MODULE 3: LINKED LIST

DEFINITION

A linked list, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. That is, each node is divided into two parts:

- The first part contains the information of the element, and
- The second part, called the ***link field*** **or** ***nextpointer*** field, contains the address of the next node in the list.

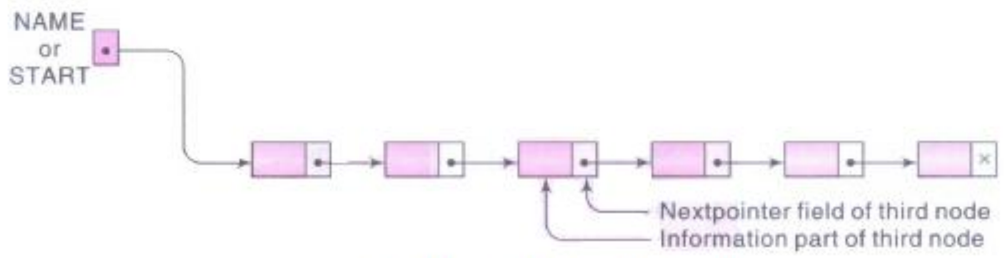

NAME
or
START
Nextpointer field of third node
Information part of third node

Fig: Linked list with 6 nodes

In the above figure each node is pictured with two parts.

- The left part represents the information part of the node, which may contain an entire record of data items.
- The right part represents the nextpointer field of the node
- An arrow drawn from a node to the next node in the list.
- The pointer of the last node contains a special value, called the *null pointer*, which is any invalid address.

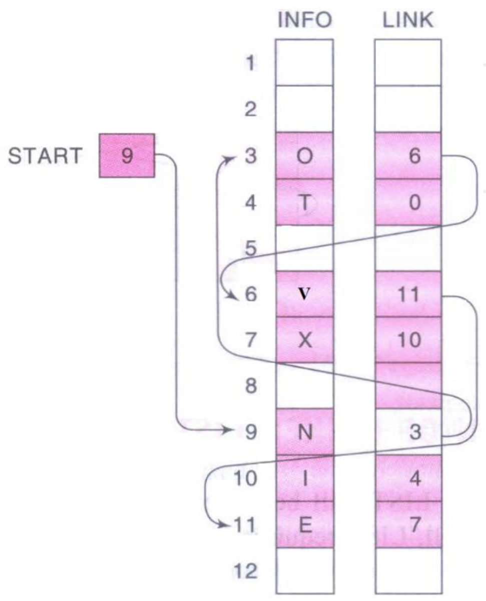
A pointer variable called **START or FIRST** which contains the address of the first node.
A special case is the list that has no nodes, such a list is called the ***null list or empty list*** and is denoted by the null pointer in the variable START.

REPRESENTATION OF LINKED LISTS IN MEMORY

Let LIST be a linked list. Then LIST will be maintained in memory as follows.

1. LIST requires two linear arrays such as INFO and LINK-such that INFO[K] and LINK[K] contains the information part and the nextpointer field of a node of LIST.
2. LIST also requires a variable name such as START which contains the location of the beginning of the list, and a nextpointer sentinel denoted by NULL-which indicates the end of the list.
3. The subscripts of the arrays INFO and LINK will be positive, so choose NULL = 0, unless otherwise stated.

The following examples of linked lists indicate that the nodes of a list need not occupy adjacent elements in the arrays INFO and LINK, and that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.



START=9	INFO[9]=N
LINK[3]=6	INFO[6]=V
LINK[6]=11	INFO[11]=E
LINK[11]=7	INFO[7]= X
LINK[7]=10	INFO[10]= I
LINK[10]=4	INFO[4]= T

LINK[4]= NULL value, So the list has ended

REPRESENTING CHAIN IN C

The following capabilities are needed to make linked representation

1. A mechanism for defining a node's structure, that is, the field it contains. So self-referential structures can be used
2. A way to create new nodes, so MALLOC functions can do this operation
3. A way to remove nodes that no longer needed. The FREE function handles this operation.

Defining a node structure

```
typedef struct listNode *listPointer
typedef struct {
                    char data[4];
                    listPointer list;
                } listNode;
```

Create a New Empty list

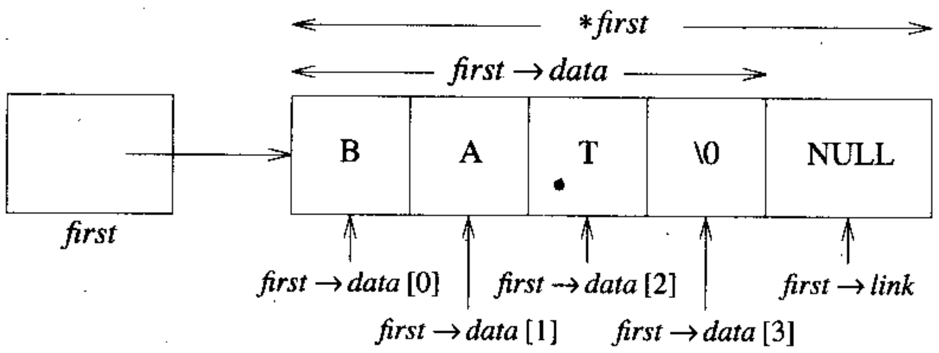
```
                    listPointer first = NULL
```

To create a New Node

```
                MALLOC (first, sizeof(*first));
```

To place the data into NODE

```
              strcpy(first→ data,"BAT");
              first→ link = NULL
```

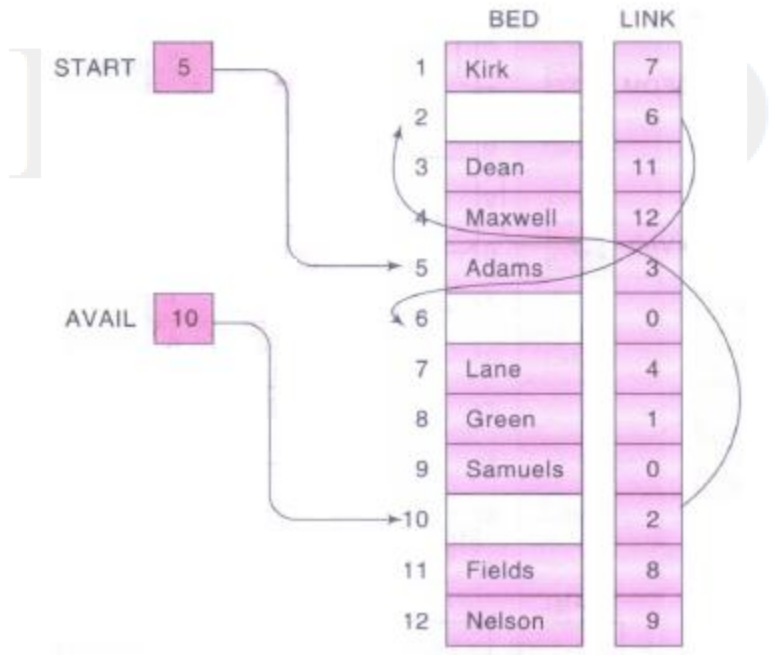


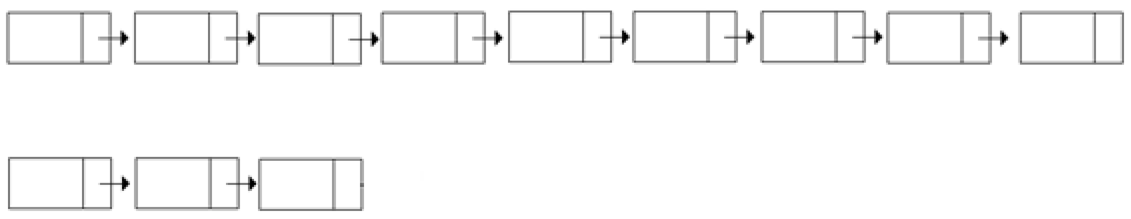
MEMORY ALLOCATION - GARBAGE COLLECTION

- The maintenance of linked lists in memory assumes the possibility of inserting new nodes into the lists and hence requires some mechanism which provides unused memory space for the new nodes.
- Mechanism is required whereby the memory space of deleted nodes becomes available for future use.
- Together with the linked lists in memory, a special list is maintained which consists of unused memory cells. This list, which has its own pointer, is called ***the list of available space or the free storage list or the free pool***.

Suppose linked lists are implemented by parallel arrays and insertions and deletions are to be performed linked lists. Then the unused memory cells in the arrays will also be linked together to form a linked list using AVAIL as its list pointer variable. Such a data structure will be denoted by

LIST (INFO, LINK, START, AVAIL)





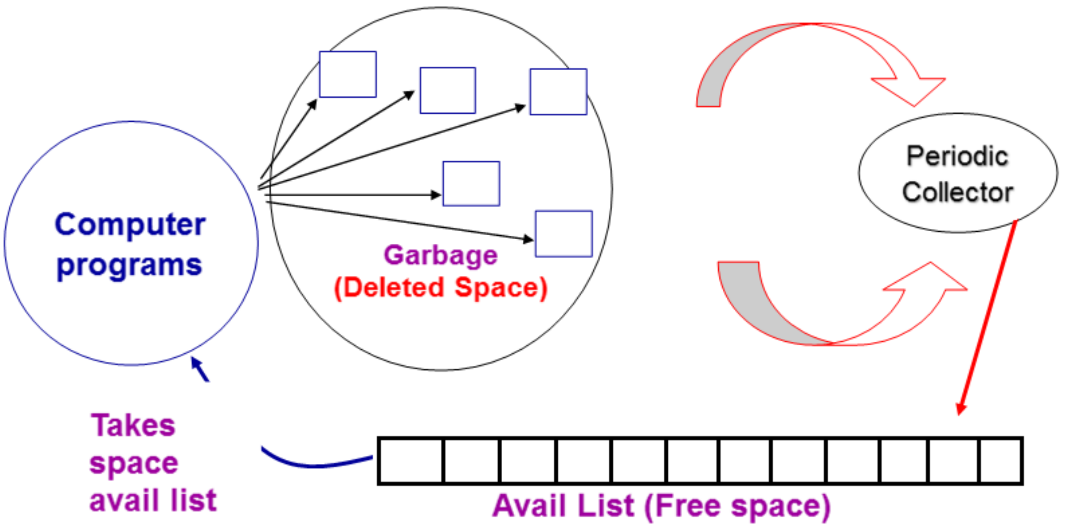
Garbage Collection

- Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program. So space is need to be available for future use.
- One way to bring this is to immediately reinsert the space into the free-storage list. However, this method may be too time-consuming for the operating system of a computer, which may choose an alternative method, as follows.

The operating system of a computer may periodically collect all the deleted space onto the freestorage list. Any technique which does this collection is called garbage collection.
Garbage collection takes place in two steps.

1. First the computer runs through all lists, tagging those cells which are currently in use
2. And then the computer runs through the memory, collecting all untagged space onto the free-storage list.

The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list, or when the CPU is idle and has time to do the collection.



Overflow

- Sometimes new data are to be inserted into a data structure but there is no available space, i.e., the free-storage list is empty. This situation is usually called ***overflow***.
- The programmer may handle overflow by printing the message OVERFLOW. In such a case, the programmer may then modify the program by adding space to the underlying arrays.
- Overflow will occur with linked lists when AVAIL = NULL and there is an insertion.

Underflow

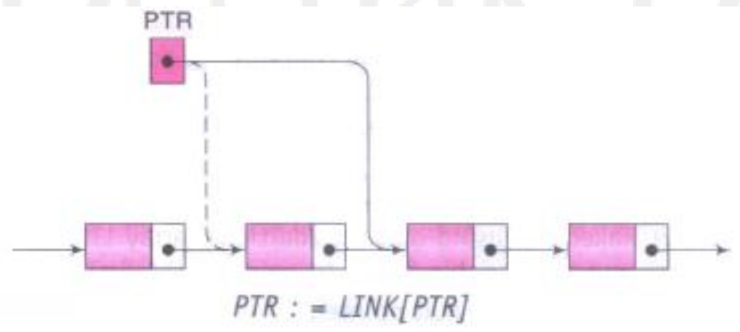
- The term underflow refers to the situation where one wants to delete data from a data structure that is empty.
- The programmer may handle underflow by printing the message UNDERFLOW.
- The underflow will occur with linked lists when START = NULL and there is a deletion.

LINKED LIST OPERATIONS

1. Traversing a Linked list

Let LIST be a linked list in memory stored in linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST.

- Traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed.
- PTR→LINK points to the next node to be processed.
- Thus the assignment PTR= PTR→LINK moves the pointer to the next node in the list, as pictured in below figure



Algorithm: (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST.
The variable PTR points to the node currently being processed.

```
1. Set PTR = START
2. Repeat Steps 3 and 4 while PTR ≠ NULL
3.      Apply PROCESS to PTR→INFO
4.      Set PTR = PTR→LINK
5. Exit.
```

The details of the algorithm are as follows.

- Initialize PTR or START.
- Then process PTR→INFO, the information at the first node.
- Update PTR by the assignment PTR = PTR→LINK, so that PTR points to the second node. Then process PTR→INFO, the information at the second node. Again update PTR by the assignment PTR = PTR→LINK, and then process PTR→INFO, the information at the third node. And so on. Continue until PTR = NULL, which signals the end of the list.

Example:

The following procedure prints the information at each node of a linked list. Since the procedure must traverse the list.

Procedure: PRINT (INFO, LINK, START)

```
1. Set PTR = START.
2. Repeat Steps 3 and 4 while PTR ≠ NULL:
3.     Write: PTR→INFO
4.     Set PTR = PTR→LINK
5. Return.
```

2. Searching a Linked list

There are two searching algorithm for finding location LOC of the node where ITEM first appears in LIST.

Let LIST be a linked list in memory. Suppose a specific ITEM of information is given.
If ITEM is actually a key value and searching through a file for the record containing ITEM, then ITEM can appear only once in LIST.

LIST Is Unsorted

Suppose the data in LIST are not sorted. Then search for ITEM in LIST by traversing through the list using a pointer variable PTR and comparing ITEM with the contents PTR→INFO of each node, one by one, of LIST. Before updating the pointer PTR by

PTR = PTR→LINK

It requires two tests.
First check whether we have reached the end of the list, i.e.,

PTR == NULL

If not, then check to see whether

PTR→INFO == ITEM

```
Algorithm: SEARCH (INFO, LINK, START, ITEM, LOC)
LIST is a linked list in memory. This algorithm finds the location LOC of the node where
ITEM first appears in LIST, or sets LOC = NULL.
        1. Set PTR: = START.
        2. Repeat Step 3 while PTR ≠ NULL
        3.      If ITEM = PTR→INFO, then:
                        Set LOC: = PTR, and Exit.
                Else
                        Set PTR: = PTR→LINK
                [End of If structure.]
                [End of Step 2 loop.]
        4. [Search is unsuccessful.] Set LOC: = NULL.
        5. Exit.
```

The complexity of this algorithm for the worst-case running time is proportional to the number *n* of elements in LIST, and the average-case running time is approximately proportional to *n/2* (with the condition that ITEM appears once in LIST but with equal probability in any node of LIST).

LIST is Sorted

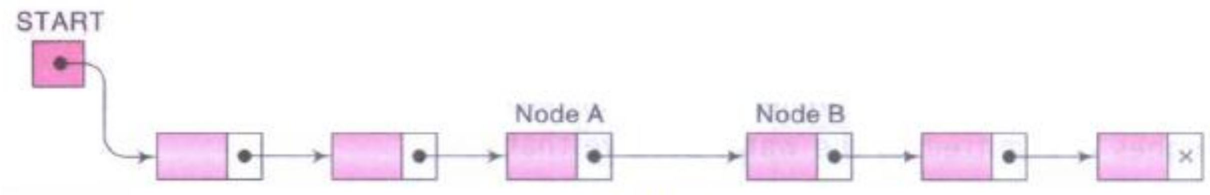
Suppose the data in LIST are sorted. Search for ITEM in LIST by traversing the list using a pointer variable PTR and comparing ITEM with the contents PTR→INFO of each node, one by one, of LIST. Now, searching can stop once ITEM exceeds PTR→INFO.

```
Algorithm: SRCHSL (INFO, LINK, START, ITEM, LOC)
LIST is a sorted list in memory. This algorithm finds the location LOC of the node where
ITEM first appears in LIST, or sets LOC = NULL.
        1. Set PTR: = START.
        2. Repeat Step 3 while PTR ≠ NULL
        3.      If ITEM < PTR→INFO, then:
                        Set PTR: = PTR→LINK
                Else if ITEM = PTR→INFO, then:
                        Set LOC: = PTR, and Exit. [Search is successful.]
                Else:
                        Set LOC: = NULL, and Exit. [ITEM now exceeds PTR→INFO]
                [End of If structure.]
                [End of Step 2 loop.]
        4. Set LOC: = NULL.
        5. Exit.
```

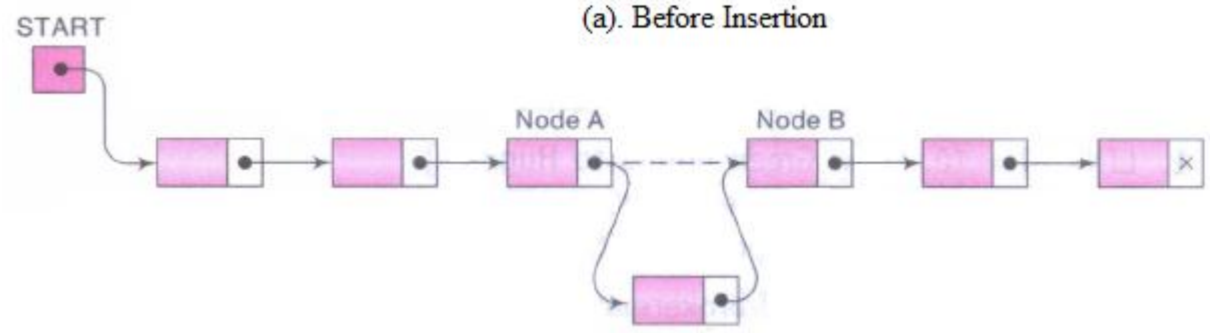
The complexity of this algorithm for the worst-case running time is proportional to the number *n* of elements in LIST, and the average-case running time is approximately proportional to *n/2*

3. Insertion into a Linked list

Let LIST be a linked list with successive nodes A and B, as pictured in Fig. (a). Suppose a node N is to be inserted into the list between nodes A and B. The schematic diagram of such an insertion appears in Fig. (b). That is, node A now points to the new node N, and node N points to node B, to which A previously pointed.

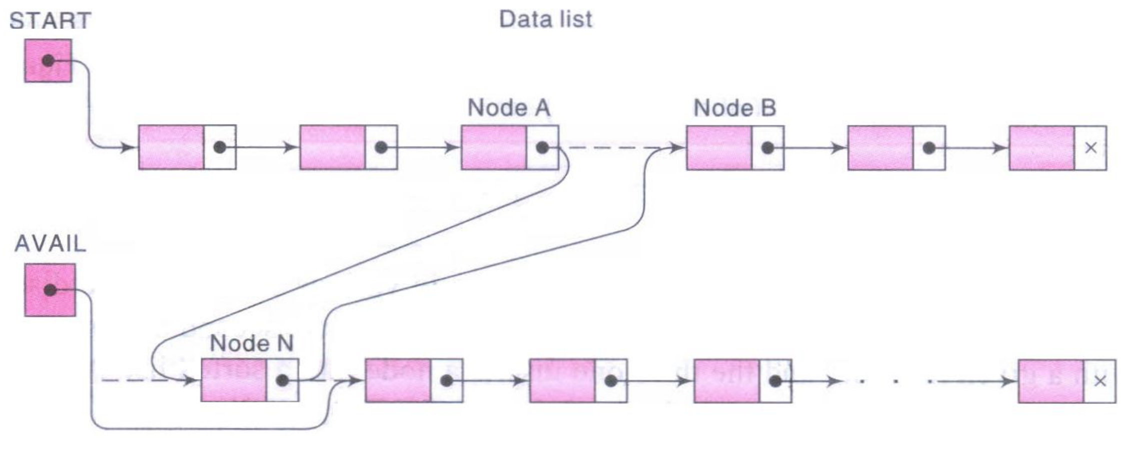


(a). Before Insertion



(b). After Insertion

The above figure does not take into account that the memory space for the new node N will come from the AVAIL list.
Specifically, for easier processing, the first node in the AVAIL list will be used for the new node N. Thus a more exact schematic diagram of such an insertion is that in below Fig.



Observe that three pointer fields are changed as follows:

1. The nextpointer field of node A now points to the new node N, to which AVAIL previously pointed.
2. AVAIL now points to the second node in the free pool, to which node N previously pointed.
3. The nextpointer field of node N now points to node B, to which node A previously pointed.

There are also two special cases.

1. If the new node N is the first node in the list, then START will point to N
2. If the new node N is the last node in the list, then N will contain the null pointer.

Insertion Algorithms

Algorithms which insert nodes into linked lists come up in various situations.

1. Inserts a node at the beginning of the list,
2. Inserts a node after the node with a given location
3. Inserts a node into a sorted list.

1. Inserting at the Beginning of a List

Inserting the node at the beginning of the list.

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

This algorithm inserts ITEM as the first node in the list.

```
1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.]
       Set NEW: = AVAIL and AVAIL: = AVAIL→LINK
3. Set NEW→INFO:= ITEM.        [Copies new data into new node]
4. Set NEW→LINK:= START.       [New node now points to original first node.]
5. Set START: = NEW.           [Changes START so it points to the new node.]
6. Exit.
```

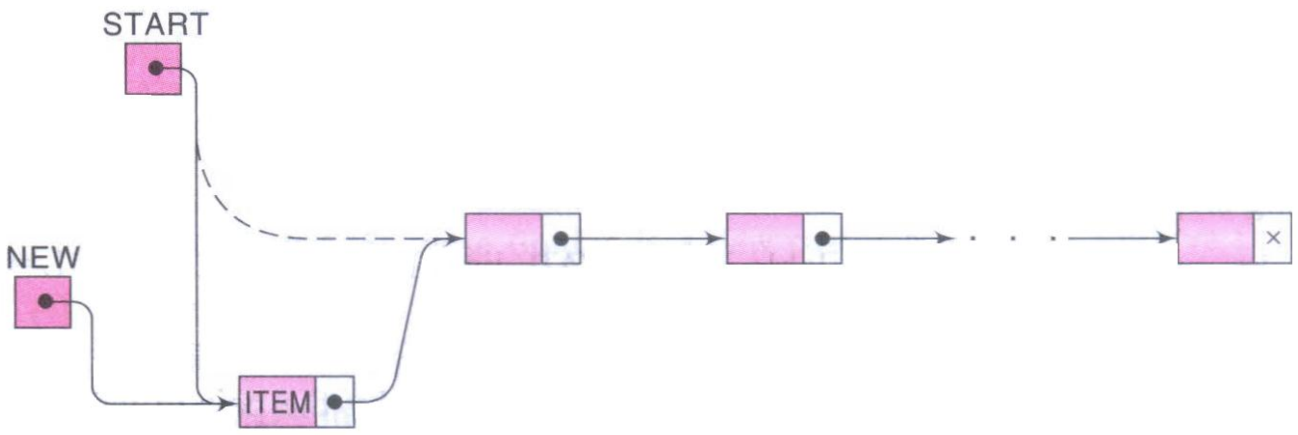


Fig: Inserting at the Beginning of a List

2. Inserting after a Given Node

Suppose the value of LOC is given where either LOC is the location of a node A in a linked LIST or LOC = NULL.

The following is an algorithm which inserts ITEM into LIST so that ITEM follows node A or, when LOC = NULL, so that ITEM is the first node.

Let N denote the new node. If LOC = NULL, then N is inserted as the first node in LIST. Otherwise, let node N point to node B by the assignment NEW→LINK:= LOC→LINK and let node A point to the new node N by the assignment LOC→LINK:= NEW

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

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

```
1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit
2. [Remove first node from AVAIL list.]
        Set NEW: = AVAIL and AVAIL: = AVAIL→LINK
3. Set NEW→INFO:= ITEM [Copies new data into new node]
4. If LOC = NULL, then: [Insert as first node]
        Set NEW→LINK:= START and START: = NEW.
   Else: [Insert after node with location LOC]
        Set NEW→LINK:= LOC→LINK and LOC→LINK:= NEW
   [End of If structure.]
5. Exit.
```

3. Inserting into a Sorted Linked List

- Suppose ITEM is to be inserted into a sorted linked LIST. Then ITEM must be inserted between nodes A and B so that

  INFO(A) < ITEM < INFO(B)
- The following is a procedure which finds the location LOC of node A, that is, which finds the location LOC of the last node in LIST whose value is less than ITEM.
- Traverse the list, using a pointer variable PTR and comparing ITEM with PTR→INFO at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in below Fig. Thus SAVE and PTR are updated by the assignments

  SAVE: = PTR and PTR: = PTR→LINK
- The traversing continues as long as PTR→INFO > ITEM, or in other words, the traversing stops as-soon as ITEM ≤ PTR→INFO. Then PTR points to node B, so SAVE will contain the location of the node A.

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

```
1. [List empty?] If START = NULL, then: Set LOC: = NULL, and Return.
2. [Special case?] If ITEM < START→INFO, then: Set LOC: = NULL, and Return.
3. Set SAVE: = START and PTR: = START→LINK. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5.      If ITEM < PTR→INFO, then:
                Set LOC: = SAVE, and Return.
        [End of If structure.]
6.      Set SAVE: = PTR and PTR: = PTR→LINK. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC: = SAVE.
8. Return.
```

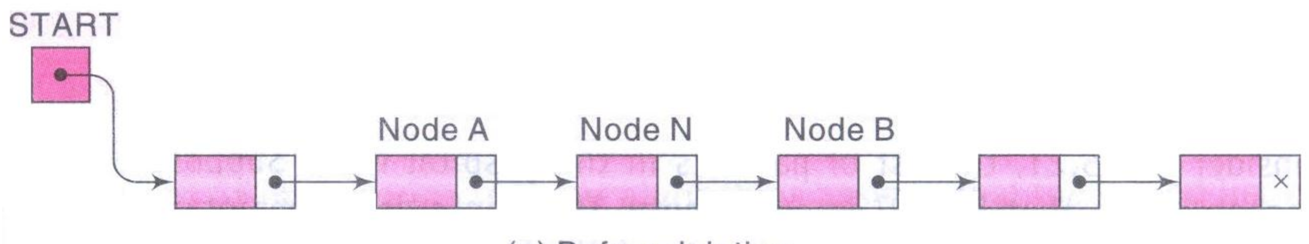
Below algorithm which inserts ITEM into a linked list. The simplicity of the algorithm comes from using the previous two procedures.

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

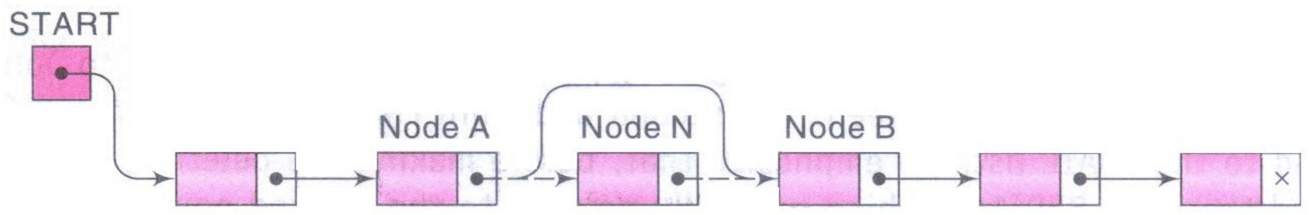
```
1. [Use Procedure to find the location of the node preceding ITEM.]
        Call FINDA (INFO, LINK, START, ITEM, LOC).
2. [Use Algorithm to insert ITEM after the node with location LOC.]
        Call INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM).
3. Exit.
```

3. Deletion into a Linked list

- Let LIST be a linked list with a node N between nodes A and B, as pictured in below Fig.(a). Suppose node N is to be deleted from the linked list. The schematic diagram of such a deletion appears in Fig.(b).
- The deletion occurs as soon as the nextpointer field of node A is changed so that it points to node B.
- Linked list is maintained in memory in the form
  LIST (INFO, LINK, START, AVAIL)

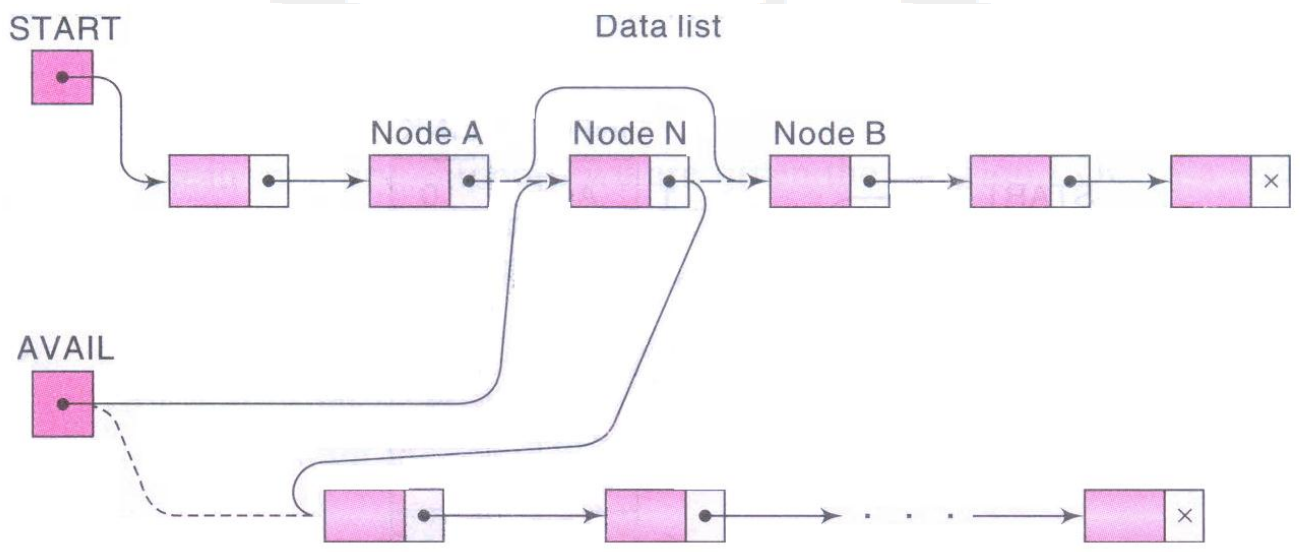


(a) Before deletion



(b) After deletion

The above figure does not take into account the fact that, when a node N is deleted from our list, immediately return its memory space to the AVAIL list. So for easier processing, it will be returned to the beginning of the AVAIL list. Thus a more exact schematic diagram of such a deletion is the one in below Fig.



Free storage list

Observe that three pointer fields are changed as follows:

1. The nextpointer field of node A now points to node B, where node N previously pointed.
2. The nextpointer field of N now points to the original first node in the free pool, where AVAIL previously pointed.
3. AVAIL now points to the deleted node N.

Deletion Algorithms

Deletion of nodes from linked lists come up in various situations.

1. Deletes the node following a given node
2. Deletes the node with a given ITEM of information.

All deletion algorithms will return the memory space of the deleted node N to the beginning of the AVAIL list.

Deleting the Node Following a Given Node

Let LIST be a linked list in memory. Suppose we are given the location LOC of a node N in LIST is given and location LOCP of the node preceding N or, when N is the first node, then LOCP = NULL is given.

The following algorithm deletes N from the list.

```
Algorithm: DEL (INFO, LINK, START, AVAIL, LOC, LOCP)
This algorithm deletes the node N with location LOC. LOCP is the location of the node which
precedes N or, when N is the first node, LOCP = NULL.
      1. If LOCP = NULL, then:
             Set START: = START→LINK.          [Deletes first node.]
         Else:
             Set LOCP→LINK:= LOC→LINK          [Deletes node N.]
        [End of If structure.]
      2. [Return deleted node to the AVAIL list.]
             Set LOC→LINK:= AVAIL and AVAIL: = LOC
      3. Exit.
```

Deleting the Node with a Given ITEM of Information

- Consider a given an ITEM of information and wants to delete from the LIST the first node N which contains ITEM. Then it is needed to know the location of the node preceding N. Accordingly, first finds the location LOC of the node N containing ITEM and the location LOCP of the node preceding node N.
- If N is the first node, then set LOCP = NULL, and if ITEM does not appear in LIST, then set LOC = NULL.
- Traverse the list, using a pointer variable PTR and comparing ITEM with PTR→INFO at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE. Thus SAVE and PTR are updated by the assignments SAVE:=PTR and PTR:= PTR→LINK
- The traversing continues as long as PTR→INFO ≠ ITEM, or in other words, the traversing stops as soon as ITEM = PTR→INFO. Then PTR contains the location LOC of node N and SAVE contains the location LOCP of the node preceding N

Procedure: FINDB (INFO, LINK, START, ITEM, LOC, LOCP)
This procedure finds the location LOC of the first node N which contains ITEM and the location LOCP of the node preceding N. If ITEM does not appear in the list, then the procedure sets LOC = NULL; and if ITEM appears in the first node, then it sets LOCP = NULL.

```
1. [List empty?] If START = NULL, then:
        Set LOC: = NULL and LOCP: = NULL, and Return.
   [End of If structure.]
2. [ITEM in first node?] If START→INFO = ITEM, then:
        Set LOC: = START and LOCP = NULL, and Return.
   [End of If structure.]
3. Set SAVE: = START and PTR: = START→LINK. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5. If PTR→INFO = ITEM, then:
        Set LOC: = PTR and LOCP: = SAVE, and Return.
   [End of If structure.]
6. Set SAVE: = PTR and PTR: = PTR→LINK. [Updates pointers.]
  [End of Step 4 loop.]
7. Set LOC: = NULL. [Search unsuccessful.]
8. Return.
```

DOUBLY LINKED LIST

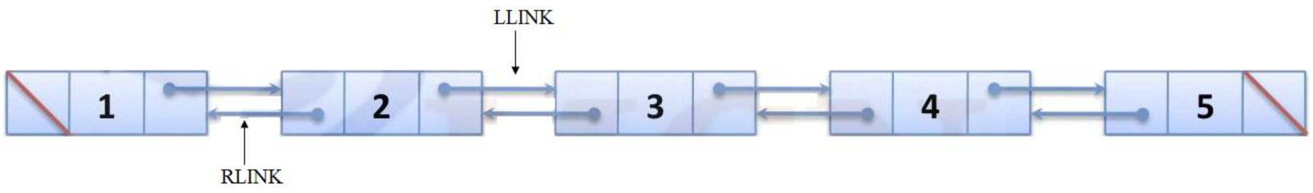
1. The difficulties with single linked lists is that, it is possible to traversal only in one direction, ie., direction of the links.
2. The only way to find the node that precedes p is to start at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a singly linked list. Hence the solution is to use doubly linked list

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

1. An information field INFO which contains the data of N
2. A pointer field LLINK (FORW) which contains the location of the next node in the list
3. A pointer field RLINK (BACK) which contains the location of the preceding node in the list

The declarations are:

```
typedef struct node *nodePointer;
typedef struct {
                nodePointer llink;
                element data;
                nodePointer rlink;
          } node;
```



Insertion into a doubly linked list

Insertion into a doubly linked list is fairly easy. Assume there are two nodes, node and newnode, node may be either a header node or an interior node in a list. The function dinsert performs the insertion operation in constant time.

```
void dinsert(nodePointer node, nodePointer newnode)
      {/* insert newnode to the right of node */
            newnode→llink = node;
            newnode→rlink = node→rlink;
            node→rlink→llink = newnode;
            node→rlink = newnode;
      }
```

Program: Insertion into a doubly linked circular list

Deletion from a doubly linked list

Deletion from a doubly linked list is equally easy. The function *ddelete* deletes the node deleted from the list pointed to by node.

To accomplish this deletion, we only need to change the link fields of the nodes that precede (deleted→llink→rlink) and follow (deleted→rlink→llink) the node we want to delete.

```
void ddelete(nodePointer node, nodePointer deleted)
        {/* delete from the doubly linked list */
                if (node == deleted)
                        printf("Deletion of header node not permitted.\n");
                else {
                        deleted→llink→rlink = deleted→rlink;
                        deleted→rlink→llink = deleted→llink;
                        free(deleted) ;
                      }
        }
```

Program: Deletion from a doubly linked circular list

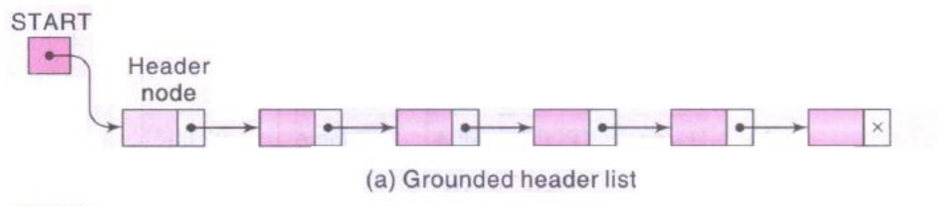
HEADER LINKED LISTS

A header linked list is a linked list which contains a special node, called the header node, at the beginning of the list.

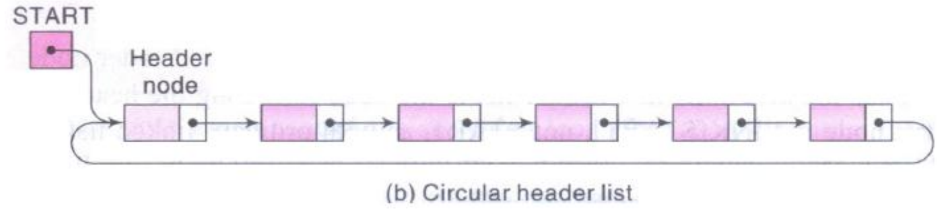
The following are two kinds of widely used header lists:

1. A grounded header list is a header list where the last node contains the null pointer.
2. A circular header list is a header list where the last node points back to the header node.

Below figure contains schematic diagrams of these header lists.



(a) Grounded header list



(b) Circular header list

Observe that the list pointer START always points to the header node.

- If START→LINK = NULL indicates that a grounded header list is empty
- If START→LINK = START indicates that a circular header list is empty.

The first node in a header list is the node following the header node, and the location of the first node is START→LINK, not START, as with ordinary linked lists.
Below algorithm, which uses a pointer variable PTR to traverse a circular header list

1. Begins with PTR = START→LINK (not PTR = START)
2. Ends when PTR = START (not PTR = NULL).

```
Algorithm: (Traversing a Circular Header List) Let LIST be a circular header list in memory.
This algorithm traverses LIST, applying an operation PROCESS to each node of LIST.
    1. Set PTR: = START→LINK. [Initializes the pointer PTR.]
    2. Repeat Steps 3 and 4 while PTR ≠ START:
    3.     Apply PROCESS to PTR→INFO.
    4.     Set PTR: = PTR→LINK. [PTR now points to the next node.]
       [End of Step 2 loop.]
    5. Exit.
```

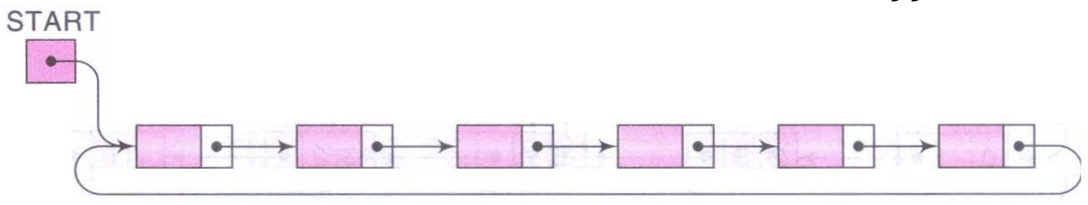
```
Algorithm: SRCHHL (INFO, LINK, START, ITEM, LOC)
LIST is a circular header list in memory. This algorithm finds the location LOC of the node
where ITEM first appears in LIST or sets LOC = NULL.
    1. Set PTR: = START→LINK
    2. Repeat while PTR→INFO [PTR] ≠ ITEM and PTR ≠ START:
           Set PTR: =PTR→LINK.    [PTR now points to the next node.]
       [End of loop.]
    3. If PTR→INFO = ITEM, then:
           Set LOC: = PTR.
       Else:
           Set LOC: = NULL.
       [End of If structure.]
    4. Exit.
```

The two properties of circular header lists:

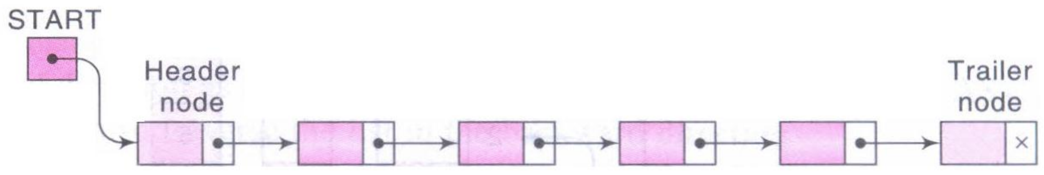
1. The null pointer is not used, and hence all pointers contain valid addresses.
2. Every (ordinary) node has a predecessor, so the first node may not require a special case.

There are two other variations of linked lists

1. A linked list whose last node points back to the first node instead of containing the null pointer, called a circular list
2. A linked list which contains both a special header node at the beginning of the list and a special trailer node at the end of the list



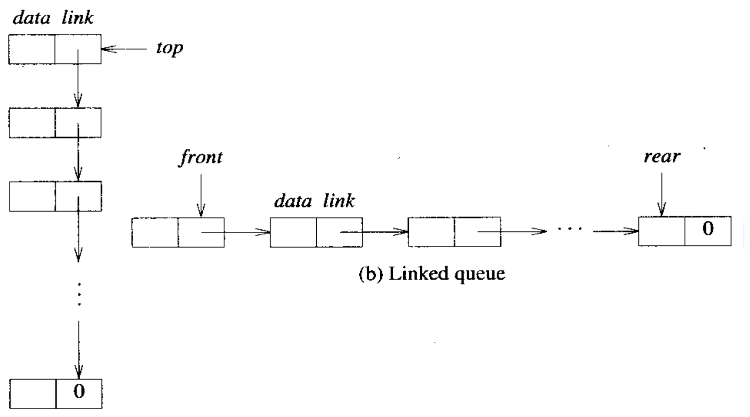
(a) Circular linked list



(b) Linked list with header and trailer nodes

LINKED STACKS AND QUEUES

The below figure shows stacks and queues using linked list. Nodes can easily add or delete a node from the top of the stack. Nodes can easily add a node to the rear of the queue and add or delete a node at the front



(b) Linked queue

(a) Linked stack

Linked Stack

The representation of n ≤ MAX_STACKS stacks, below is the declarations:

```
#define MAX_STACKS 10        /* maximum number of stacks */
typedef struct {
                    int key;
                    /* other fields */
              }element;
typedef struct stack *stackPointer;
typedef struct {
                    element data;
                    stackPointer link;
              } stack;
stackPointer top[MAX_STACKS];
```

The initial condition for the stacks is:

$$top[i] = NULL,\ 0 \leq i < MAX\_STACKS$$

The boundary condition is:

top [i] = NULL iff the i^{th} stack is empty

Functions push and pop add and delete items to/from a stack.

```
void push(int i, element item)
        {                                       /* add item to the ith stack */
                stackPointer temp;
                MALLOC(temp, sizeof(*temp));
                temp→data = item;
                temp→link = top[i];
                top[i] = temp;
        }
```

Program: Add to a linked stack

Function push creates a new node, temp, and places item in the data field and top in the link field. The variable top is then changed to point to temp. A typical function call to add an element to the ith stack would be push (i,item).

```
element pop(int i)
        {                                       /* remove top element from the ith stack */
                stackPointer temp = top[i];
                element item;
                if (! temp)
                        return stackEmpty();
                item = temp→data;
                top[i] = temp→link;
                free (temp) ;
                return item;
        }
```

Program: Delete from a linked stack

Function pop returns the top element and changes top to point to the address contained in its link field. The removed node is then returned to system memory. A typical function call to delete an element from the ith stack would be item = pop (i);

Linked Queue

The representation of m ≤ MAX_QUEUES queues, below is the declarations:

```
#define MAX-QUEUES 10 /* maximum number of queues */
typedef.struct queue *queuePointer;
typedef struct {
                element data;
                queuePointer link;
 } queue;
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

The initial condition for the queues is:

front[i] = NULL, $0 \le i <$ MAX_QUEUES

The boundary condition is:

front[i] = NULL iff the ith queue is empty

Functions addq and deleteq implement the add and delete operations for multiple queues.

```
void addq(i, item)
      {            /* add item to the rear of queue i */
            queuePointer temp;
            MALLOC(temp, sizeof(*temp));
            temp→data = item;
            temp→link = NULL;

            if (front[i])
                  rear[i] →link = temp;
            else
                  front[i] = temp;
            rear[i] = temp;
      }
```

Program: Add to the rear of a linked queue

Function addq is more complex than push because we must check for an empty queue. If the queue is empty, then change front to point to the new node; otherwise change rear's link field to point to the new node. In either case, we then change rear to point to the new node.

```
        element deleteq(int i)
                {                                       /* delete an element from queue i */
                        queuePointer temp = front[i];
                        element item;
                        if (! temp)
                                return queueEmpty();
                        item = temp→data;
                        front[i]= temp→link;
                        free (temp) ;
                        return item;
                }
```

Program: Delete from the front of a linked queue

Function deleteq is similar to pop since nodes are removing that is currently at the start of the list. Typical function calls would be addq (i, item); and item = deleteq (i);

<u>APPLICATIONS OF LINKED LISTS – POLYNOMIALS</u>

<u>Representation of the polynomial:</u>

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$

where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > ... > e_1 > e_0 \geq 0$.

Present each term as a node containing coefficient and exponent fields, as well as a pointer to the next term.

<u>Assuming that the coefficients are integers, the type declarations are:</u>

```
        typedef struct polyNode *polyPointer;
        typedef struct {
                                int coef;
                                int expon;
                                polyPointer link;
                        } polyNode;
        polyPointer a,b;
```

coef	expon	link

Figure shows how we would store the polynomials

$$a = 3x^{14} + 2x^{8} + 1$$

and

$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

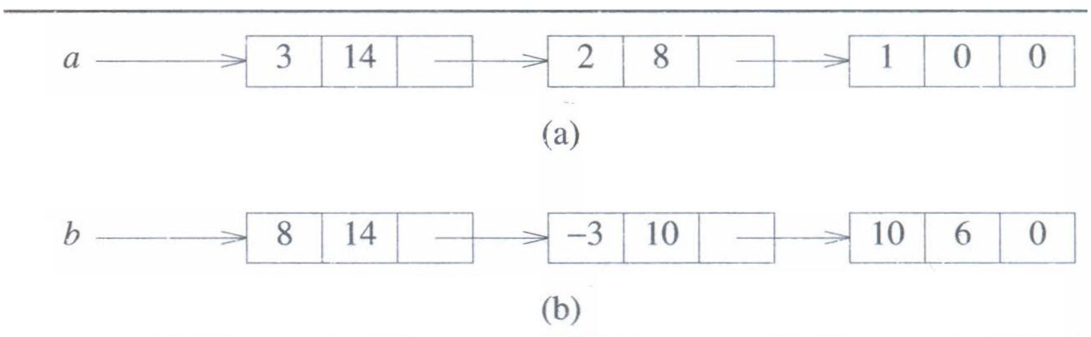


Figure: Representation of $3x^{14}+2x^{8}+1$ and $8x^{14}-3x^{10}+10x^{6}$

<u>Adding Polynomials</u>

To add two polynomials, examine their terms starting at the nodes pointed to by ***a*** and ***b***.

- If the exponents of the two terms are equal, then add the two coefficients and create a new term for the result, and also move the pointers to the next nodes in ***a*** and ***b***.
- If the exponent of the current term in ***a*** is less than the exponent of the current term in ***b***, then create a duplicate term of ***b***, attach this term to the result, called ***c***, and advance the pointer to the next term in ***b***.
- If the exponent of the current term in ***b*** is less than the exponent of the current term in ***a***, then create a duplicate term of ***a***, attach this term to the result, called ***c***, and advance the pointer to the next term in ***a***

Below figure illustrates this process for the polynomials addition.

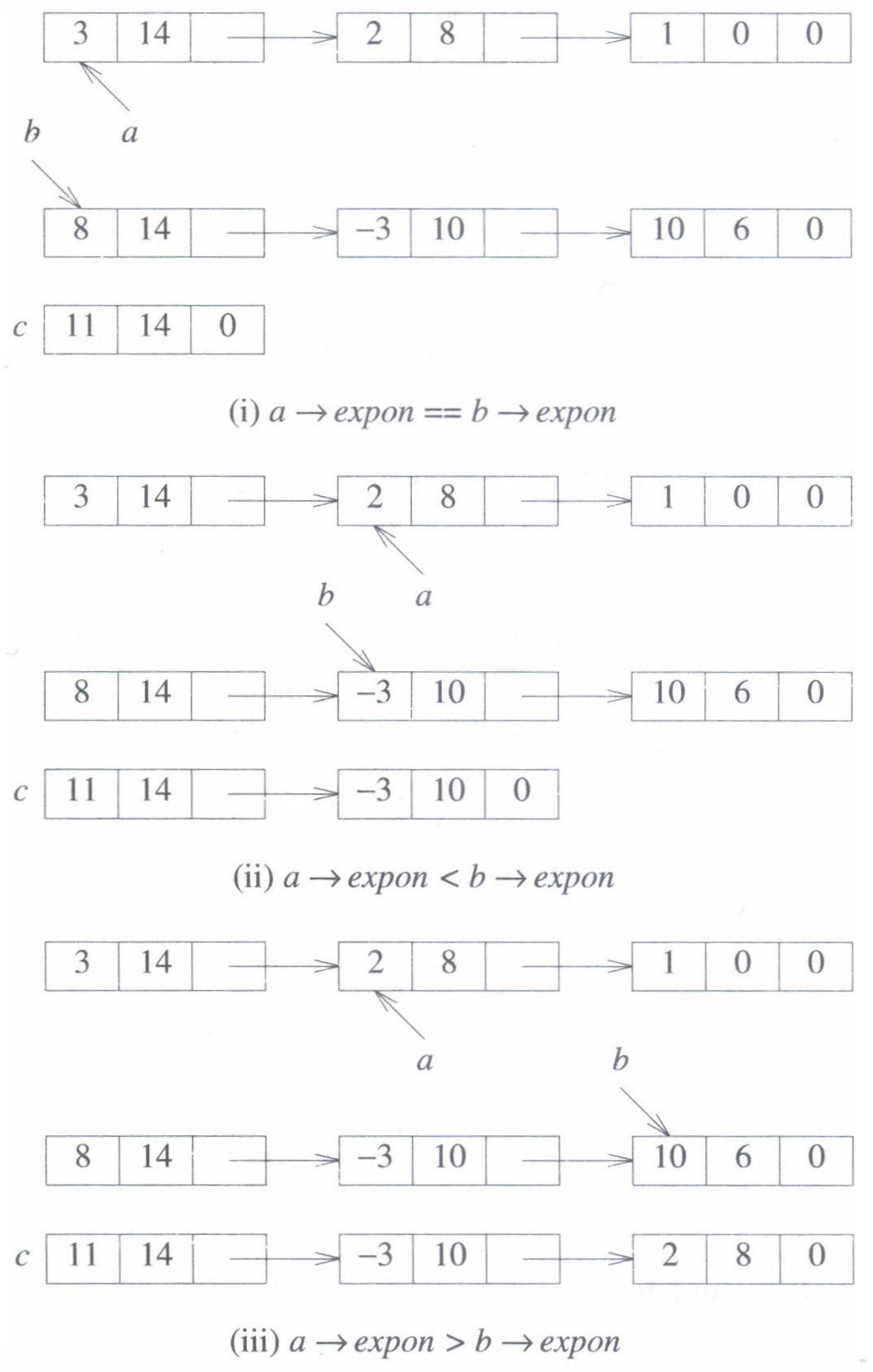

a
b
c
(i) $a \rightarrow expon == b \rightarrow expon$
(ii) $a \rightarrow expon < b \rightarrow expon$
(iii) $a \rightarrow expon > b \rightarrow expon$

Figure: Generating the first three terms of $c = a + b$

The complete addition algorithm is specified by padd( )

```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and b */
  polyPointer c, rear, temp;
  int sum;
  MALLOC(rear, sizeof(*rear));
  c = rear;
  while (a && b)
    switch (COMPARE(a→expon,b→expon)) {
      case -1: /* a→expon < b→expon */
           attach(b→coef,b→expon,&rear);
           b = b→link;
           break;
      case 0: /* a→expon = b→expon */
           sum = a→coef + b→coef;
           if (sum) attach(sum,a→expon,&rear);
           a = a→link;  b = b→link; break;
      case 1: /* a→expon > b→expon */
           attach(a→coef,a→expon,&rear);
           a = a→link;
    }
  /* copy rest of list a and then list b */
  for (; a; a = a→link) attach(a→coef,a→expon,&rear);
  for (; b; b = b→link) attach(b→coef,b→expon,&rear);
  rear→link = NULL;
  /* delete extra initial node */
  temp = c; c = c→link;  free(temp);
  return c;
}
```

Program : Add two polynomials

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{/* create a new node with coef = coefficient and expon =
    exponent, attach it to the node pointed to by ptr.
    ptr is updated to point to this new node */
  polyPointer temp;
  MALLOC(temp, sizeof(*temp));
  temp→coef = coefficient;
  temp→expon = exponent;
  (*ptr)→link = temp;
  *ptr = temp;
}
```

Program : Attach a node to the end of a list

Analysis of padd:

To determine the computing time of padd, first determine which operations contribute to the cost. For this algorithm, there are three cost measures:

(l) Coefficient additions

(2) Exponent comparisons

(3) Creation of new nodes for c

The maximum number of executions of any statement in padd is bounded above by m + n. Therefore, the computing time is O(m+n). This means that if we implement and run the algorithm on a computer, the time it takes will be $C_1m + C_2n + C_3$, where C1, C2, C3 are constants. Since any algorithm that adds two polynomials must look at each nonzero term at least once, padd is optimal to within a constant factor.

SPARSE MATRIX REPRESENTATION

A linked list representation for sparse matrices.

In data representation, each column of a sparse matrix is represented as a circularly linked list with a header node. A similar representation is used for each row of a sparse matrix.
Each node has a tag field, which is used to distinguish between header nodes and entry nodes.

Header Node:

- Each header node has three fields: down, right, and next as shown in figure (a).
- The down field is used to link into a column list and the right field to link into a row list.
- The next field links the header nodes together.
- The header node for row i is also the header node for column i, and the total number of header nodes is max {number of rows, number of columns}.

Element node:

- Each element node has five fields in addition in addition to the tag field: row, col, down, right, value as shown in figure (b).
- The down field is used to link to the next nonzero term in the same column and the right field to link to the next nonzero term in the same row. Thus, if $a_{ij} \neq 0$, there is a node with tag field = entry, value = a_{ij}, row = i, and col = j as shown in figure (c).
- We link this node into the circular linked lists for row i and column j. Hence, it is simultaneously linked into two different lists.

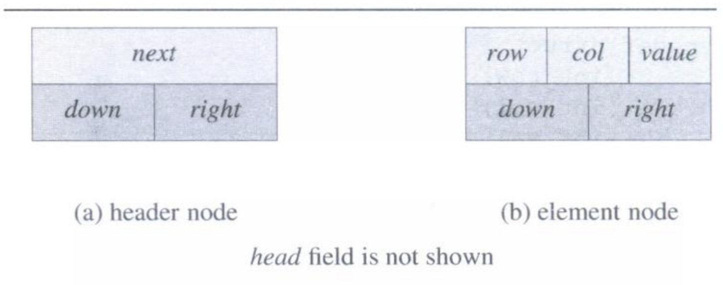


Figure: Node structure for sparse matrices

Consider the sparse matrix, as shown in below figure (2).

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

Figure (2): 4 × 4 sparse matrix *a*

Figure (3) shows the linked representation of this matrix. Although we have not shown the value of the tag fields, we can easily determine these values from the node structure.
For each nonzero term of a, have one entry node that is in exactly one row list and one column list. The header nodes are marked HO-H3. As the figure shows, we use the right field of the header node list header to link into the list of header nodes.

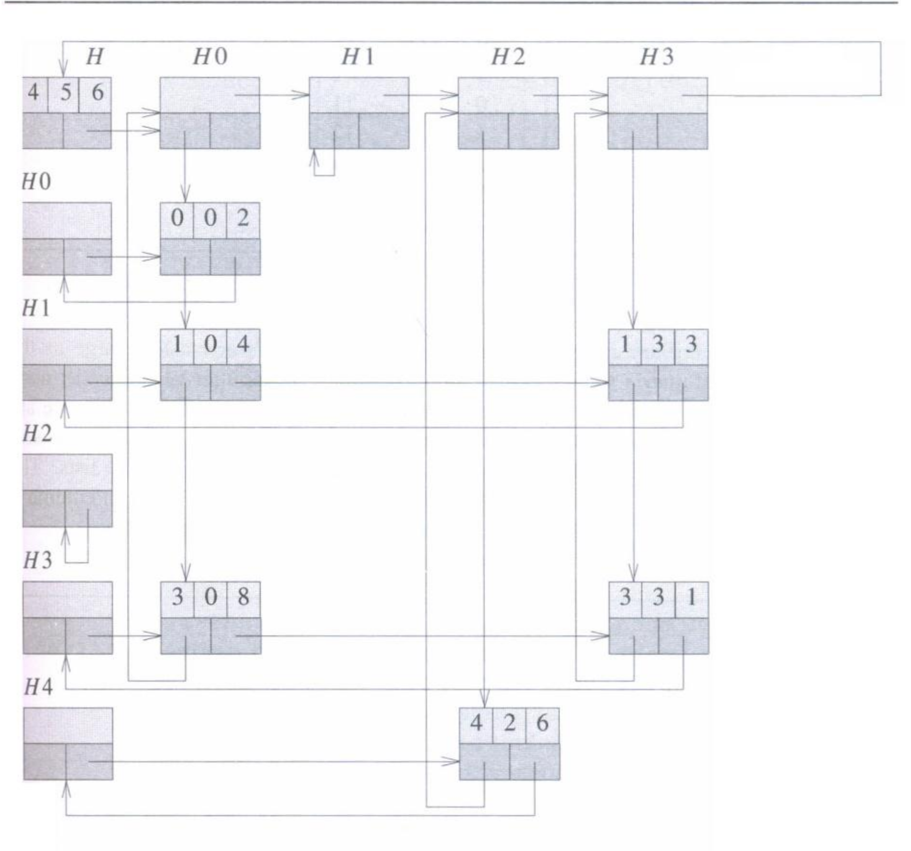


Figure (3) : Linked representation of the sparse matrix of Figure (2) (the *head* field of a node is not shown)

To represent a *numRows x numCols* matrix with *numTerms* nonzero terms, then we need max *{numRows, numCols}* + *numTerms* + *1* nodes. While each node may require several words of memory, the total storage will be less than *numRows x numCols* when *numTerms* is sufficiently small.

There are two different types of nodes in representation, so unions are used to create the appropriate data structure. The C declarations are as follows:

```
#define MAX-SIZE 50 /*size of largest matrix*/
typedef enum {head, entry} tagfield;
typedef struet matrixNode *matrixPointer;

typedef struet {
                   int row;
                   int eol;
                   int value;
               } entryNode;

typedef struet {
                   matrixPointer down;
                   matrixPointer right;
                   tagfield tag;

                   union {
                            matrixPointer next;
                            entryNode entry;
                           } u;
                } matrixNode;
matrixPointer hdnode[MAX-SIZE];
```