```
    q = q->right;
    while (p != q)              /* As long as nodes "p" and "q" are not the same */
    {
      int t;
      t = p->data;
      p->data = q->data;
      q->data = t;
      p = p->right;
      if  (p == q)      /* Takes care if the list contains even number of nodes. */
            break;
      q = q -> left;
    }
}
```

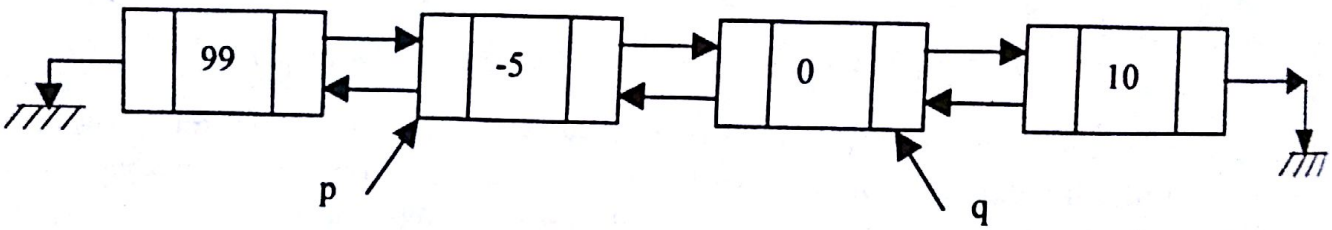


Figure 3.16 (b) After the first iteration of the second while loop.

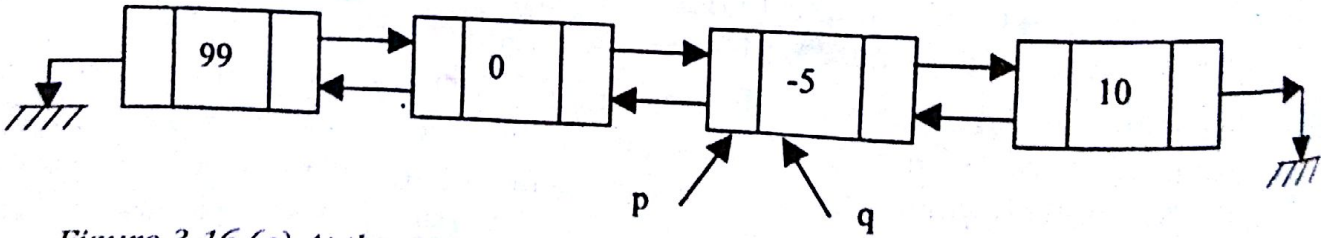


Figure 3.16 (c) At the end of the second iteration of the second while loop.

3.4 Circular lists

In some applications, it may be more convenient if the link field of the last node in a singly connected linked list points back to the first node of the list. Such a list is called a circular linked list as shown in Figure 3.17.

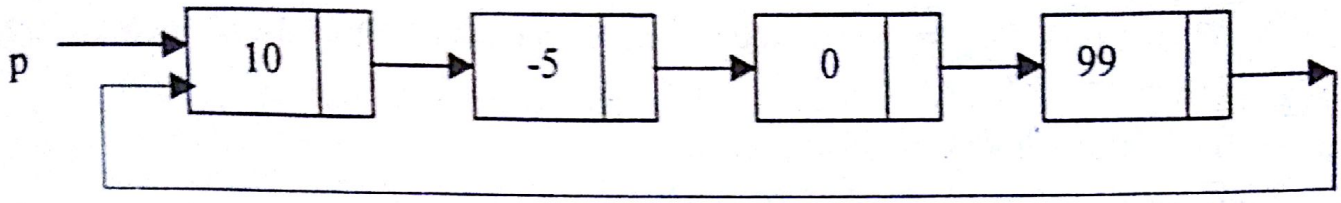


Figure 3.17 Example of a circular linked list.

Algorithm for traversing a circular list is slightly different than the same algorithm for a singly connected linked list because "NULL" is not encountered as the link field for any node in a circular list. For traversing a circular list, one needs to save the starting pointer and traverse as long as the link field of a node becomes equal to the saved pointer. Sometimes it is advantageous to identify a circular list by the pointer to the last node in the list as shown in Figure 3.18.

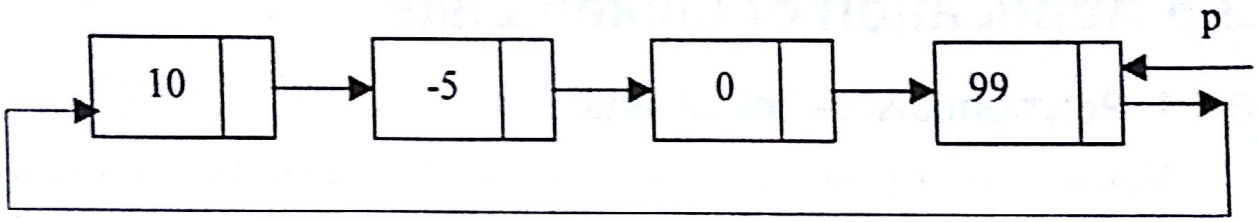


Figure. 3.18 Example of a circular linked list identified by a pointer to the tail of the list.

Particularly, insertion of a node at the start or end of a circular list identified by a pointer to the last node of the list takes a constant amount of time irrespective of the number of nodes in the list. Similar benefit is also enjoyed when two circular lists are concatenated. The algorithms for the problems mentioned are left as exercises.

Doubly connected linked lists may also be circular. Sometimes a head node is introduced to distinguish between an empty and a non-empty list. The head node is like any other node except that the data field of the head node is not relevant. The head node provides symmetry to the nodes in a circular list. Consider Figure 3.19 where a head node is introduced to a circular doubly linked list.

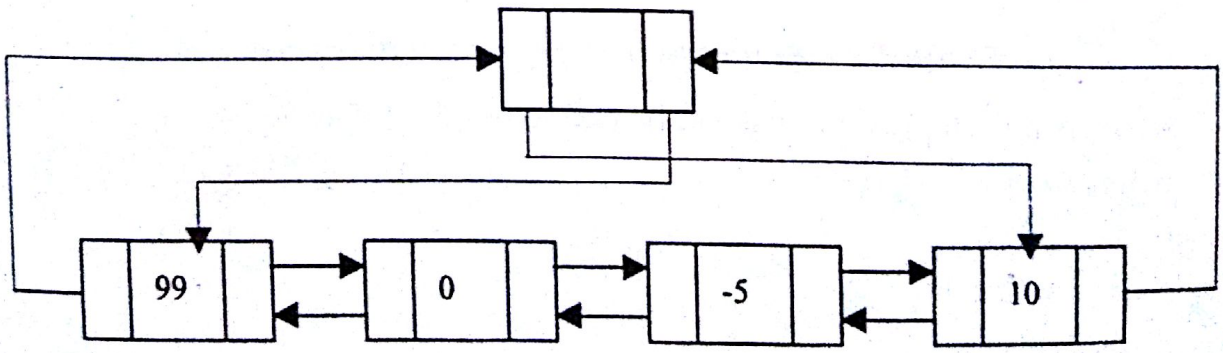


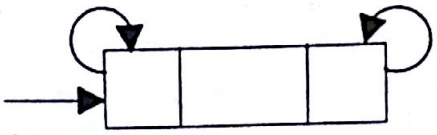
Figure 3.19 Doubly linked circular list with a head node.

Now, suppose that "p" points to any node in the list in Figure 3.19. The following observations are true about "p".

right link of left link of "p" is "p".

left link of right link of "p" is "p".

Without the head node, the above observations are not true for the nodes at either extremes. With the inclusion of a head node, an empty list is not really empty as it definitely contains a head node as shown in the following.



3.5 Application of Linked Lists

3.5.1 Polynomials as linked lists

Representation of polynomials has already been discussed in the previous chapter. It is mentioned that polynomials can be thought of as an ordered list of non-zero terms. Each non-zero term is a 2-tuple containing two pieces of information: (i) the exponent part and (ii) the coefficient part.

Thus, the polynomial $2x^4-5x^2+6$ is represented as:

((0,6), (2, -5), (4,2))

The first tuple indicates the term $6.x^0$, i.e., 6; the second tuple indicates $-5.x^2$ and the last tuple denotes $2x^4$. The tuples are arranged in increasing order of their exponent part. That means the first tuple contains the non-zero term with the least power of "x" and the last tuple contains the non-zero term with the highest power of "x". The linked list representation of the said polynomial is shown in Figure 3.20.

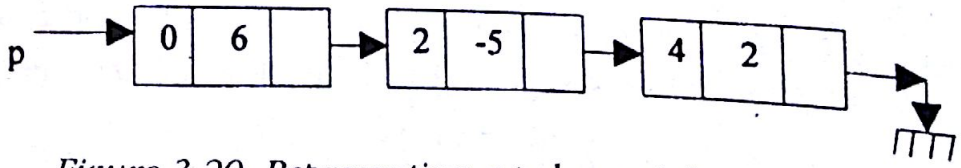


Figure 3.20 Representing a polynomial as a linked list.

Each term in the polynomial can be defined as the following struct.

```
typedef struct polynode
{
    int coefficient;
    int power;
    struct polynode *nextnode;
```

} polynode;

The algorithm to add two polynomials denoted by "p" and "q" is similar to that described for array representation of polynomials. Both the lists are scanned term by term. Whenever the "powers" of the current terms of "p" and "q" are same, their coefficients are added to obtain the coefficient of the term of the new polynomial. The exponent part of this term is the same as that of the current term of either of "p" or "q". Subsequently, both pointers pointing to the current nodes are advanced to point to the next node. If the "power" part of one polynomial, say "p", is less then a new node is created which is a copy of the term of "p" and this new node is added to the resultant list. When one of the lists of "p" and "q" is exhausted then the remaining nodes of the other list is simply copied to the resultant list.

```
void polyadd (polynode *poly1, polynode *poly2, polynode **polysum)
{
    polynode *p, *q, *r, *s;
    int newcoefficient, newpower;
    p = poly1;
    q = poly2;
    r = NULL;
    while((p!= NULL) && (q != NULL))
    {
        if (p->power == q->power)
        {
            newcoefficient = p->coefficient + q->coefficient;
            newpower = p->power;
            p=p->nextnode;
            q = q->nextnode;
        }
        else if (p->power > q->power)
        {
            newcoefficient = q->coefficient;
            newpower = q->power;
            q = q->nextnode;
        }
        else
        {
            newcoefficient = p->coefficient;
```

```
                newpower = p->power;
                p = p->nextnode;
        }
        if (newcoefficient != 0)
        {
                /* Insert this term to the resultant list */
                s = (polynode *) malloc (sizeof (polynode));
                s->coefficient = newcoefficient;
                s->power = newpower;
                if (r == null)
                        *polysum = s;
                else
                        r->nextnode = s;
                r = s;
                s->nextnode = NULL;
        }
}/* while ((p!= NULL) && (q != null)) */
if (p == null)
{
        while (q != NULL)
        {
                s = (polynode *) malloc (sizeof (polynode));
                s->coefficient = q-> coefficient;
                s->power = q->power;
                if (r == NULL)
                        *polysum = s;
                else
                        r->nextnode = s;
                r = s;
                s->nextnode = NULL;
                q = q->nextnode;
        }
} else
```

```
    {
        while (p != NULL)
        {
            s = (polynode *) malloc (sizeof (polynode));
            s->coefficient = p->coefficient;
            s->power = p->power;
            if (r == NULL)
                *polysum = s;
            else
                r->nextnode = s;
            r = s;
            s->nextnode = NULL;
            p = p->nextnode;
        }
    }
}
```

3.5.2 Josephus problem

There is an interesting programming problem known in the literature as Josephus problem. The problem can be stated as follows. Suppose there are "n" children standing in a circle playing a game. Arbitrarily one of them is designated as number 1 and others are numbered in a clockwise fashion starting from the child with number 1. Then they choose a lucky number, say "m". Immediately after, they start counting from the child designated number 1 and counting proceeds in a clockwise manner until the m-th child is identified. Then, the m-th child is eliminated from the circle and as a result the circle shrinks. Counting for the next round begins from the child next to the eliminated one and proceeds until the m-th child is identified. This child is then eliminated and the circle shrinks further. Thus, in each round of counting, one child gets eliminated. Consequently, after some round of counting only one child will be left and this child is declared the winner of the game.

It is required to write a program to identify the winner in the game described above. Inputs to this program are two integers "n" and "m" and the output is the integer identifying the winner. From a programming point of view, the problem can be tackled easily if we represent the children in a circular linked list. There will be no special head node but a pointer to the node representing the child from where the counting begins in each round. Each node for a child contains an integer specifying its number in the original circle and a pointer to the next child in the

current circle. The major operation involved in this algorithm is deletion from a circular linked list. The structure of the "child" is shown below.

```
typedef  struct child
{
   int originalPosition;
   struct child *nextChild;
}child;
```

Algorithm for identifying the winner is shown next.

```
int findwinner (int n, int m)
{
    create the circular linked list with n children.
    let p be a pointer to the beginning of the list;
    while (the list contains more then one element)
    {
            Set a counter to zero;
            Go to the next node and increment counter as long as counter
            is less than m;
            Delete the current node;
    }
    Get the original position of the only node present and
    return the original position;
}
```

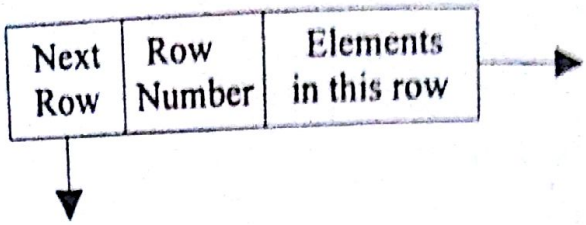
The " C" function of the above algorithm is described in the following:

```
int findwinner (int n, int m)
{
    int i, counter;
    child *p, *q, *r;
/* Initializing the circular linked list */
    p = (child*) malloc (sizeof (child));
    p->originalPosition = 1;
    p->nextChild = p;
    q = p;
    for (i = 2; i <= n; i ++)
```

```
{
        r = (child *)malloc (sizeof (child));
        r->originalPosition = i;
        r->nextChild = p;
        q->nextChild = r;
        q = r;
}
/* Starting the game */
while (p->nextChild != p)
{
        q = p;
        r = q;
        for (counter = 1; counter < m; counter ++)
        {
                r = q;
                q = q->nextChild;
        }
        r->nextChild = q->nextChild;
        p = r->nextChild;
        q->nextChild = NULL;
        free (q);
        q = NULL;
  }
  return (p->originalPosition);
}
```

3.5.3 Sparse Matrix

Representation of a sparse matrix has already been discussed in the previous chapter. In this chapter, a linked list representation of sparse matrices is considered. As before, an element of a sparse matrix is assumed to consist of three integers: (i) its row number, i, (ii) its column number, j and (iii) its value, val. Unlike array representation, the matrix is not thought of as a chain of such triplets. Instead, consider "head" nodes for each row and each column pointing to the elements in a particular row or column. A head node for a row contains three parts as shown below:



"Row number" indicates the row to which this "head" node is pointing to by the component "Element". This "head" node also points to another "head" node for the next row. Thus a "row" is defined as follows:

```
typedef struct row
{
    int row_number
    Element *right;
    struct row *nextRow;
} row;
```

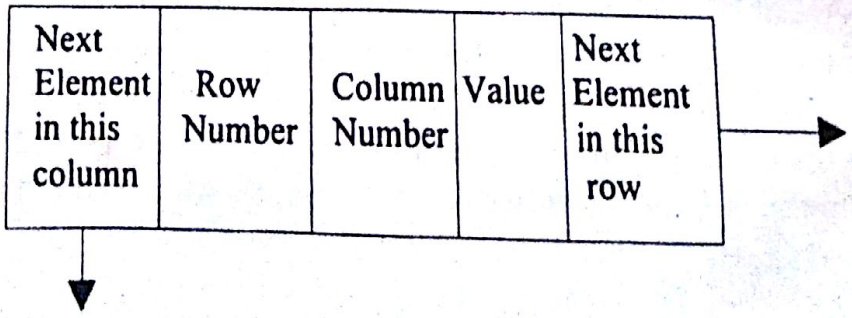
Similar a "column" consists of its column number, a pointer pointing to the next column and a pointer to elements in that column A "column" is defined as:

```
typedef struct column
{
        int column_number;
        Element *down;
        struct column *nextColumn;
}column;
```

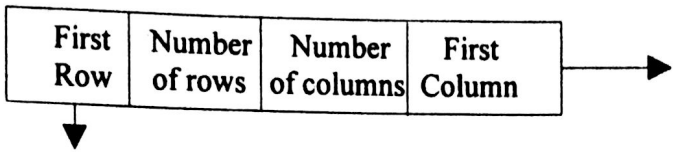
An element is defined as:

```
typedef struct Element
{
        int i, j, val;
        struct Element *right;
        struct Element *down;
}Element;
```

The structure of one element is pictorially illustrated in the following figure.



A sparse matrix, therefore, can be defined as a node having two pointers one pointing to the list of rows and the other pointing to the list of columns. In addition, this node should contain two integers specifying the number of rows and the number of columns. Thus, such a node can be depicted as follows:



The type "spmat" may be defined as the following structure:

```
typedef struct spmat
{
        row * firstrow;
        column * firstcolumn;
        int no_of_rows, no_of_columns;
} spmat;
```

The initial configuration of a 3 X 3 matrix is shown in Figure 3.21. If A be a 3 X 3 matrix as given below:

$$A = \begin{vmatrix} 0 & 5 & 0 \\ 9 & 0 & 7 \\ 1 & 2 & 3 \end{vmatrix}$$

Then its list representation is shown in Figure 3.22.

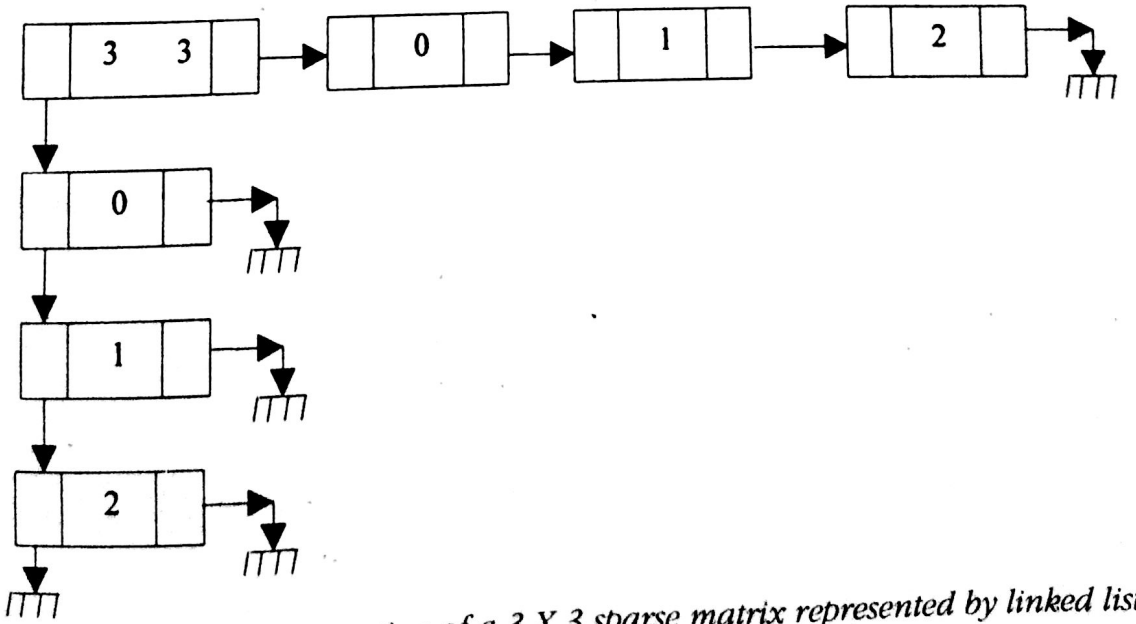


Figure 3.21 Initial configuration of a 3 X 3 sparse matrix represented by linked lists.

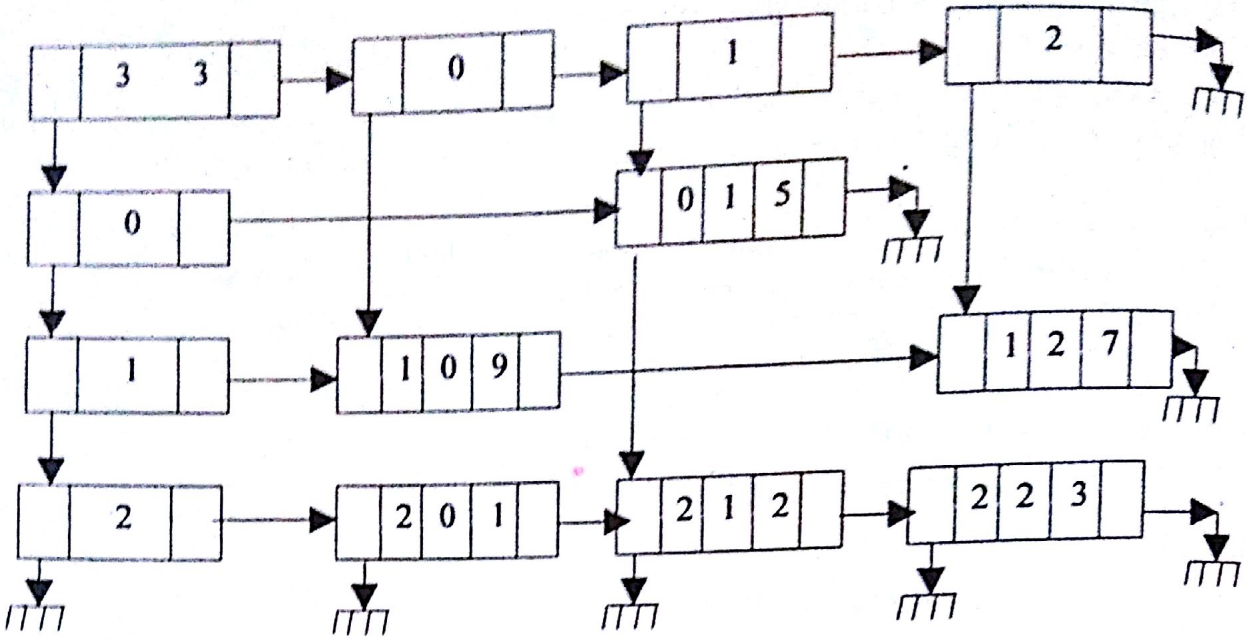


Figure 3.22 The linked list representation of the sparse matrix "A".

Observe that the elements in each row are arranged in increasing order of their column numbers. Similarly, the elements in each column are arranged in increasing order of their row numbers.

In order to create a linked list version of a sparse matrix, the elements of the matrix represented as triplets of (row, column, value) must be inserted repeatedly to a partially constructed sparse matrix by linked list. In the following, a function "insert" is written which inserts an element of a sparse matrix into a partially constructed sparse matrix pointed to by "a". It is assumed that the matrix (*a) is already initialized.

```
void insert (spmat *a, int val, int rowno, int colno)
{
   int i;
   Element *p, *q, *s;
   row *r; column *c;        /* First allocate memory for Element and fill up its fields
*/
   p = (Element *) malloc (sizeof (Element));
   p->val = val;
   p->i = rowno;
   p->j = colno;
   /* Find the row pointer, "r", where this element has to be added */
   r = a->firstrow;
   for (i = 0; i < rowno; i ++)
```

```
    r = r->nextRow;
    /* Now add this element in a proper position in this row, "r" */
q = r->right;
if (q == NULL)
/* This row, "r", may not have any element as yet */
{
    r->right = p;
    p->right = NULL;
}
else
{
/* Elements in  a row are sorted on its column number. The current element p is, therefore, properly inserted in the list so that the list remains sorted after the insertion. */
    while ((q != NULL) && (q->j < colno))
    {
            s = q;
            q = q->right;
    }
    s->right = p;
    p->right = q;
}
/* Element p is to be added to a column as well. */
    c = a->firstcolumn;
    for (i = 0; i < colno; i++)
            c = c->nextColumn;
    q = c->down;
    if (q == NULL)
    /* This column, q, may be empty. */
    {
            c->down = p;
            p->down = NULL;
    }
 else
```

```
{  /* Elements in a column are sorted on its row number. The current element p is
to be properly inserted so that the list remains sorted after insertion. */
        while ((q!= NULL) && (q->i < rowno))
        {
                s = q;
                q = q->down;
        }
        s->down = p;
        p->down = q;
    }
}
```

Next, consider adding two sparse matrices pointed to by "a" and "b" respectively to produce a third matrix pointed to by "c". First, the resultant matrix pointed to by "c" is to be initialized. Then, the elements in matrices represented by "a" and "b" are taken up row by row. If the column numbers of the elements in "a" and "b" match then the values of these elements are added and a new element is inserted in the matrix "c". If the column numbers of the elements concerned are not the same then the element with lesser column number is inserted to "c". Note the striking similarity of this routine with the function to add two polynomials.

```
spmat* addmatrix (spmat *a, spmat *b)
{
  int i, v,j;
  row *r1, *r2;
  Element *p, *q;
  spmat *c;
  initialize(&c, a-> no_of_rows, a->no_of_columns);
 r1 = a->firstrow;
 r2 = b->firstrow;
 for (i = 0; i < c->no_of_rows; i ++)
 {
        p = r1->right;
        q = r2->right;
        while ((p!= NULL) && (q != NULL))
        {
                if (p->j == q->j)
                {
```

```
                insert (c, p->val+q->val, i, p->j);
                p = p->right;
                q = q->right;
            }
            else if (p->j < q->j)
            {
                insert (c, p->val, i, p->j);
                p = p->right;
            }
            else
            {
                insert (c, q->val, i, q->j);
                q = q->right;
            }
        }
        /* Copy the remaining elements from the row which is not exhausted */
        while (p != NULL)
        {
            insert (c, p->val, i, p->j);
            p = p->right;
        }
        while (q != NULL)
        {
            insert (c, q->val, i, q->j);
            q = q->right;
        }
        r1=r1->nextRow;
        r2=r2->nextRow;
    }
}
```

3.6 Array Representation of linked lists

It has been repeatedly emphasized that linked lists may be implemented even without using pointers. In this section, representation of linked lists using arrays will be explained. For simplicity and ease of explanation, linked lists of integers