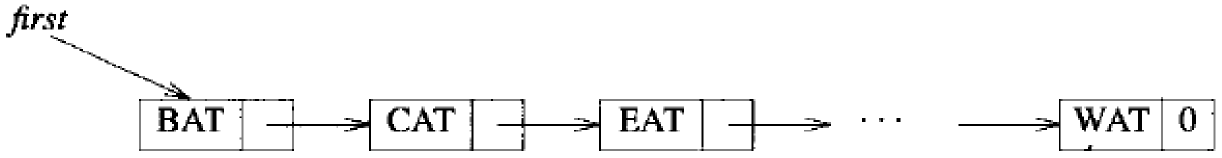
MODULE 3: LINKED LIST

DEFINITION

A **linked list** is a dynamic data structure where each element (called a node) is made up of two items - the data and a reference (or pointer) which points to the next node. A **linked list** is a collection of nodes where each node is connected to the next node through a pointer.



Usual way to draw a linked list

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 link 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*.

A pointer variable called ***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 ***first***.

REPRESENTATION OF LINKED LISTS IN MEMORY

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

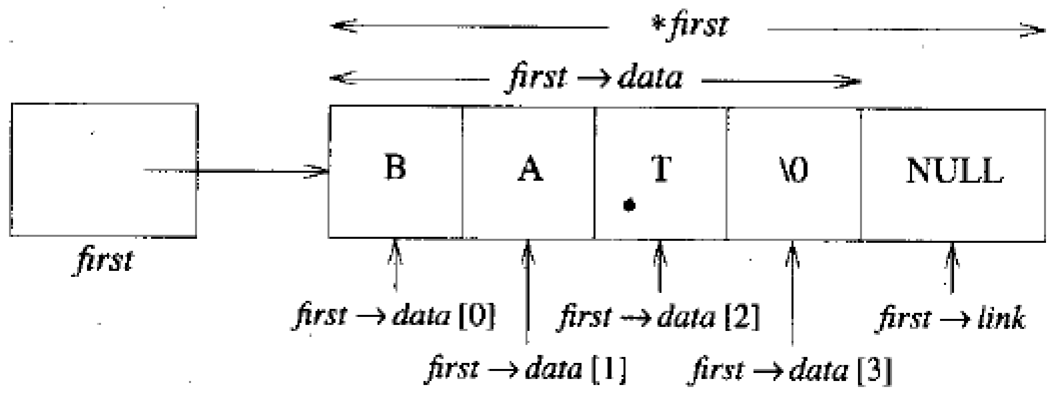
```
listPointer first = NULL;
```

To create a New Node

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

To place the data into Node

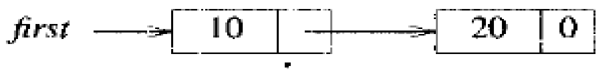
```
strcpy(first→ data,"BAT");
first→ link = NULL
```



LINKED LIST OPERATIONS

1. ***Creating two node lists*:**

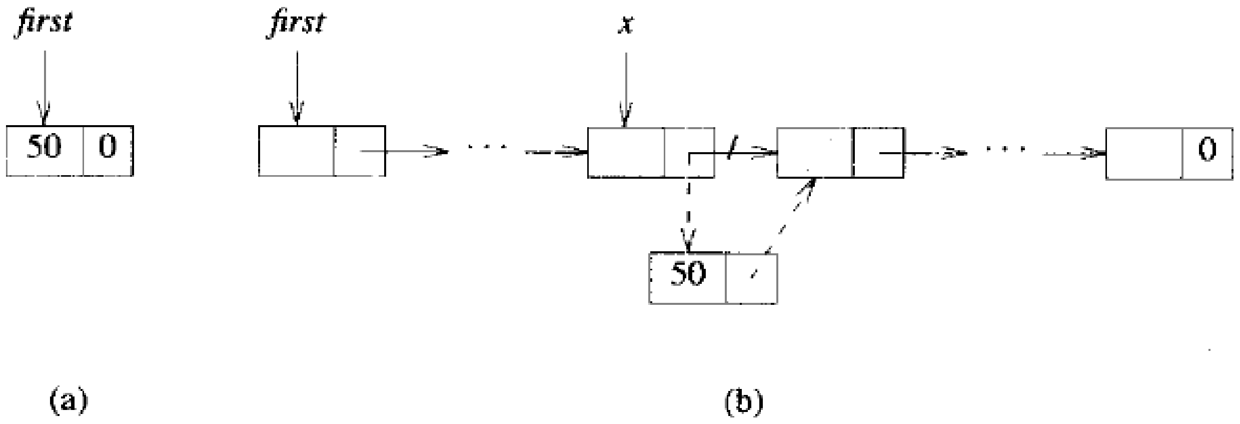
```
listPointer create2()
{/* create a linked list with two nodes */
   listPointer first, second;
   MALLOC(first, sizeof(*first));
   MALLOC(second, sizeof(*second));
   second→link = NULL;
   second→data = 20;
   first→data = 10;
   first→link = second;
   return first;
}
```



A two-node list

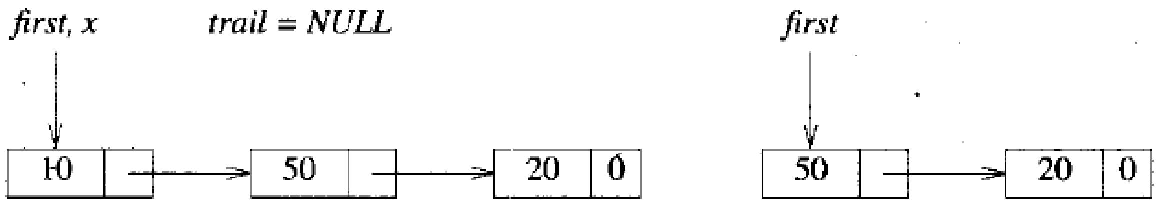
2. ***Insertion into front of list***

```
void insert(listPointer *first, listPointer x)
{/* insert a new node with data = 50 into the chain
    first after node x */
   listPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→data = 50;
   if (*first) {
      temp→link = x→link;
      x→link = temp;
   }
   else {
      temp→link = NULL;
      *first = temp;
   }
}
```



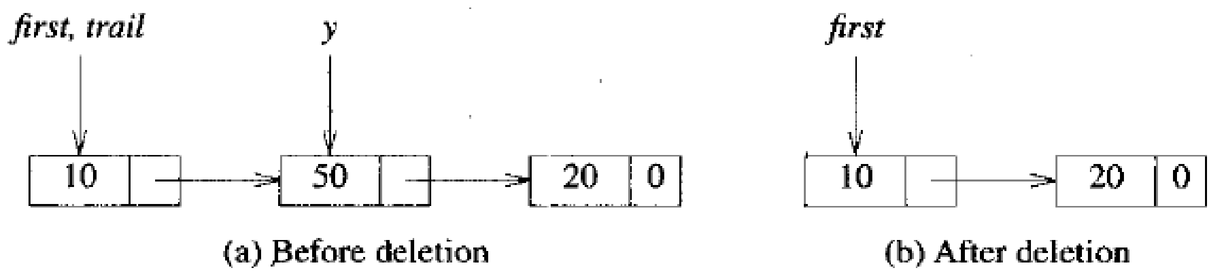
Inserting into an empty and nonempty list

3. ***Deletion from the list*:**



(a) Before deletion (b) After deletion

List before and after the function call *delete(&first, NULL, first);*



(a) Before deletion (b) After deletion

List after the function call *delete(&first, y, y→link);*

```
void delete(listPointer *first, listPointer trail,
                                listPointer x)
{/* delete x from the list, trail is the preceding node
    and *first is the front of the list */
   if (trail)
     trail→link = x→link;
   else
     *first = (*first)→link;
   free(x);
}
```

4. **Traversing a linked List**

```
void printList(listPointer first)
{
   printf("The list contains: ");
   for (; first; first = first→link)
     printf("%4d",first→data);
   printf("\n");
}
```

Additional List operations

```
typedef struct listNode *listPointer;
typedef struct {
        char data;
        listPointer link;
        } listNode;
```

1. **Inverting (Reversing) Singly linked List**

```
listPointer invert(listPointer lead)
{/* invert the list pointed to by lead */
   listPointer middle,trail;
   middle = NULL;
   while (lead) {
      trail = middle;
      middle = lead;
      lead = lead→link;
      middle→link = trail;
   }
   return middle;
}
```

2. **Concatenating singly Linked list**

```
listPointer concatenate(listPointer ptr1, listPointer ptr2)
{/* produce a new list that contains the list
    ptr1 followed by the list ptr2. The
    list pointed to by ptr1 is changed permanently */
   listPointer temp;
   /* check for empty lists */
   if (!ptr1) return ptr2;
   if (!ptr2) return ptr1;

   /* neither list is empty, find end of first list */
   for (temp = ptr1; temp→link; temp = temp→link) ;

   /* link end of first to start of second */
   temp→link = ptr2;
}
```

3. Searching:

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.

<u>LIST Is Unsorted</u>

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).

<u>LIST is Sorted</u>

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.

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*

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.

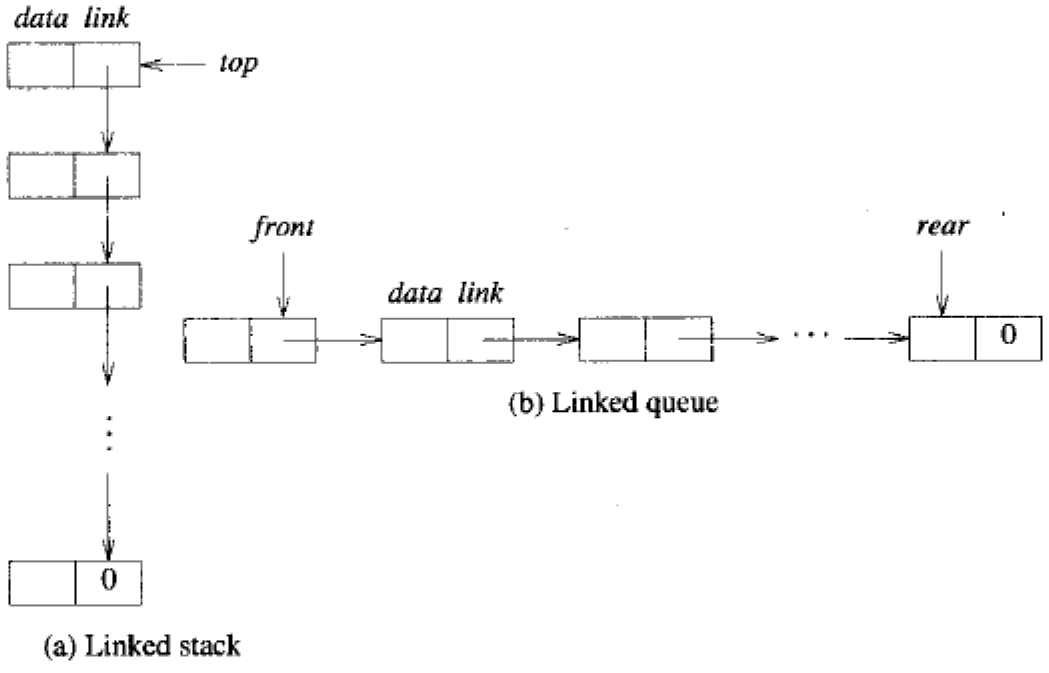
```
    1. Set PTR: = START.
    2. Repeat Step 3 while PTR ≠ NULL
    3.      If ITEM < PTR→INFO, then:
                 Set PTR: = PTR→LINK
            Else if ITEM = PTR→INFO, then:
                 Set LOC: = PTR, and Exit. [Search is successful.]
            Else:
                 Set LOC: = NULL, and Exit. [ITEM now exceeds PTR→INFO]
            [End of If structure.]
            [End of Step 2 loop.]
    4. Set LOC: = NULL.
    5. Exit.
```

Deleting the Node with a Given ITEM of Information on

- 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.
```

LINKED STACKS



Linked stack and queue

The above 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

If we wish to represent $n \le MAX\_STACKS$ stacks simultaneously, we begin with 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];
```

We assume that the initial condition for the stacks is:

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

and the boundary condition is:

$$top[i] = NULL \text{ iff the } i\text{th stack is empty}$$

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).

```
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;
}
```

Add to 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 i^{th} stack would be item = pop (i);

```
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;
}
```

Delete from a linked stack

LINKED QUEUES

To represent $m \leq MAX\_QUEUES$ queues simultaneously, we begin with 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];
```

We assume that the initial condition for the queues is:

$$front[i] = NULL, 0 \leq i < MAX\_QUEUES$$

and the boundary condition is:

$$front[i] = \text{NULL iff the } i\text{th queue is empty}$$

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.

```
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;
}
```

Add to the rear 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);

```
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;
}
```

Delete from the front of a linked queue

APPLICATIONS OF LINKED LISTS

POLYNOMIALS

Representation of the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \cdots > e_1 > e_0 \geq 0$. We represent each term as a node containing coefficient and exponent fields, as well as a pointer to the next term. Assuming that the coefficients are integers, the type declarations are:

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

We draw *polyNode*s as:

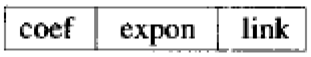
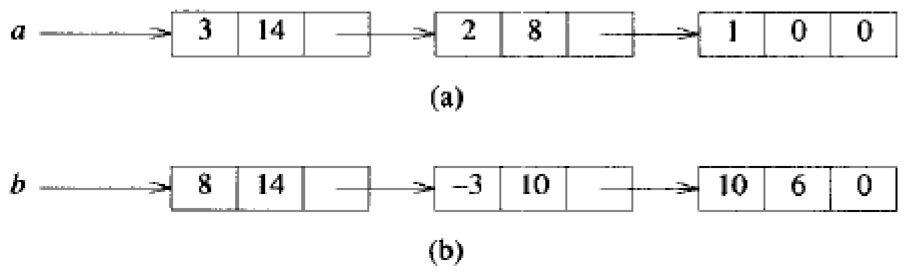


Figure 4.12 shows how we would store the polynomials

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

and

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



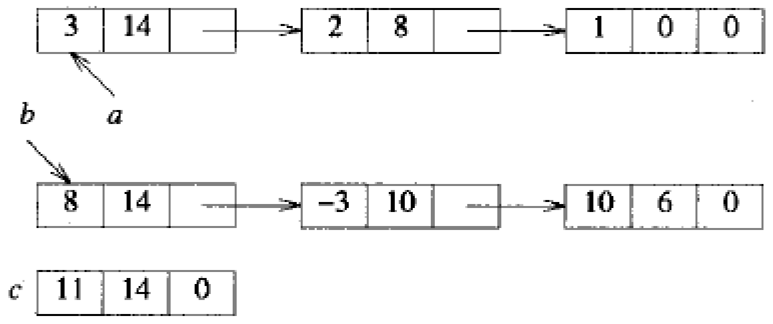
Representation of $3x^{14}+2x^8+1$ and $8x^{14}-3x^{10}+10x^6$

Adding Polynomials

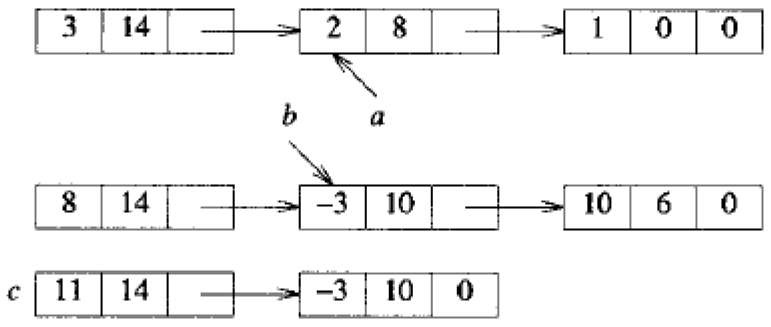
To add two polynomials, examine their terms starting at the nodes pointed to by ***a*** and ***b***.

- If the exponents of the two terms are equal, then add the two coefficients and cre ate 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***

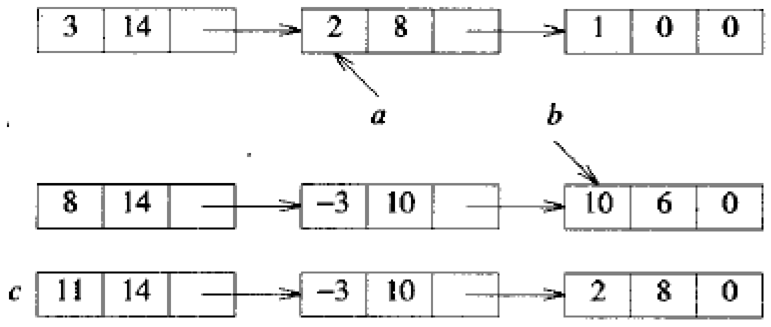
Below figure illustrates this process for the polynomial s addition.



(i) $a \rightarrow expon == b \rightarrow expon$



(ii) $a \rightarrow expon < b \rightarrow expon$



(iii) $a \rightarrow expon > b \rightarrow expon$

```
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;
}
```

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;
}
```

Attach a node to the end of a list

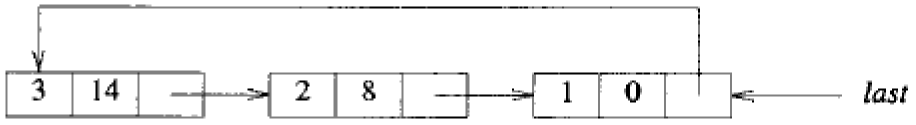
Erase polynomials

```
void erase(polyPointer *ptr)
{/* erase the polynomial pointed to by ptr */
   polyPointer temp;
   while (*ptr) {
      temp = *ptr;
      *ptr = (*ptr)→link;
      free(temp);
   }
}
```

Erasing a polynomial

Circular List Representation of Polynomials

We can free all the nodes of a polynomial more efficiently if we modify our list structure so that the link field of the last node points to the first node in the list (see Figure 4.14). We call this a *circular list*. A singly linked list in which the last node has a null link is called a *chain*.



Circular representation of $3x^{14} + 2x^{8} + 1$

As we indicated earlier, we free nodes that are no longer in use so that we may reuse these nodes later. We can meet this objective, and obtain an efficient erase algorithm for circular lists, by maintaining our own list (as a chain) of nodes that have been "freed." When we need a new node, we examine this list. If the list is not empty, then we may use one of its nodes. Only when the list is empty do we need to use *malloc* to create a new node.

Let *avail* be a variable of type *polyPointer* that points to the first node in our list of freed nodes. Henceforth, we call this list the available space list or *avail* list. Initially, we set *avail* to *NULL*. Instead of using *malloc* and *free*, we now use *getNode* (Program 4.12) and *retNode* (Program 4.13).

We may erase a circular list in a fixed amount of time independent of the number of nodes in the list using *cerase*

```
void cerase(polyPointer *ptr)
{/* erase the circular list pointed to by ptr */
   polyPointer temp;
   if (*ptr) {
      temp = (*ptr)→link;
      (*ptr)→link = avail;
      avail = temp;
      *ptr = NULL;
   }
}
```

Erasing a circular list

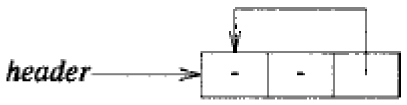
```
polyPointer getNode(void)
{/* provide a node for use */
   polyPointer node;
   if (avail) {
      node = avail;
      avail = avail→link;
   }
   else
      MALLOC(node, sizeof(*node));
   return node;
}
```

Program 4.12: *getNode* function

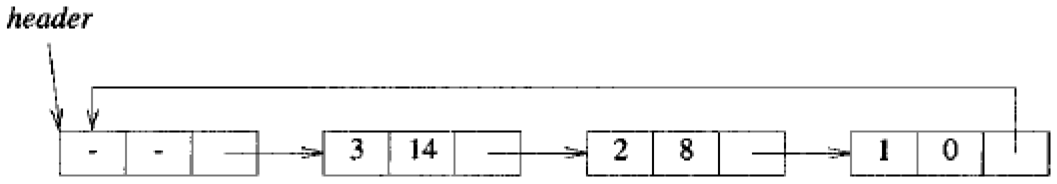
```
void retNode(polyPointer node)
{/* return a node to the available list */
   node→link = avail;
   avail = node;
}
```

Program 4.13: *retNode* function

Circular List representation of Polynomials with Header Nodes



(a) Zero polynomial



(b) $3x^{14} + 2x^{8} + 1$

Adding two polynomials represented as circular lists with header nodes

```
polyPointer cpadd(polyPointer a, polyPointer b)
{/* polynomials a and b are singly linked circular lists
    with a header node. Return a polynomial which is
    the sum of a and b */
  polyPointer startA, c, lastC;
  int sum, done = FALSE;
  startA = a;          /* record start of a */
  a = a→link;          /* skip header node for a and b*/
  b = b→link;
  c = getNode();       /* get a header node for sum */
  c→expon = -1; lastC = c;
  do {
     switch (COMPARE(a→expon, b→expon)) {
        case -1: /* a→expon < b→expon */
              attach(b→coef,b→expon,&lastC);
              b = b→link;
              break;
        case 0:  /* a→expon = b→expon */
              if (startA == a)  done = TRUE;
              else {
                 sum = a→coef + b→coef;
                 if (sum) attach(sum,a→expon,&lastC);
                 a = a→link; b = b→link;
              }
              break;
        case 1:  /* a→expon > b→expon */
              attach(a→coef,a→expon,&lastC);
              a = a→link;
     }
  } while (!done);
  lastC→link = c;
  return c;
}
```

```
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;
}
```

Attach a node to the end of a list

<u>Operations on circular singly linked List</u>

```
void insertFront(listPointer *last, listPointer node)
{/* insert node at the front of the circular list whose
    last node is last */
  if (!(*last)) {
  /* list is empty, change last to point to new entry */
    *last = node;
    node→link = node;
  }
  else {
  /* list is not empty, add new entry at front */
    node→link = (*last)→link;
    (*last)→link = node;
  }
}
```

Inserting at the front of a list

To insert at the rear, we only need to add the additional statement **last = node* to the **else** clause of *insertFront*

```
int length(listPointer last)
{/* find the length of the circular list last */
   listPointer temp;
   int count = 0;
   if (last) {
      temp = last;
      do {
         count++;
         temp = temp→link;
      } while (temp != last);
   }
   return count;
}
```

Finding the length of a circular list

SPARSE MATRIX REPRESENTATON

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.

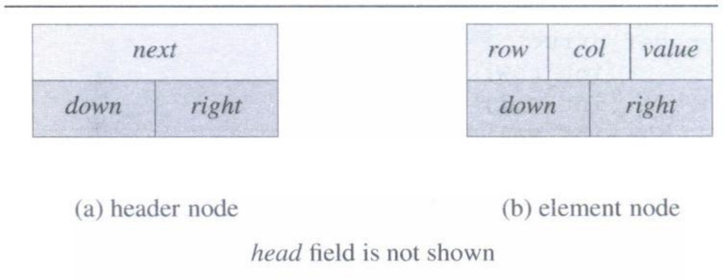


Figure: Node structure for sparse matrices

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

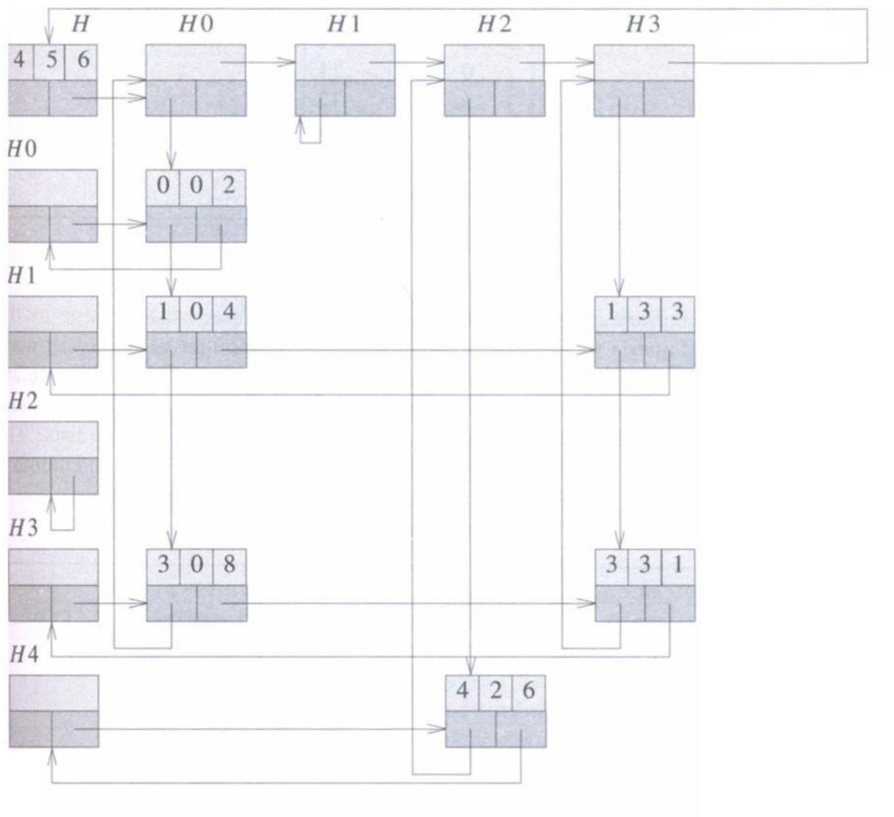


Figure (3) : Linked representation of the sparse matrix of Figure (2) (the *head* field of a node is not shown)

To represent a *numRows x numCols* matrix with *numTerms* nonzero terms, then we need max *{numRows, numCols} + numTerms + 1* node. 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 struct matrixNode *matrixPointer;
typedef struct {
        int row;
        int col;
        int value;
        } entryNode;
typedef struct {
        matrixPointer down;
        matrixPointer right;
        tagfield tag;
        union {
             matrixPointer next;
             entryNode entry;
             } u;
        } matrixNode;
matrixPointer hdnode[MAX_SIZE];
```

DOUBLY LINKED LIST

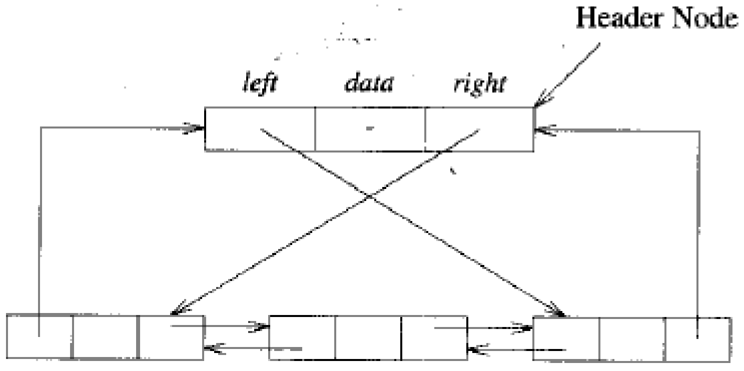
1. The difficulties with single linked lists is that, it is possible to traversal only in one direction, ie., direction of the links.
2. The only way to find the node that precedes p is to start at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a singly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

A node in a doubly linked list has at least three fields, a left link field (*llink*), a data field (*data*), and a right link field (*rlink*). The necessary declarations are:

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



Doubly linked circular list with header node

Insertion into a doubly linked list

Insertion into a doubly linked list is fairly easy. Assume there are two nodes, node and new node, 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;
}
```

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);
   }
}
```

Deletion from a doubly linked circular list

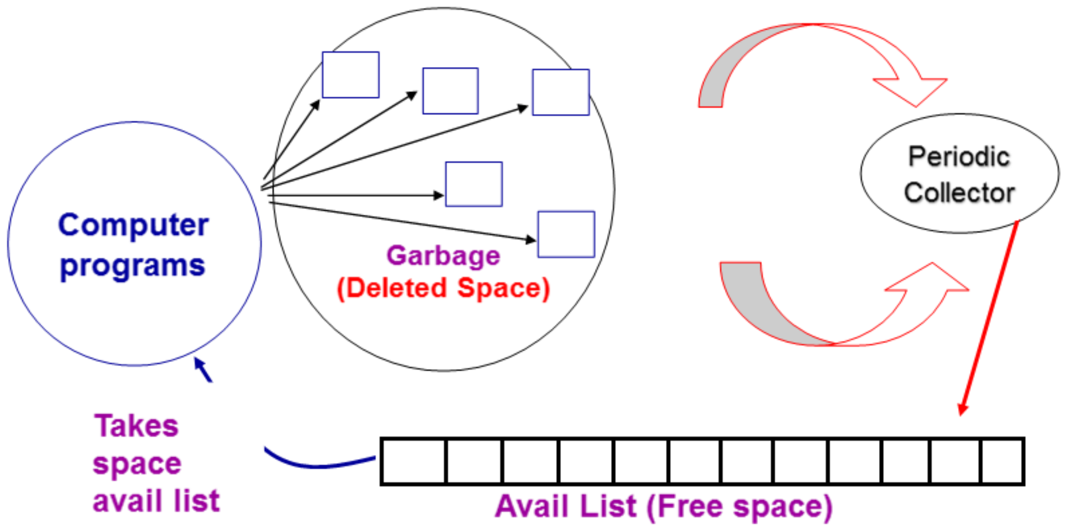
<u>Garbage Collection</u>

- 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 needed 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 del eted space onto the free storage 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