately. Deleting a node from the front of a double linked list

Algorithm DELETE_DL_FRONT( )

Input: A double linked list with data.
Output: A reduced double linked list.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

```
Steps:
 1. ptr = HEADER.RLINK                 //Pointer to the first node
 2. If (ptr = NULL) then               //If the list is empty
    1. Print "List is empty: No deletion is made"
    2. Exit
 3. Else
    1. ptr1 = ptr.RLINK                //Pointer to the second node
    2. HEADER.RLINK = ptr1             //Change the pointer shown as 1 in Figure 3.12(a)
    3. If (ptr1 ≠ NULL)                //If the list contains a node after the first node
                                       //of deletion
       1. ptr1.LLINK = HEADER          //Change the pointer shown as 2 in Figure 3.12(a)

    4. EndIf
    5. RETURNNODE (ptr)                //Return the deleted node to the memory bank
 4. EndIf
 5. Stop
```

<u>Deleting a node at the end of a double linked list</u>

The algorithm is as follows:

Algorithm DELETE_DL_END( )

Input: A double linked list with data.
Output: A reduced double linked list.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

```
Steps:
 1. ptr = HEADER
 2. While (ptr.RLINK ≠ NULL) do        //Move to the last node
    1. ptr = ptr.RLINK
 3. EndWhile
 4. If (ptr = HEADER) then             //If the list is empty
    1. Print "List is empty: No deletion is made"
    2. Exit                            //Quit the program
 5. Else
    1. ptr1 = ptr.LLINK                //Pointer to the last but one node
    2. ptr1.RLINK = NULL               //Change the pointer shown as 1 in Figure 3.12(b)
    3. RETURNNODE(ptr)                 //Return the node to the memory bank
 6. EndIf
 7. Stop
```

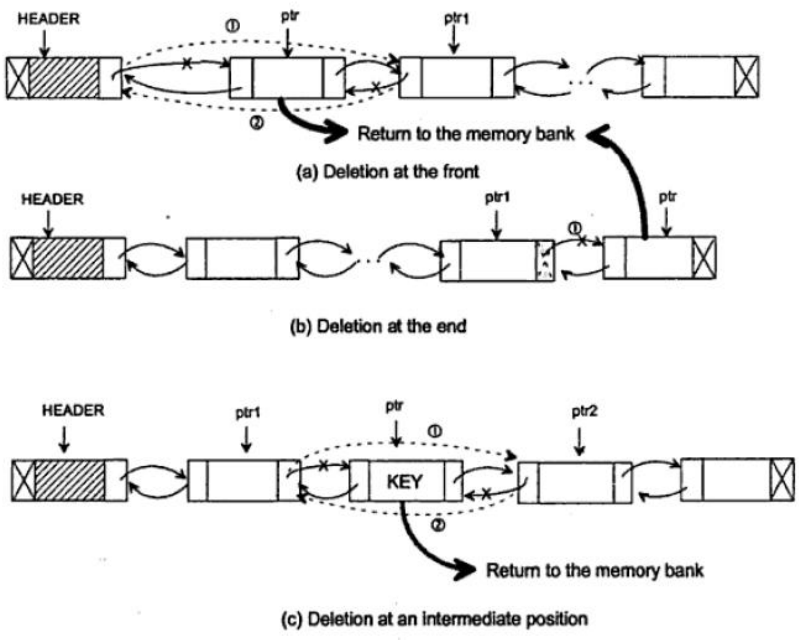
<u>Deleting a node from any position in a double linked list</u>

The algorithm is as follows:

Algorithm DELETE_DL_ANY(KEY)

Input: A double linked list with data, and KEY, the data content of the key node to be deleted.
Output: A double linked list without a node having data content KEY, if any.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

```
Steps:
 1. ptr = HEADER.RLINK                 //Move to the first node
 2. If (ptr = NULL) then
    1. Print "List is empty: No deletion is made"
    2. Exit
 3. EndIf                              //Quit the program
 4. While (ptr.DATA ≠ KEY) and (ptr.RLINK ≠ NULL) do
                                       //Move to the desired node
    1. ptr = ptr.RLINK
 5. EndWhile
 6. If (ptr.DATA = KEY) then           //If the node is found
    1. ptr1 = ptr.LLINK                //Track to the predecessor node
    2. ptr2 = ptr.RLINK                //Track to the successor node
    3. ptr1.RLINK = ptr2               //Change the pointer as shown 1 in Figure 3.12(c)
    4. If (ptr2 ≠ NULL) then           //If the deleted node is the last node
       1. ptr2.LLINK = ptr1            //Change the pointer shown as 2 in Figure 3.12(c)
    5. EndIf
    6. RETURNNODE(ptr)                 //Return the free node to the memory bank
 7. Else
    Print "The node does not exist in the given list"
 8. EndIf
 9. Stop
```


HEADER
ptr
ptr1
Return to the memory bank
(a) Deletion at the front
HEADER
ptr1
ptr
(b) Deletion at the end
HEADER
ptr1
ptr
ptr2
KEY
Return to the memory bank
(c) Deletion at an intermediate position
Fig. 3.12 Deletion at various position in a double linked list.

5. CIRCULAR DOUBLE LINKED LIST

The advantages of both double linked list and circular linked list are incorporated into another type of list structure called circular double linked list and it is known to be the best of its kind. Figure shows a schematic representation of a circular double linked list.

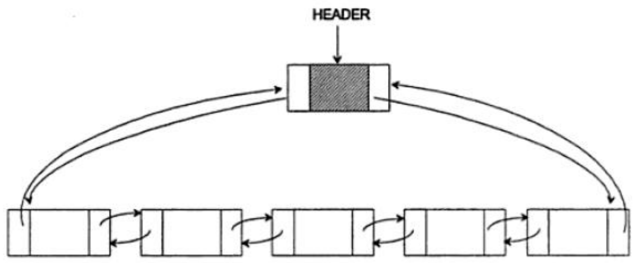


Fig. 3.13 A circular double linked list.

Here, note that the RLINK (right link) of the rightmost node and LLINK (left link) of the leftmost node contain the address of the header node; again the RLINK and LLINK of the header node contain the address of the rightmost node and the leftmost node, respectively. An empty circular double linked list is represented as shown in Figure. In case of an empty list, both LLINK and RLINK of the header node point to itself.

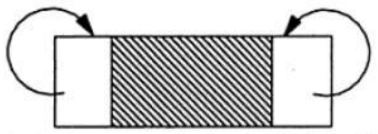


Fig. 3.14 An empty circular double linked list.

Operations on Circular Double Linked List

Sorting operation with a *circular double linked list.*

The algorithm *SorCCDL* shows the sorting of elements stored in a circular double linked list

Algorithm SORT_CDL( )

Input: A circularly double linked with elements.

Output: Sorted version of the circularly double linked list.

Data structures: Circular double linked list structure with HEADER being the pointer to the header node.

```
Steps:
 1. ptr_beg = HEADER.LLINK             //Pointer to the first node—the beginning node
 2. ptr_end = HEADER.RLINK             //Pointer to the last node—the ending node
 3. While (ptr_beg ≠ ptr_end) do       //To traverse the entire list—outer loop
    1. ptr1 = ptr_beg                  //ptr1 and ptr2 are
    2. ptr2 = ptr1.RLINK               //Two variable pointers
    3. While (ptr2 ≠ ptr_end) do           //For compare and interchange—inner loop
        1. If ORDER(ptr1.DATA, ptr2.DATA) = FALSE then
                                           //If keys are not in order
            1. SWAP (ptr1, ptr2)           //Interchange the data content at ptr1 and ptr2
        2. EndIf
        3. ptr1 = ptr1.RLINK               //Move the first pointer to the next
        4. ptr2 = ptr2.RLINK               //Move the second pointer to the next
    4. EndWhile
    5. ptr_end = ptr_end.LLINK             //Right most node is now sorted out
4. EndWhile
5. Stop
```

In the above algorithm, we have assumed the procedure *Order(datal, data2)* to test whether two data are in a desired order or not; it will return TRUE if they are in order else FALSE.

The above algorithm uses the bubble sorting technique. The execution of each outer loop bubbles up the largest node towards the right end of sorting (say, in ascending order) and each inner loop is used to compare the successive nodes and push up the largest towards the right if they are not in order. Figure illustrates the sorting procedure. Students may see whether the algorithm *SorcCDL* is also applicable to the double linked list data structure or not.

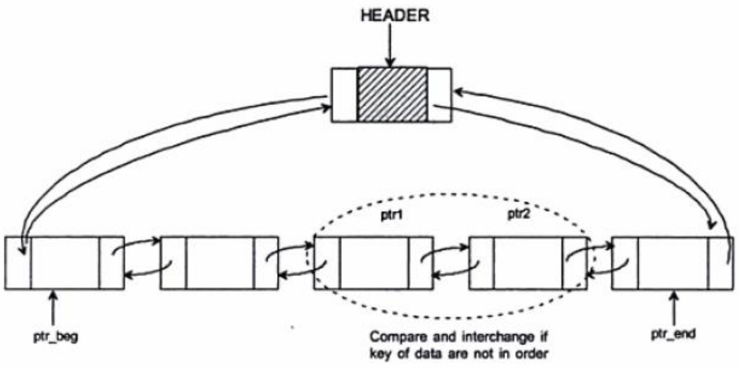


Fig. 3.15 Sorting operation and use of various pointers.

6. APPLICATIONS OF LINKED LISTS

Sparse Matrix Manipulation

In Figure, the fields *i* and *j* store the row and column numbers for a matrix element, respectively. DATA field stores the matrix element at the ith row and the jth column, i.e. a_{ij}. The ROWLINK points the next node in the same row and COLLINK points the next node in the same column. The principle is that all the nodes particularly in a row (column) are circular linked with each other; each row (column) contains a header node. Thus, for a sparse matrix of order *m* x *n,* we have to maintain *m* headers for all rows and *n* headers for all columns, plus one extra node the use of which would be evident from Figure (b). For an illustration, a sparse matrix of order 6 x 5 is assumed, as shown in Figure (a).

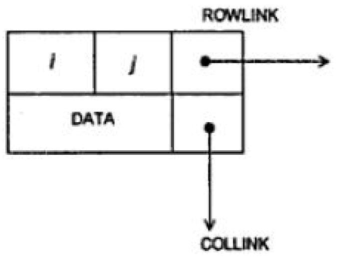


Fig. 3.16 Structure of a node to represent sparse matrices.

Figure (b) describes the representation of a sparse matrix. Here, CHI, CH2, ... , CH5 are the 5 headers heading 5 columns and RHl, RH2, ... , RH6 are the 6 headers heading 6 rows. HEADER is one additional header node keeping the starting address of the sparse matrix.

With this representation, any node is accessible from any other node. Now let us consider the algorithm *Createsparsebdatrxi Ll.* to create a linked list and hence to store a sparse matrix.

Algorithm MAKE_SPARSE_&_INSERT( )

Input: An $m \times n$ sparse matrix.

Output: Linked representation of the sparse matrix.

Data structures: Linked structure for sparse matrix.

Steps:

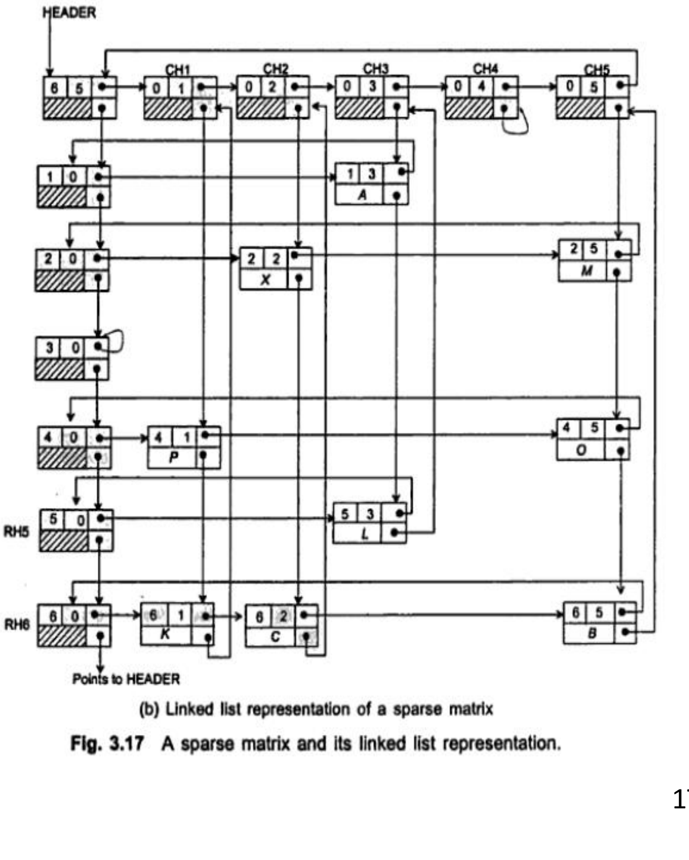
```
1. Read m, n                              //m—number of rows, n—number of columns
   //Get a node for header//
2. HEADER = GETNODE(NODE)        //Get a node from free storage
3. If (HEADER = NULL) then
   1. Print "Non-availability of storage space: Quit"
   2. Exit                         //The program is aborted
4. Else
   //Initialize the node HEADER as the matrix is initially empty//
   1.  HEADER.i = m
   2.  HEADER.j = n
   3.  HEADER.ROWLINK = HEADER
   4.  HEADER.COLLINK = HEADER
   5.  HEADER.DATA = NULL

//Get header nodes for column header//
6.  ptr = HEADER
7.  For col = 1 to n do
    1. new = GETNODE(NODE)
    2. new.i = 0, new.j = col, new.DATA = NULL
    3. ptr.ROWLINK = new
    4. new.ROWLINK = HEADER, new.COLLINK = new
    5. ptr = new
8.  EndFor
    //Get header nodes for row header//
9.  ptr = HEADER
10. For row = 1 to m
    1. new = GETNODE(NODE)
    2. new.i = row, new.j = 0, new.DATA = NULL
    3. ptr.COLLINK = new
    4. new.COLLINK = HEADER, new.ROWLINK = new
    5. ptr = new
11. EndFor
    //Insertion of elements a_ij into the sparse matrix//
12. Read (data, row, col)          //Read the element to be inserted and its position
13. rowheader = HEADER.COLLINK, colheader = HEADER.ROWLINK
    //Go to the row header of the i-th row//
14. While (row < rowheader.i)
    1. rowheader = rowheader.COLLINK
15. EndWhile
    //Go to the column header of the j-th column//
16. While (col < colheader.j)
    1. colheader = colheader.ROWLINK
17. EndWhile
    //Find the position for insertion//
18. rowptr = rowheader                //ON ROW
19. While (rowptr.j < col) do         //This is to find the predecessor and successor
20. 1. ptr1 = rowptr                  //ptr1 points the predecessor i.e. the node at
                                      //j-th column
    2. rowptr = rowptr.ROWLINK        //rowptr is the pointer to successor
    3. If (rowptr = rowheader) then
       1. Break                       //Exit the loop
    4. EndIf
21. EndWhile
22. colptr = colheader                //On Column
23. While (colptr.i < row)            //This is to find the predecessor and successor
    1. ptr2 = colptr                  //ptr2 points the predecessor, i.e. the node in
                                      //i - 1-th row
    2. colptr = colptr. COLLINK       //colptr is the pointer to successor
    3. If (colptr = coolheaded)
       1. Break                       //Exit the loop
    4. EndIf
24. EndWhile

    //Insert the node//
25. new = GETNODE(NODE)          //Get a new node from the free storage
26. If (new = NULL) then
    1. Print "Non-availability of storage space: Quit"
    2. Exit                      //The program is aborted
27. Else                         //Initialize the node to be inserted
    1. ptr1.ROWLINK = new
    2. ptr2.COLLINK = new
    3. new.ROWLINK = rowptr
    4. new.COLLINK = colptr
    5. new.DATA = KEY
28. EndIf
29. If more insert then
    1. Go to Step 4.12
30. EndIf
5. EndIf
6. Stop
```

$$
\begin{array}{cc|ccccc|}
 & & \multicolumn{5}{c}{\text{Column}} \\
 & & 1 & 2 & 3 & 4 & 5 \\
 & 1 & \cdot & \cdot & A & \cdot & \cdot \\
 & 2 & \cdot & X & \cdot & \cdot & M \\
 & 3 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\text{Row} & 4 & P & \cdot & \cdot & \cdot & O \\
 & 5 & \cdot & \cdot & L & \cdot & \cdot \\
 & 6 & K & C & \cdot & \cdot & B
\end{array}
$$

(a) A sparse matrix of order 6 × 5 containing 9 elements only



(b) Linked list representation of a sparse matrix

Fig. 3.17 A sparse matrix and its linked list representation.

Polynomial Representation

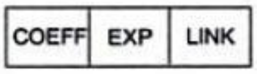
An important application of linked lists is to represent polynomials and their manipulations. The main advantage of a linked list for polynomial representation is that it can accommodate number of polynomials of growing sizes so that their combined size does not exceed the total memory available. The methodology of representing polynomials and the operations on them are discussed in this section. First, let us consider the case of representation of polynomials.

Polynomial with single variable

Let us consider the general form of a polynomial with single variable:

$$P(x) = a_n x^{e_n} + a_{n-1} x^{e_{n-1}} + \cdots + a_1 x^{e_1}$$

where $a_i x^{e_i}$ is a term in the polynomial so that a_i is a non-zero coefficient and e_i is the exponent. We will assume an ordering of the terms in the polynomial such that $e_n > e_{n-1} > \ldots > e_2 > e_1 \geq 0$. The structure of a node in order to represent a term can be decided as shown below:



Considering the single linked list representation, a node should have three fields: COEFF (to store the coefficient *Qi),* EXP (to store the exponent *ei)* and a LINK (to store the pointer to the next node representing the next term). It is evident that the number of nodes required to represent a polynomial is the same as the number of terms in the polynomial. An additional node may be considered for a header. As an example, let us consider that the single linked list representation of the polynomial $P(x) = 3x^8 - 7x^6 + 14x^3 + 10x - 5$ would be stored as shown in Figure.

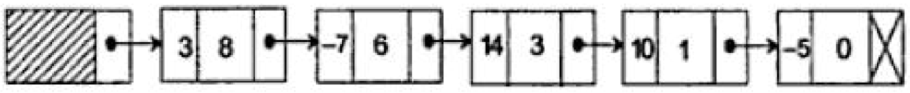


Fig. 3.18 Linked list representation of a polynomial (single variable).

Note that the terms whose coefficients are zero are not stored here. Next let us consider two basic operations, namely the addition and multiplication of two polynomials using this representation.

<u>Polynomial addition</u>

In order to add two polynomials, say P and Q, to get a resultant polynomial R, we have to compare their terms starting at their first nodes and moving towards the end one by one. Two pointers Pptr and Qptr are used to move along the terms of P and Q. There may arise three cases during the comparison between the terms of two polynomials.

Case 1: The exponents of two terms are equal. In this case the coefficients in the two nodes are added and a new term is created with the values

$$\text{Rptr}\rightarrow\text{COEFF} = \text{Pptr}\rightarrow\text{COEFF} + \text{Qptr}\rightarrow\text{COEFF}$$

and

$$\text{Rptr}\rightarrow\text{EXP} = \text{Pptr}\rightarrow\text{EXP}$$

Case 2: Pptr→EXP > Qptr→EXP, i.e. the exponent of the current term in P is greater than the exponent of the current term in Q. Then, a duplicate of the current term in P is created and inserted in the polynomial R.

Case 3: Pptr→EXP < Qptr→EXP, i.e. the case when the exponent of the current term in P is less than the exponent of the current term in Q. In this case, a duplicate of the current term of Q is created and inserted in the polynomial R. The algorithm *PolynomialAdd_LL* is described as below:

Algorithm PolynomialAdd_LL

Input: Two polynomials P and Q whose header pointers are *PHEADER* and *QHEADER.*

Output: A polynomial R is the sum of P and Q having the header *RHEADER.*

Data structure: Single linked list structure for representing a term in a single variable polynomial.

```
1. Pptr = PHEADER→LINK, Qptr = QHEADER→LINK
   //Get a header node for the resultant polynomial//
2. RHEADER = GetNode(NODE)
3. RHEADER→LINK = NULL, RHEADER→EXP = NULL, RHEADER→COEFF = NULL
```

```
4.  Rptr = RHEADER // Current pointer to the resultant polynomial R
5.  While (Pptr != NULL) and (Qptr != NULL) do
6.     CASE: Pptr→EXP = Qptr→EXP                                    // Case 1
7.        new = GetNode (NODE)
8.        Rptr→LINK = new, Rptr = new
9.        Rptr→COEFF = Pptr→COEFF + Qptr→COEFF
10.       Rptr→EXP = Pptr→EXP
11.       Rptr→LINK = NULL
12.       Pptr = Pptr→LINK, Qptr = Qptr→LINK
13.    CASE: Pptr→EXP > Qptr→EXP                                    // Case 2
14.       new = GetNode (NODE)
15.       Rptr→LINK = new, Rptr = new
16.          Rptr→COEFF = Pptr→COEFF
17.          Rptr→EXP = Pptr→EXP
18.          Rptr→LINK = NULL
19.          Pptr = Pptr→LINK
20.   CASE: Pptr→EXP < Qptr→EXP                                     // Case 3
21.             new = GetNode (NODE)
22.             Rptr→LINK = new, Rptr=new
23.             Rptr→COEFF = Qptr→COEFF
24.             Rptr→EXP = Qptr→EXP
25.             Rptr→LINK = NULL
26.         Qptr = Qptr→LINK
27.   EndWhile
28.      If (Pptr !=NULL) and (Qptr = NULL) then          // To add the trailing part of P, if any
29.      While (Pptr != NULL) do
30.         new = GetNode(NODE
31.         Rptr→LINK= new, Rptr = new
32.         Rptr→COEFF = Pptr→COEFF
33.         Rptr→EXP = Pptr→Exp
34.         Rptr→LINK = NULL
35.         Pptr = Pptr→LINK
36.      EndWhile
37.   EndIf
38.   If (Pptr = NULL) and (Qptr != NULL) then             //To add the trailing part of Q, if any
39.      While (Qptr !=NULL) do
40.         new = GetNode→NODE)
41.         Rptr→LINK = new, Rptr = new
42.         Rptr→COEFF = Qptr→COEFF
43.         Rptr→EXP = Qptr→EXP
44.         Rptr→LINK = NULL
45.         Qptr = Qptr→LINK
46.      EndWhile
47.   EndIf
48.   Return(RHEADER)
49.   Stop
```

Polynomial Multiplication

Suppose, we have to multiply two polynomials P and Q so that the result will be stored in R, another polynomial. The method is quite straightforward: let Pptr denote the current term in P and Qptr be that of in Q. For each term of P we have to visit all the terms in Q; the exponent values in two terms are added (R→EXP = P→EXP + Q→EXP), the coefficient values are multiplied (R→COEFF = P→COEFF x Q→COEFF), and these values are included into R in such a way that if there is no term in R whose exponent value is the same as the exponent value obtained by adding the exponents from P and Q, then create a new node and insert it to *R* with the values so obtained (that is, R→COEFF, and R→EXP); on the other hand, if a node is found in R having same exponent value R→EXP, then update the coefficient value of it by adding the resultant coefficient (R→COEFF) into it. The algorithm *PolynomialMultiply_LL* is described as below:

Algorithm PolynomialMultiply _LL

Input: Two polynomials P and Q having their headers as *PHEADER, QHEADER.*
Output: A polynomial R storing the result of multiplication of P and Q.
Data structure: Single linked list structure for representing a term in a single variable polynomial.

```
Pptr = PHEADER, Qptr = QHEADER
        1* Get a node for the header of R */
RHEADER = GetNode(NODE)
RHEADER→LINK = NULL, RHEADER→COEFF = NULL, RHEADER→EXP = NULL
If (Pptr→LINK = NULL) or (Qptr→LINK = NULL) then
            Exit      II No valid operation possible
EndIf
 Pptr = Pptr→LINK
 While (Pptr !=. NULL) do                                   II For each term of P
        While (Qptr !=. NULL) do
                 C = Pptr→COEFF x Qptr→COEFF
                  X = Ppt→EXP + Qptr→EXP
        /* Search for the equal exponent value in R */
                 Rptr = RHEADER
                 While (Rptr !=. NULL) and (Rptr→EXP > X) do
                         Rptrl = Rptr
                         Rptr = Rptr→LINK
                         If (Rptr→EXP = X) then
                                     Rpt→COEFF = Rptr→COEFF + C
                        Else                        II Add a new node at the correct position in R
                                     new = GetNode(NODE)
                                     new→EXP = X, new→COEFF = C
                                     If (Rptr→LINK = NULL) then
                                           Rptr→LINK = new             II Append the node at the end
                                          new~LINK = NULL
                                      Else
                                          Rptr1→LINK = new         II Insert the node in ordered position
                                            New→LINK = Rptr
                                    Endff
                         Endff
                 EndWhile
        EndWhile
EndWhile
Return (RHEADER)
Stop
```

Dynamic Storage Management

The basic task of any program is to manipulate data. These data should be stored in memory during their manipulation. There are two memory management schemes for the storage allocations of data:

1. Static storage management
2. Dynamic storage management

In the case of the *static storage management* scheme, the net amount of memory required for various data for a program is allocated before the start of the execution of the program. Once memory is allocated, it can neither be extended nor be returned to the memory bank for the use of other programs at the same time. On the other hand, the *dynamic storage management* scheme allows the user to allocate and deallocate as per the requirement during the execution of programs. This dynamic memory management scheme is suitable in multiprogramming as well as in single-user environment where generally more than one program reside in the memory and their memory requirement can be known only during their execution. An operating system (OS) generally provides the service of dynamic memory management. The data structure for implementing such a scheme is a linked list.