## 6.7.1 Creating a singly linked list : Create() function

We expect `Create()` function to accept the address of the first node and element to be added to the linked list. It should return the modified linked list to the calling program. We will show how to start from an empty list, creation and add nodes one by one. This function adds only one node at a time. To add more nodes, it should be invoked many times.

To build a linked list we can add at the front of the first node or add at the end. In this section we shall follow the earlier approach. Figure 6.5 shows a sample list with two elements (do not worry about how these two elements have been added!).

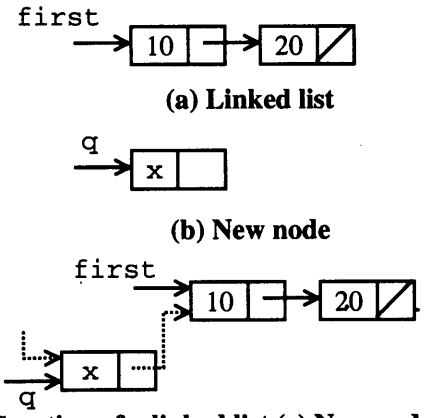



**Fig. 6.5 Creation of a linked list (c) New node added at front**

The variable `q` is a pointer to this new node to be added (Figure 6.5(b)). To attach this new node at front of first node, the links shown in dotted lines must be done - see Figure 6.5(c). The statements required to do this are,

```
q->link = first;
first = q;
return first;
```

Now the question is, whether the same piece of code will work for `first = NULL`? We will see soon (see Figure 6.6).

Assume, `first = NULL;`

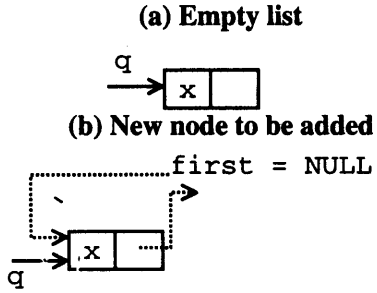



**Fig. 6.6 Adding multiple nodes (c) Node x added to front, when first = 0**

when we execute the code earlier with `first = NULL`, we have

```
q->link = NULL;
first = q;
return first;
```

Indeed it has created the first node successfully. Hence, the same piece of code works for all conditions and the complete code for `Create()` function appears in Program 6.1.

*Program 6.1*
*Linked List creation*

```
NODE Create (NODE first, int x)
{ /* adds x to the front of the list */
     NODE q;
     q = (NODE) malloc (sizeof(struct List));
     q->info = x;
     q->link = first; /* add to front of first */
     first = q;
     return first;
}
```

**Before `Create()` is invoked, you must ensure that `first` is initialized to `NULL`.** The following *main* program does the initialization and builds a linked list of 10 elements with `info` 1,2,3,....,10.

```
void main()
{
     first = NULL;

     for (i = 1; i <= 10; i++)
          first = Create(first, i);
}
```

## 6.7.2 Inserting a node into a linked list : Insert() function

This section develops a function `Insert()` to insert an element `x` after the *k*th element. The parameter `k` can take only values from 0 to maximum length of the list. If `k = 0`, the node `x` will be added before the first node and if `k` > length of the list an error will be displayed. Figure 6.7 shows the method in which the node `x` be added after the kth node.

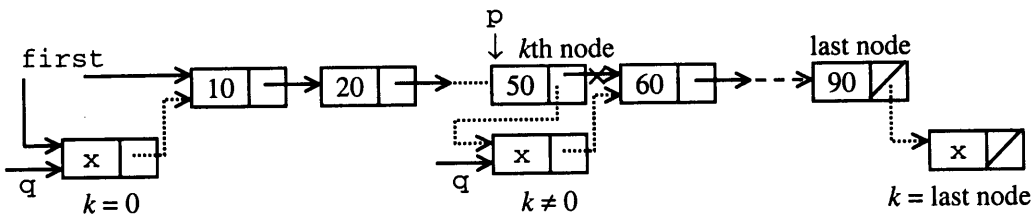



**Fig. 6.7 Inserting x after *k*th element**

Before we attempt to insert element x, firstly we should know whether the kth element exists in the list or not. There are two situations for the error message to the displayed.

(1) When k < 0.

(2) When k > length of the list.

Putting all the points together an algorithm shown in Algorithm 6.1 can be written.

***Algorithm 6.1 to insert* x *after* kth**

```
Algorithm Insert(first, k, x)
{
     // first - address of the first node.
     // k - kth element.
     // x - to be inserted after kth element.
     if(k < 0)
     {
          print("Error");
          return first;
     }

     p = first;
     for (index = 1; p && index < k; index++)
          p = link(p);     // move p to kth position.
     if (k > 0 && !p) // takes care of empty list also
     {
          print("out of list");
          return first;
     }

     // found kth position, so insert x
     q = getnode();
     info(q) = x;
     if (k)          //insert after p
     {
          link(q) = link(p);
          link(p) = q;
     }
```

```
    else              // insert at front of first
    {
          link(q) = first;
          first = q;
    }
    return first;
}
```

To implement Algorithm 6.1 in C language you need minimum effort and so the code is shown in Program 6.2 without any explanation.

***Program 6.2***
***Insertion of x after kth position***

```
NODE Insert (NODE first, int k, int x)
{ /* Insert x after kth element          */
  /* If no kth element, then print error */

    NODE q, current;
    int index;
    current = first;
    if (k < 0)
    {
          printf("Error-no insertion\n");
          return first;
    }

    /* current points to the kth element */
    for (index = 1; current && index < k; index++)
          current = current->link;

    if (k > 0 && !current)
    {
          printf("Error-Out of bound\n");
          return first;
    }

    /* Insert x */
    q = (NODE) malloc (sizeof(struct List));
    q->info = x;
    if (k)  /* k > 0, so insert after current */
    {
          q->link = current->link;
          current->link = q;
    }
    else   /* k = 0, so insert as first element */
```

```
    {
        q->link = current;
        first = q;
    }
    return first;
}
```

The `for` loop scans through the list looking for the *k*th node. It uses `index` as the index variable. The scanning stops when either the *k*th node is found or end of list is reached. If `p = NULL` with `k > 0`, this means that it is out of bound and hence node `x` cannot be inserted, else you can insert either at front of the first node or after *k*th node. So, variable `p` points to the *k*th node. One final confirmation is to be done to check whether the same code works when *k*th node is the last node (Check it yourself).

## 6.7.3. Deleting an element x : LDelete()

The objective of this function is to delete a node whose info field contains `x` and return the modified list. It is invoked with two parameters,

(1) The pointer to the linked list.
(2) The element to be deleted.

The return type is selected as `NODE` (because the modified list should be returned to the calling program). Now, we will develop the code for this problem first by explaining with a diagram (see Figure 6.8). The Figure shows for two cases of x,

(1) when x = 20 and
(2) when x = 60.

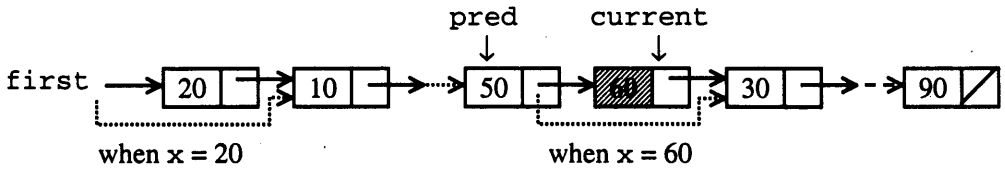



**Fig. 6.8 Deleting node x**

We need the following six steps to carry out the task of deletion.

**Step 1:** If the list is empty, then return 0.
**Step 2:** Maintain a pointer *pred* to move behind *current*.
**Step 3:** Find the address of the node to be deleted using *current* and updating *pred* appropriately.
**Step 4:** If *current* runs out of list – error (node not found)
else link(*pred*) = link(*current*);
**Step 5:** free(*current*)
**Step 6:** return *first*;

The C code for function `LDelete()` appears in program 6.3. When `current` points to `first`, `pred` should be one node behind it, which is NULL. Hence, it is initialized to NULL. The `while` loop may get terminated when either `current` reaches end of list (node not found case) or node x has been found.

*Program 6.3*
*Deletion of a node*

```
NODE LDelete (NODE first , int x, int *flag)
{
     NODE current, pred;
     if (!first)  /* empty list */
     {
          *flag = 0;
          return first;
     }

     /* more than one node case */
     pred = NULL; current = first;
     while (current && (current->info != x))
     {
          pred = current;
          current = current->link;
     }

     if (current == NULL) *flag = 0;/* node not found */
     else if (!pred) /* key is first node */
     {
          first = first->link;
          *flag = 1;
     }
     else  /* middle node */
     {
          pred->link = current->link; /* remove */
          *flag = 1;
     }

     return first;
}
```

If it is found, delete using the statement,

```
pred->link = current->link;
```

This statement drops the element pointed by `current`. Now this node can be freed. The linking is shown with broken lines in Figure 6.8.

### 6.7.4 Length of a Linked List: Length() function

This function accepts the linked list as its parameter and returns the number of nodes (i.e., length) in the list. Obviously the return type is integer and there is no need to return the list, because we do not modify the list within the function. See Program 6.4 for the C code of `Length()` function.

*Program 6.4*
*List Length*

```
int Length (NODE first)
{
     int len = 0; /* counter */
     while (first)
     {
          len++;
          first = first->link;
     }
     return len;
}
```

The function uses the counter `count` with initial value 0 and incremented as we move through the list until we reach the end of the list. Finally the local variable `len` with the number of nodes counted is returned to the calling program.

### 6.7.5 Searching a node in linked list: Search() function

Assume that we are given a key node `x` (`info` field) and our aim is to search the list to find whether this node exists in the linked list or not. If it is found, return the position of `x` (nodes are numbered as 1 for the first node, 2 for the second node, and so on). Supposing, if the node is not found, then a 0 is returned (see Program 6.5).

*Program 6.5*
*Search for a key*

```
int Search (NODE first, int x)
{
     int index = 1; /* indexing the nodes */
     while (first && first->info != x)
```

```
    {
        index++;
        first = first->link;
    }
    if (first)
        return index; /* key found */
    return 0; /* key not found */
}
```

Since the node number is not stored in the node itself, we maintain a variable `index` to return the position of `x`. Similar to `Length()`, the return type for this function is also `int`. The `while` loop checks for either the end of the list (node not found) or the key node `x` (found). When the list is scanned, `index` is incremented to remember the node number.

When `current` is not `NULL`, position of `x` is nothing but the value of `index` and its value is returned, otherwise a 0 is returned.

## 6.7.6 Finding the *k*th element: Find()

In the function `Find(k, x)`, `k` is the *k*th element in the list. If such element exists, it is set to `x` and a *true* is returned to the calling routine (hence it should be a reference parameter). If the value of `k` is invalid, that is no *k*th element, then return *false*. Hence, the return type for `Find()` is `int`. For example, a sample list shown if Figure 6.9 gives answers for various values of `k` as,

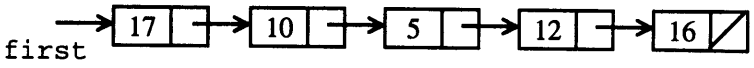



**Fig. 6.9 A sample linked list**

1. *Find*(2, *x*) ⇒ x = 10 and return value is *true*
2. *Find*(6, *x*) ⇒ x = ? and return value is *false*
3. *Find*(0, *x*) ⇒ x = ? and return value is *false*
4. *Find*(5, *x*) ⇒ x = 16 and returned value is *true*

The C code is shown for the function `Find()` in Program 6.6. The `while` loop searches for the *k*th node using `index` (starting from node 1). In most of the linked list based programs, you will have an `&&` condition for the termination of the loop, in case if it goes out of the list. Here again we have such a condition

```
while (current && index < k) {. . .}
```

If `current` is `NULL`, the required *k*th element is not found and *false* is returned.

*Program 6.6*
*Finding a node in the list*

```
int Find (NODE first, int k, int *x)
{
     int index = 1; /* indexing the nodes */

     if (k < 1) return 0;
     while (first && index < k)
     {
          index++;
          first = first->link;
     }
     if (first) /* kth element found */
     {
          *x = first->info;
          return 1;
     }
     return 0; /* kth element not found */
}
```

### 6.7.7 Erasing all nodes in linked lists: Destroy ()

Freeing of a node in a linked list, when it is no longer useful must be done for efficient memory management. Some compilers take care of this by deallocating all dynamically allocated memory before they return to the calling routine. However, there are compilers that may leave this work to the programmers. What ever be the strategy followed by the compilers, we shall assume that it is our responsibility to deallocate all memory allocated for the nodes in a linked list. The function `Destroy()` does this (see program 6.7).

*Program 6.7*
*Delete all nodes*

```
void Destroy (NODE first)
{ /* delete all nodes in the linked list */

     NODE next;
     while (first)
     {
          next = first->link; /* go to next node */
          free(first);
```

*Program 6.9*
*Push and Pop*

```
NODE Push (NODE first, int x)
{
     NODE q;
     q = (NODE) malloc (sizeof(struct List));
     q->info = x;
     q->link = first;
     first = q;
     return first;
}

NODE Pop (NODE first , int *x)
{
     if (!first) /* empty list */
     {
          *x = -1;
          return first;
     }
     *x = first->info;
     first = first->link;
     return first;
}
```

Since, `first` always points to the top of the stack, popping means return the element pointed by `first`. The pointer `first` should be advanced to the next node in the list (see Program 6.9). Only underflow condition need to be checked during *pop* operation and overflow case does not arise here because, `malloc()` allocates memory dynamically until system memory is available.

## 6.9 QUEUE AS LINKED LIST

The FIFO structure or a queue can also be implemented using a linked list. We will again maintain two pointers `front` and `rear` for the queue similar to the array implementation Note that you can design even with a single pointer like an ordinary linked list. However, the basic definition of queue- *rear* insertion and *front* deletion is followed. Figure 6.11 shows the operation of a queue with few additions and deletions.

`front = rear = NULL;`

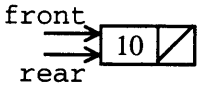
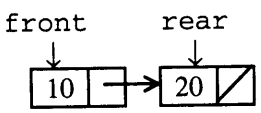
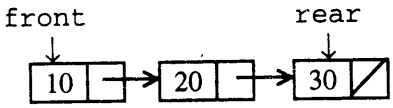



**Fig. 6.11(a) Empty Queue** **(b) Insert 10**




(c) Element 20 is added at rear




(d) Element 30 is added at rear end

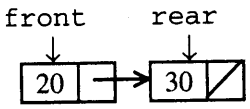



**Fig. 6.11(e) Element 10 is deleted**

Queue elements are added at rear end, using the `rear` pointer. So, we add nodes at the end of the list. When an elements is to be deleted, the `front` pointer is advanced by one node (note the similarly with `Pop()` of stack). In general, `front` points to the oldest element and `rear` points to the most recent elements.

## 6.9.1 Queue Insertion and Deletion: InsertRight() and RemLeft()

The Function `InsertRight()`, adds elements at the rear end based on the `rear` pointer. Similarly, `RemLeft()` removes an element from the front end pointed by `front`, provided the list is not empty.

As both the functions alter `front` and `rear` pointers, we cannot return the address of these two pointers. Hence, `InsertRight()` is to be designed with `void` as its return value and also we assume that both `front` and `rear` are global variables. The function `RemLeft()` can be designed as a function which returns the deleted element. Both of these functions appear in Program 6.10 and 6.11 respectively.

*Program 6.10*
*Rear Insertion*

```
void InsertRight (int x)
{
     NODE q;
     q = (NODE) malloc (sizeof(struct List));
     q->info = x; q->link = NULL;
     if (!front) /* front is NULL)
     {
          front = rear = q;
          return;
     }
     rear->link = q;  rear = q;
}
```

```
    current = front;
    while (current ->link != rear)
          current = current ->link;
    rear = current;
    rear->link = NULL;
}
```

## 6.11 ORDERED LINKED LIST

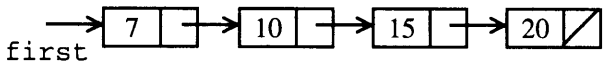
An ordered list is one in which the elements of the list are in a particular order - ascending or descending. Imagine that a set of names are stored in a linked list in some arbitrary order. Hence, the names cannot be displayed alphabetically by one scan of the list. If, however, the list is created with an intrinsic alphabetical ordering the display function can directly print the names using a single looping structure.

Therefore, it is more efficient if the list in constructed with the ordering built-in. The objective of this section is to devise a C program to build an ordered list.

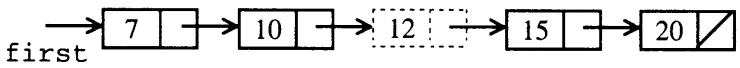
The elements are inserted in the existing list keeping the list order in mind even after the new element is added. Consider the following list of elements that have been constructed (some how!) as an ordered list.

7, 10, 15, 20

The corresponding list is shown in Figure 6.14(a).




**(a) Initial list**




**(b) List after adding element 12**

**Fig. 6.14 Ordered list**

When we wish to add a new element 12, it should occupy after 10 to retain the ascending order as shown in Figure 6.14(b). To do this, we have three possible cases,

Case (1) the new element x may be less than the first element.

Case (2) the element x may find its location in some *k*th posotion.

Case (3) the element x may be greater than all the elements in the list (to be inserted as the last element).

### The Method and C code

To find the position of x for insertion, start scanning from the first node using p and maintaining a predecessor or trail pointer, tp. The pointer p will move as long as the new element x is greater than p->info (see Figure 6.15).

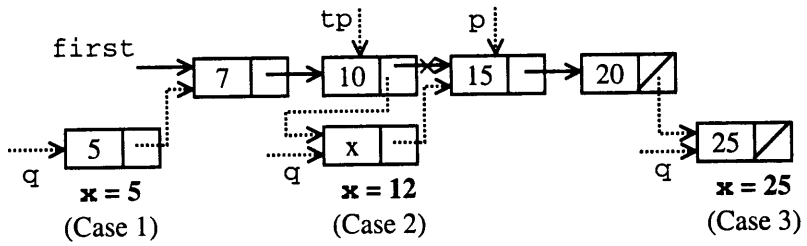



**Fig. 6.15 All three cases for inserting x**

One single for loop will find the location for insertion and is shown below:

```
tp = 0;
for (; p && p-> info < x; tp = p, p = p->link);
```

1. When tp = 0; case 1 is handled (for loop will not be executed even once).
2. When tp! = 0 and p != 0; case 2 is handled (tp will be one node behind p - i.e., tp will point to node 10).
3. When tp != 0 and p = 0; case 3 is handled (tp will point to last node i.e., 25).

After the appropriate address of the node is located, the next work is to link the new node with the existing list. The function SortedInsert() is shown in Program 6.13.

*Program 6.13*
*Ordered linked list*

```
NODE SortedInsert (NODE first, int  x)
{ /* To insert elements in ascending order */
     NODE p = first;
     NODE q;          /* new node       */
     NODE tp = 0;     /* trail pointer */

     /* move tp to appropriate place,       */
     /* so that x can be inserted after tp */
     for (; p && p->info < x; tp = p, p = p->link);

     q =  (NODE) malloc(sizeof (struct List));
     q->info = x;
     q->link = p;
     if (tp)
          tp->link = q; /* insert in middle or end */
```

*Program 6.14*
*Simple Merge*

```
/* Merges two lists pointed by p1 and p2.                          */
/* The pointer p3 points to the merged list.                       */
/* p3 contains the addresses of the elements in p1 and p2          */
/* in an ascending order. Therefore, list pointed                  */
/* by p3 is a list of addresses and not the actual elements */
NODE Merge (NODE p1, NODE p2)
{
     NODE p3;
     p3 = NULL;
     /* store the address of the smaller of p1 and p2 */
     while (p1 && p2)
          if (p1->info <= p2->info)
          {
               p3 = InsRear(p3, p1);
               p1 = p1->link;
          }
          else
          {
               p3 = InsRear(p3, p2);
               p2 = p2->link;
          }

          /* copy remaining elements of p1, if any */
     while (p1)
     {
          p3 = InsRear(p3, p1);
          p1 = p1->link;
     }
     /* copy remaining elements of p2, if any */
     while (p2)
     {
          p3 = InsRear(p3, p2);
          p2 = p2->link;
     }
     return p3;
}

NODE InsRear(NODE p, NODE x)
{  /* inserts x at the rear of p3 */
     NODE t, q;
     q = (NODE) malloc (sizeof(struct List));
     q->info = (int) x; /* typecast x as int */
     q->link = 0;
```

```
    if (p)     /* p points to the last node */
    {
          for (t = p; t->link != NULL; t = t->link);
          t->link = q;  /* add x to rear */
    }
    else p = q; /* first node */
    return p;
}
```

The working of `Merge()` is very simple. In this function, the first while loop scans `p1` or `p2` which ever has a smaller element and its address is stored in `p3`. Any node added to `p3` will be at the end so that the ascending order is maintained. For this purpose we shall use another function `InsRear(NODE, NODE)`. The second parameter is of type `NODE` because we must save the address of `p1` (or `p2`) which is of type `NODE`. As explained already, the address is taken as `int` and stored in the info filed of `p3`.

As we may have different lengths of `p1` and `p2`, when any one list gets exhausted, the remaining elements addresses from `p1` (or `p2`) are copied to `p3` with the second or third `while` loop. Notice that the addresses are appended to `p3` by using `InsRear()` again.

At the end of this function execution, we will have `p3` pointing to a list of addresses of the elements in `p1` and `p2` in the ascending order. We can display the contents of `p3` using a slightly different technique and is shown in Program 6.14(a).

*Program 6.14 (a)*
*Display of p3*

```
void MDisplay (NODE p)
{
     NODE t;
     if (!p)
     {
          printf("Empty List\n");
          return;
     }
     while (p)
     {
          t =(NODE) p->info;
          printf("%d ",t->info);
          p = p->link;
     }
     printf("\n");
}
```

*Example 6.4: Copying one linked list to another*

### Problem statement

String copy, `strcpy()`, in C language is defined in `<string.h>` header file to copy the source string to a destination string. Similarly, in this problem, given a linked list the task is to copy this to a destination linked list.

The pseudo code for this problem appears in Figure 6.21. The destination linked list is built by reading the elements from the source linked list until the list is exhausted.

```
Algorithm Copy(dst, src)
{
     // src- source linked list.
     // dst - destination list - replica of src.
     dst = NULL;
     while (src) do
     {
          q = getnode();
          get the info from src and put it in q.
          if (dst)   // add new node at the end.
          {
               link(last) = q;
               last = q;
          }
          else
               dst = last = q;  // first node
          src = link(src);      // go to the next node
     }
     return dst;      // return the copied list
}
```

**Fig. 6.21 Pseudo code for linked list copy**

### Implementation

The implementation of `Copy(dst, src)` is similar to `strcpy`. Copy one node at a time from `src`; this means that we add the node at the end of the partial `dst` list. Program 6.17 shows this function in C.

*Program 6.17*
*Linked List Copy*

```
void Copy (NODE *dst, NODE src)
{  /* copy src to dst */
     NODE q;
     NODE last; /* to add nodes at the rear of dst */
     if (!src)  /* src is empty */
```

```
    {
        *dst = 0;
        return;
    }
    *dst = 0;
    while (src)
    {
        q = (NODE) malloc (sizeof (struct List));
        q->info = src->info;
        q->link = NULL;
        if (*dst == NULL)
            *dst = last = q;
        else
        {
            last->link = q;
            last = q;
        }
        src = src->link;
    }
}
```

The `while` loop scans through the source list, `src` and the `info` of each node is added at rear of destination list, `dst`. For this purpose the pointer `last` is used. The `if-else` construct differentiates whether first node creation is done or more than one node creation.

Notice that to make it uniform with respect to string copy function, the return type of `Copy()` is declared as `void`. The copied list is to be returned via `dst`. Hence, it is declared as a reference parameter. The `dst` parameter is already a pointer and to obtain the reference parameter treatment, it should be declared as pointer to pointer or `**`. You may invoke the function as,

```
Copy(&dst, src);
```

where, `dst` is the receiving parameter and `src` is the list whose copy is to be obtained in `dst`.

## 6.13 A COMPARISON BETWEEN ARRAYS AND LINKED LISTS

This section addresses an important data representation problem - array or linked list. We cannot arbitrarily conclude that array (or linked list) is a better choice for a given problem. First of all, we must decide whether the data size is known in advance or not. If it is known, then array may be a better choice. Without going into further discussion, we shall formally list the advantages and disadvantages of arrays versus linked lists.

(1) Arrays are formula-based representation of data. The formula stores successive elements of a list in contiguous memory locations.
In a linked list representation the elements of the list may be stored in any arbitrary locations. Each element has an explicit pointer (or link) that tells us the address of the next element in the list

(2) Through accessing of array elements is random (`a[5]` will take you to 6th element), for every accessing the associated formula must be evaluated which consumes lot of time.

In linked lists, no explicit formula is required to move to the next element, making it time efficient. However, random accessing is not possible in linked lists. For example, to reach sixth node, you must start from the first node.

(3) Memory allocation for arrays is done at compilation time, so at the run time there is no extra work required for memory allocation and deallocation.

In a linked list, on the other hand, memory allocation and deallocation is done at run time with run time manager software. Generally, in dynamic memory allocation there are two major problems (1) garbage collection (2) dangling reference.

(4) The insertion and deletion are two common operations required for most of the applications. In arrays inserting an element in the middle requires shifting of all remaining elements to right side. Similarly, deleting a middle element requires compaction of array elements. Both these are extra work and most of the time it is undesirable.

Linked lists are best suited for insertion and deletion without shifting remaining elements.

(5) The adjacency information need not be stored in the current element, when the data objects are stored in arrays.

On the other hand, in linked list the next node's address is stored in the current node itself. If by chance, the link is lost we can't get back the original list. This is not the case with arrays, because arrays remain alive through out the program execution.

## 6.14 CIRCULAR LINKED LISTS

A **circular linked list** is one in which the link address of the last node is connected back to the first node. Applications can be made simplified and can be run faster with circular lists.

The circular list with $n$ data elements appear in Figure 6.22. The list consists of $e_1$, $e_2$, ...., $e_n$ nodes and notice that the link address of the last node is pointing to first node $e_1$.

You may be wondering that why `first` is pointing to node $e_n$ and not $e_1$. In a circular list there is no node called **first** and no node as **last**, because it is in a circular

fashion. When first points to $e_n$, it is easier to add new nodes to the list rather than when it points to the so called **first** node $e_1$.

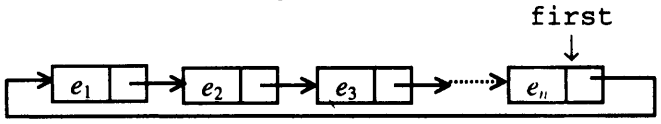
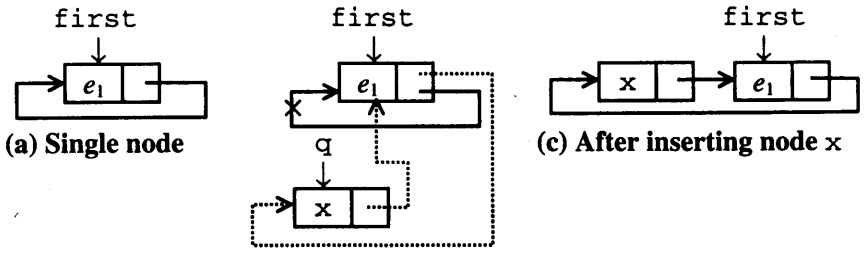



**Fig. 6.22 Circular list**

In Section 6.5, we discussed about the ADT for a singly linked list. The same specification can be used for a circular list as well. But, we shall not show all the operations for the circular list but only few of them.

## 6.14.1 Creating a circular list with front insertion Create() function

An empty circular list is one when `first = NULL` and once the nodes are added to the circular list, the `NULL` won't appear in the list. In this section, we will develop a function `Create()` to build a circular list such that it can be used for further operations:




**(b) Node x added to front of $e_1$**

**Fig. 6.23 Circular list**

Figure 6.23 shows a single node case of a circular list. Now to add a new node x to the front of $e_1$, we must do the following steps.

**Step 1** Create a new node and make q to point to it.

**Step 2** `link(q) = (link)first;`

**Step 3** `link(first) = q;`

The address of $e_1$ will always be available with `link(q)` or `q->link`. Therefore, Steps 2 can easily be implemented. This is the advantage of keeping the `first` to point to the **last** node. Step 3 breaks the circular link it was attached to the **old** first node with the new **first** node. The C function `Create()` is shown in Program 6.18.

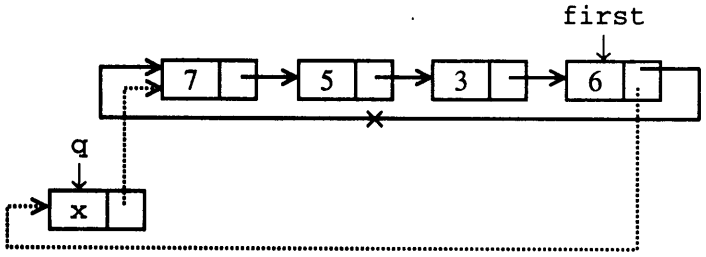
*Program 6.18*
*Creating a Circular list*

```
NODE Create (NODE first, int x)
{
     NODE q;
     q =(NODE) malloc (sizeof(struct List));
     q->info = x;
     if (first == NULL)
     {
          q->link = q;
          first = q;
     }
     else
     {
          q->link = first->link;
          first->link = q;
     }
     return first;
}
```

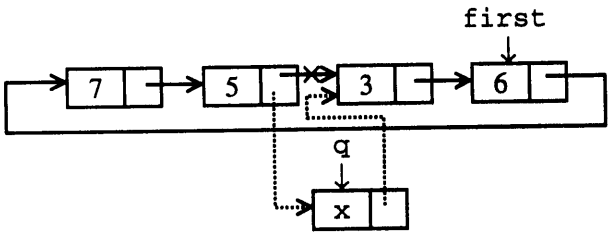
The first node creation as shown in Figure 6.23(a) is done with an `if` construct. After creating the first node the remaining nodes can be constructed using Step 2 and Step 3.

## 6.14.2 Inserting a node into a linked list: Insert() function

In Section 6.7.2 we developed a similar kind of function using a linked list. We will again repeat that here, but this time it is for a circular list. In `Insert(first,k,x)`, recall that we need to insert x after the *k*th node. Figure 6.24 shows a diagram to insert x when $k = 0$ and $k \neq 0$.




**(a) When $k = 0$**




**(b) When $k \neq 0$**

**Fig. 6.24 Inserting element after *k*th position**

Program 6.23 shows the `Insert()` function for a circular list. The main difference of this program with Program 6.2 is in the `for` loop. That is,

```
for (index = 1; index < k && p != first; index ++)
     p = p->link;
```

Assuming that `p` is initialized to the first node, i.e. `p = first->link;` the looping structure scans the list until `p` catches `first`. The other difference lies in the `if-else` construct. When $k = 0$, the linking for circular list is little different, see Figure 6.24(a). The steps required for this case are:

**Step 1** `link(q) = link(first);`
**Step 2** `link(first) = q;`

All cases of $k$ is taken care in Program 6.19.

---

*Program 6.19*
*Insert x after kth node*

---

```
NODE Insert (NODE first, int k, int x)
{
     NODE p, q;
     int index;
     if (k < 0) return first; /* error */

     p = first->link;
     for (index = 1; index < k && p != first; index++)
          p = p->link;

     if (k > 0 && p == first)
     {
          q = getnode();
          q->info = x;
          q->link = p->link;
```

```
            p->link = q;
            first = q;
            return first;
      }

      /* insert x  */
      q = getnode();
      q->info = x;
      if (k)
      {
            q->link = p->link;
            p->link = q;
      }
      else
      {
            q->link = first->link; /* k = 0 */
            first->link = q;
      }
      return first;
}
```

## 6.14.3 Deleting an element from the circular list: LDelete()

The objective of the function `LDelete()` is to delete an element whose `info` field is given as a parameter. In this function, if the element is found in the list, then delete the same. Otherwise, return an error flag with 0 and the list will be unaltered.

Figure 6.25 shows a typical situation where the list contains only one node and multiple nodes in the list.

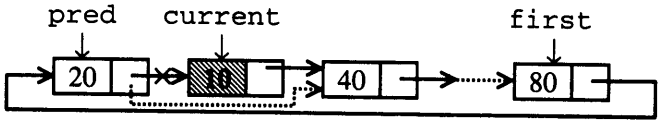



**Fig. 6.25(a) $x = 10$, middle node case**

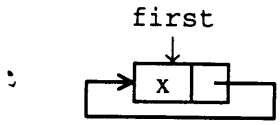



**Fig. 6.25(b) $x = 10$ single node case**

Deleting a node when it is in the middle is little different from with respect to deletion in an ordinary linked list. Assuming x = 10 in Figure 6.25(a) with pred pointing to the previous node of current and also current is not pointing to first, we use

```
pred->link = current->link;
```

In case current points to first, then first should be repositioned to predecessor node (not shown in figure). That is why we have a nested if-else structure for he middle node case. When there is only one node in the list (Figure 6.25(b)), it can be deleted with an if statement.

```
if (first == first->link && first->info == x)
    return (first = 0);
```

In case, x does not match with info, we set the flag to 0. The complete C code is for this problem is shown in Program 6.20.

*Program 6.20*
*Deleting a node*

```
NODE LDelete (NODE first, int x, int *flag)
{
     /* x - element to be deleted       */
     /* flag - 1:successful deletion */
     /* flag - 0:element not found            */
     NODE pred, current;
     pred = first;
     *flag = 1;
     if (!first) { *flag = 0; return 0; }

      /* one node case */
      if (first == first->link && first->info == x)
      {
          return (first = 0);
      }  /* found */
      if (first == first->link && first->info != x)
      {
          *flag = 0;
          return first;
      } /* not found */

     /* get the address of node x */
     current = first;
     do {
         pred = current;
         current = current->link;
     } while (current != first && current->info != x);
```

```
    if (current == first && current->info == x)
    {
         pred->link = current->link; /* middle node */
         first = pred; /* first needs adjustment */
    }
    else if (current != first) pred->link = current->link;
      else  *flag = 0; /* not found */

 return first;
```

## 6.14.4 Displaying the contents of a circular list: Display()

As circular lists do not have NULL to indicate the end of a list, unlike ordinary linked lists, here we use a slightly different technique. The Program 6.21 shows this:

*Program 6.21*
*Display of info field*

```
void Display (NODE first)
{
    NODE q;

    q = first;

    if (first) /* if list not empty - display*/
    do {
         first = first->link;
         printf("%d ",first->info);
    } while (first != q);

}
```

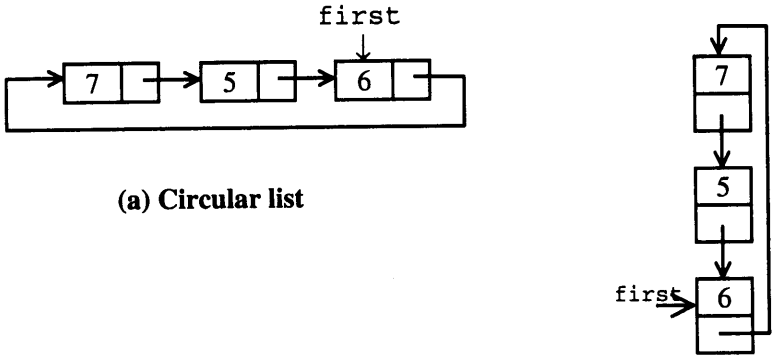
The first `if` statement is to check whether the list is not empty. If it is so, no node info will be displayed. When the list is not empty, `do-while` loop (ideal for circular lists) start displaying from the *first* node to *last* node. The pointer variable `q` does the roll of NULL in an ordinary linked list.

Perhaps the other operations of circular list are not discussed in this text (see Exercise).

## 6.15 STACK AS CIRCULAR LIST

Stack can be represented and in turn implemented using a circular list as well. Section 6.8 explained how a stack could be implemented using an ordinary singly linked list. We need to design `Push()` and `Pop()` functions as we did earlier.

Though we do not get any special advantage of realizing stack using circular list, it does help in some applications wherever we need front insertion (*push*).



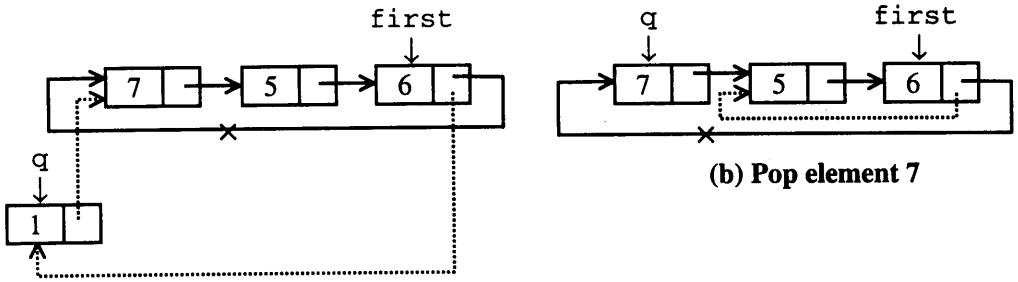

**(a) Circular list**

**(b) Circular list represented as a stack**

**Fig. 6.26 Stack as a Circular list**

By looking at Figure 6.26, we understand that the function `Insert()` of Section 6.14.2 can be used to implement `Push()`. The only change we require is to invoke `Insert()` with $k = 0$. However, we will write a separate function `Push()` and `Pop()` to implement a stack.

Generally, the stack pointer points to the top most element, but in a circular list we shall make `first` to point to the **bottom** element. If we add (*push*) a new element, say 1, it would be inserted above element 7. Similarly, when Pop() is invoked element 7 should be removed from the list (see Figure 6.27).




**(a) Push element 1**

**(b) Pop element 7**

**Fig. 6.27 Stack operations**

Not worrying about single node case, we can write the main code for `Push()` and `Pop()` as follows,

- *Push:*
```
link(q) = link(first);
link(first) = q;
```
- *Pop:*
```
q = link(first);
link(first) = link(q);
```

The new links are shown in broken lines and special cases like empty list, single node case, etc., must be taken care. Program 6.22 and Program 6.23 show the code for `Push()` and `Pop()` functions respectively.

*Program 6.22*
*Push operation*

```
NODE Push (NODE first, int x)
{
     NODE q;
     q = getnode(); /* get dynamic memory */
     q->info = x;
     if (first == NULL)
     {
          q->link = q;
          first = q;
     }
     else
     {
          q->link = first->link;
          first->link = q;
     }
     return first;
}
```

*Program 6.23*
*Pop operation*

```
NODE Pop (NODE first, int *x)
{ /* return popped element through x */
     NODE q;
     if (first == NULL)
     {
          *x = -1;
          return first;
     }

     /* single node case */
```

```
    if (first == first->link)
    {
          *x = first->info;
          first = NULL;
          return first;
    }
    /* more than one node case */
    q = first->link;
    *x = q->info; /* get the last inserted element */
    first->link = q->link;
    return first;
}
```

## 6.16 QUEUE AS CIRCULAR LIST

The objective of this section is to develop an efficient technique for queue insertion and deletion. As usual, maintaining two pointers `front` and `rear` make our design simple and efficient. In this design, `rear` pointer points to the last element (just like pointer `first`) and `front` pointer points to the earliest element (see Figure 6.28).

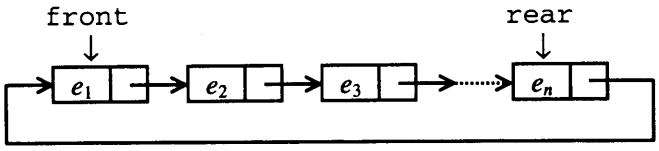



**Fig. 6.28 Queue as a circular list**

When a new element $e_{n+1}$ is added, then the `rear` pointer should point to this node and `front` is unaltered. When an element is deleted, `front` points to $e_2$ and `link(rear)` should point to node $e_2$. The single node case is handled separately and this occurs when `front = = rear`. The diagram showing these operations appears in Figure 6.29.

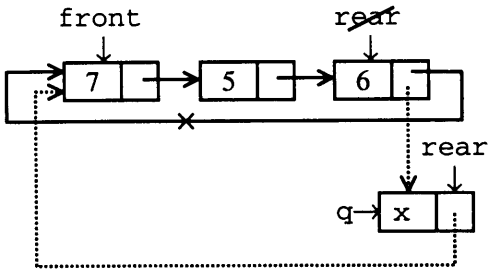



**Fig. 6.29(a) Queue Insertion**

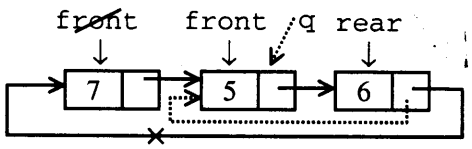



**Fig 6.29(b) Queue Delete**

Also note that, in the figure, the elements are inserted as 7, 5, and 6. Hence, during queue deletion, element 7 is deleted by following the FIFO policy. Subsequent deletion will return element 5 and `front` and `rear` would point to element 6, which is the only element in the list. Calling `QDelete()` again will reset both the pointers `front` and `rear` to NULL. For queue insertion we can use,

```
q->link = front;
rear->link = q;
rear = q;
```

For queue delete,

```
q = front->link;
front->link = q->link;
front = q;
```

Programs 6.24 and 6.25 shows the `QInsert()` and `QDelete()` functions respectively. **Remember that our implementation assumes `front` and `rear` pointers as global.**

---

*Program 6.24*
*Inserting an element in the Queue*

---

```
void QInsert (int x)
{
      NODE q;
      q = getnode(); /* get dynamic memory */
      q->info = x;

      if (!front)
      {
            q->link = q;
            front = rear = q;
      }
      else
      {
            q->link = front;
            rear->link = q;
            rear = q;
      }
}
```

*Program 6.25*
*Deleting an element from the Queue*

```
int QDelete ()
{
      NODE q;
      int x;
      if (!front)
      {
            x = -1;
            return x;
      }
      /* single node case */
      if (front == rear)
      {
            x = front->info;
            front = rear = NULL;
            return x;
      }

      /* more than one node case */
      x = front->info;
      q = front->link;
      front->link = q->link;  /* drop the first node */
      front = q;
      return x;
}
```

## 6.17 CIRCULAR LIST WITH A HEADER NODE

The circular list discussed in the previous section did not have any specific node called as **head** node and a **tail** node. In fact, such a head and tail nodes cannot be designated explicitly in a circular list. To avoid this problem and to make programming easier with circular lists, we add a special node designated as a **header node**. The header node is different from other data element nodes as its information field in filled with -1. This is under the assumption that the regular elements are all positive data. Figure 6.30 shows a sample circular list with a header node.

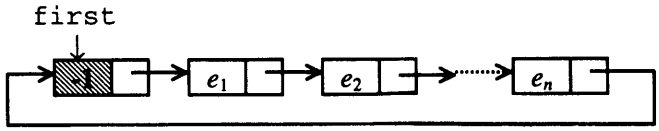



**Fig. 6.30 Circular list with a header node**

We always make the pointer `first` to point to the header node (shaded).
The empty list in these types of circular lists is when the list has only header node. Additions and deletions are done with respect to the leader node. To check for an empty list we could use,

```
if (first == first->link)
{ . . . }
```

Figure 6.31 shows the pieces of C code for the various operations that can be done with the circular lists (for complete programs see the Exercises).

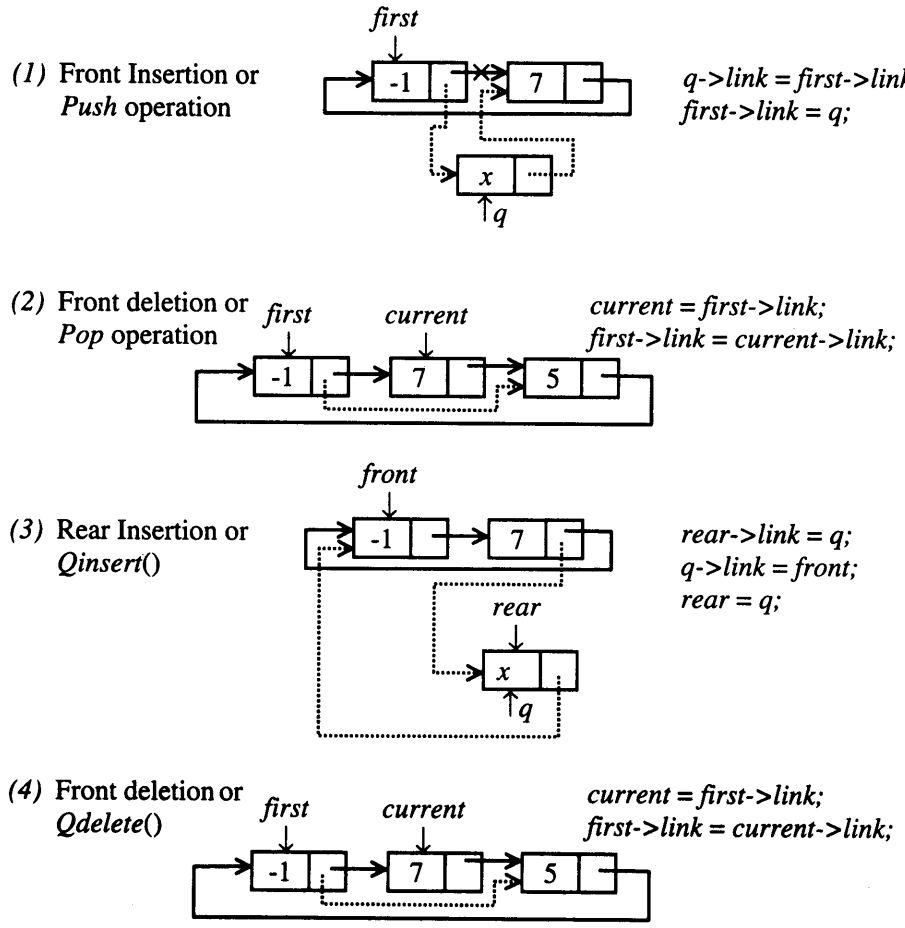



**Fig. 6.31 Various operations of Circular list with a header node**

The other operations will be discussed in Section 6.19.2.

## 6.18 DOUBLY LINKED LIST

There is no restriction imposed on the number of links that a node can have. However, remember that each link is a pointer variable occupying 2 bytes of storage space per node. Therefore, if we have $n$-node list, the storage space needed is $2n$ bytes only for the link fields. In an ordinary singly linked list and a circular list we have only one link for connecting the nodes. Unfortunately, the chain is one way- this means you can move from left to right and not from right to left.

A list consisting of $n$ nodes, each having two link fields is called a **doubly linked list.** That is,

$$DL = \{e_1, e_2, \ldots, e_n\}$$

The link fields are called as **left link** and a **right link.** The left link of $i$th node connects to the $(i - 1)$th node and the right link points to the $(i + 1)$th node (see Figure 6.32).

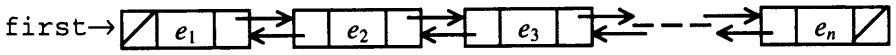



**Fig. 6.32 Doubly linked list with $n$ nodes**

Notice that, the left link of $e_1$, is NULL as it is the first node and no other nodes exist prior to it. Similarly, the right link of $e_n$ is NULL, as it is the last node and no other nodes exist after this node. Using right links, it is very straightforward to scan the doubly linked list from left to right. Also, using left links, you can move from right to left.

The node structure of singly linked cannot be used for a doubly linked list and so the modified node structure is defined below:

```
struct  DList
{
   int info;
   struct DList *left;
   struct DList *right;
};
typedef struct DList *DNODE;
```

For all our future discussions, we follow this definition and each node will be of type DNODE. This is to differentiate with respect to the singly linked list structure called NODE.

### 6.18.1 Creation of a Doubly Linked List: Create() function

Firstly let us present a method to build a doubly linked list by inserting nodes in front of `first`, where `first` is the doubly linked pointer to the first node. Unlike, singly

linked list construction, here we need to distinguish between first node creation and subsequent node creations.

Let us assume a sample list of 3 elements and show how a new nodes x can be added to the front of the first element (see Figure 6.33). The Figure 6.33 clearly shows the steps required.

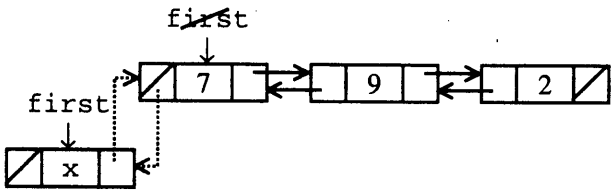



**Fig. 6.33 Front Insertion**

to add element x. For the convenience of the readers, we shall explicitly write those steps as,

```
q->right = first;
first->left = q;
first = q;
```

Assume that, the dynamic memory for q is available with info set to x and its left pointer initialized to NULL. The complete C function Create() appears in Program 6.26.

---

***Program 6.26***
***Creating a Doubly linked list***

---

```
DNODE Create (DNODE first, int x)
{
     DNODE q;
     q = getnode();
     q->info = x;
     q->left =  NULL;
     q->right = NULL;
     if (!first) /* first is NULL */
     {
          /* creation of the first node */
          first = q;
          return first;
     }
     q->right = first;
     first->left = q;
     first = q;
     return first;
}
```

---

The program first creates the new node q and makes the initialization. If we are creating the first node, then `first` points to q and the current node is returned. When the list not empty, the code shown before is used and updated list is returned to the calling program.

## 6.18.2 Insert a new node before a key node: Insert() function

The requirement in this problem is that in a given doubly linked list, we need to find a key element. If the key node is found, insert the new node (x) to the front of this node and return the modified list, else return the list to the calling program unaltered.

Let us consider a sample list with four elements and show all possible cases. The cases include the key being the first node, middle node and the last node. A typical situation is shown in Figure 6.34, when *key* = 7 and *key* = 2.

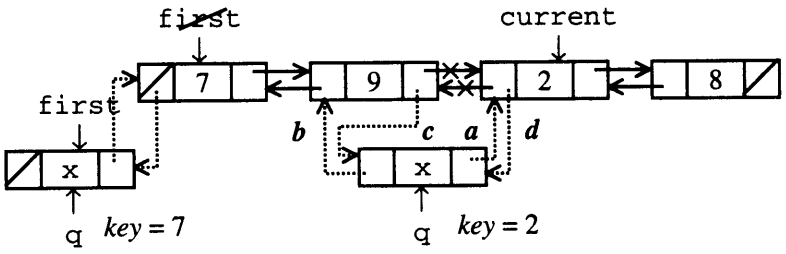



**Fig. 6.34 Insert x front of key node when *key* = 7 and *key* = 2**

If *key* = 7, the code required is different from when *key* = 2. However both the cases are shown in a single figure. For these two cases (if the code works for these two cases then it will work for any key value)

*key* = 7 :
```
q->right = first;
first->left = q;
first = q;
```

Assume that the key is pointed by current (this is done by establishing a loop), the code immediately follows in the order specified as a, b, c and d (see the Figure).

***a*** `q->right = current`
***b*** `q->left = current->left;`
***c*** `current->left->right = q;`
***d*** `current->left = q;`

The operations ***a*** and ***b*** connects the new node q with current and its predecessor. Next, ***c*** and ***d*** completes the rest by breaking the old links. Observe the ordering used in linking the new node with the list. Program 6.27 gives a C code to accomplish the insert operation.

*Program 6.27*
*Insertion*

```
DNODE Insert (DNODE first, int x, int key)
{  /* Insert element x before the key element */
     DNODE current, q;
     for (current = first;
          current && (current->info != key);
               current = current->right);
     if (!current) /* current is empty */
     {
          printf("The Key node not Found!\n");
          return first;
     }
     q = getnode();
     q->info = x;
     if (current == first)  /* first node is the key */
     {
          q->left = NULL;
          q->right = current;
          current->left = q;
          first = q;
          return first;
     }
     /* key is middle node, insert before current */
     q->right = current;
     q->left = current->left;
     current->left->right = q;
     current->left = q;

     return first;
}
```

## 6.18.3 Delete the node whose info is given: LDelete()

Often it is required to remove an element from the list based upon the `info` filed. For example, the employee records of an organization may be stored in a doubly linked list. If an employee leaves the organization, his/her record should be deleted from the list. The parameters required for this function are:

(1) `first` – pointer to the list.
(2) `x` – key element to be deleted.

(3) `flag` – {1 - successful deletion; 0 - key not found}

The function can then be invoked as `LDelete(first, x, flag)`. Since, the list is modified because of deletion, the return type should be `DNODE`. Before we present the program code, the logic for deletion can be shown as a diagram - see Figure 6.35.

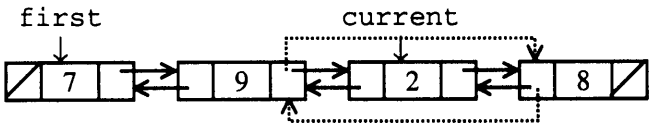



**Fig. 6.35 Delete node 2**

Once, we locate the node to be deleted, pointed by `current`, we can use the following piece of code to delete the node pointed by `current`.

```
current->left->right = current->right;
current->right->left = current->left;
```

For linking node 9 and node 8, the `current->right` should point to node 8. Similarly, the left pointer of node 8 should point to node 9. To access the predecessor and successor of the current node to be deleted can be accessed as just shown. This is the advantage of having a left and right pointer. This code is applicable for only middle node deletion and won't work, when the key is first node. See the Program 6.28 for more details and for all cases.

---

*Program 6.28*
*Deleting a node*

---

```
DNODE LDelete (DNODE first, int x, int *flag)
{  /* delete x and set flag = 1 for successful deletion */
   /* and 0 for unsuccessful deletion                    */
    DNODE current;
    *flag = 1;
    for (current = first; current && (current->info != x);
                    current = current->right);
    if (!current)
    { *flag = 0; return first; }
    /* first node is key node and the only node */
    if (current == first && current->right == NULL)
    { first = 0; return first; }

    if (current == first)
    {
         first = first->right;/* first one is key node */
         first->left = 0;
```

```
    }
    else
    {
        current->left->right = current->right;
        current->right->left = current->left;
    }
    free(current);
    return first;
}
```

Whenever 0 is assigned to a link address it means a NULL value, because the value of NULL is 0. Recall that the definition is already made in *stdio.h*.

The other operations of doubly linked list like Search, Find, Length, etc. are left as exercises.

## 6.19 APPLICATIONS

The linked lists, in general, are useful in solving many problems. In this section we pick some of the very important applications and provide steps to understand the solution process. All the applications that we discuss will be provided with appropriate C source code. The applications that we discuss are,

- Polynomial Addition using singly linked lists.
- Adding two long positive integers using circular lists.

### 6.19.1 Polynomial Addition using a singly linked list

A **univariate polynomial** of degree $d$ has the form,

$$c_d x^d + c_{d-1}. x^{d-1} + c_{d-2}. x^{d-2} + \ldots + c_0$$

where, $c_d \neq 0$. The $c_i$s are the coefficients and the $e_i$s are the exponents. By definition the exponents are nonnegative integers.

Each $c_i x^i$ is a term of the polynomial. For example,

$$A(x) = 5x^3 + 17x^2 - 2x \quad \ldots.(6.1)$$

In this polynomial $5x^3$, $17x^2$ and $-2x$ are the three terms and this polynomial is a single variable, i.e., $x$. A Polynomial with two variables is as follows,

$$B(x, y) = 7xy^2 + 2xy + y^2x + 10 \quad \ldots(6.2)$$

Each variable $x$ and $y$ will have the exponents and a coefficient. For instance, in the first term of $B(x, y)$ 7 is the coefficient, the exponent of $x$ is 1 and the exponent of $y$ is 2.

## Representation

We wish to develop a linked list node structure in C language to store each term in the polynomial. The template for the term will consist of three *info* fields and a *link* field as shown in Figure 6.36.

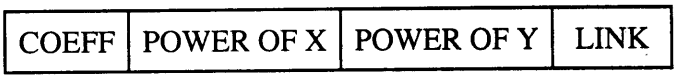
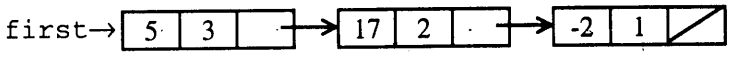
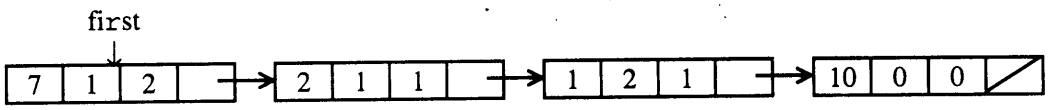



**Fig. 6.36 A polynomial Term with 2 variables**

For a single variable, there will be only one exponent filed. The polynomials of Equations (6.1) and (6.2) are shown as linked lists in Figure 6.37.




**(a) Linked list for $A(x) = 5x^3 + 17x^2 - 2x$**




**(b) Linked list for $B(x, y) = 7xy^2 + 2xy + y^2x + 10$**

**Fig. 6.37 Polynomials as linked lists**

## Additions of Two Polynomials

The objective of this section is to develop an appropriate algorithm to represent and manipulate polynomials. Then, we shall develop a C program for the same. Let us consider only the basic manipulation – to add two polynomials and the rest are covered in exercises.

To enable us the addition process easier, we must store the polynomial to be a sequence of numbers of the form,

**(i) for single variable:**

$$c_1 e_1, c_2 e_2, c_3 e_3 \ldots c_n e_n$$

where, $e_i$ is the exponent and $c_i$ is the coefficient.

$$e_1 > e_2 > e_3 \ldots > e_n$$

This means that the power of $x_s$ are assumed to be in descending order.

**(ii) for two variables:**

$$c_1 e_1 d_1, c_2 e_2 d_2 \ldots c_n e_n d_n$$

where, $e_i$ is the power of $x$, $d_i$ is the power of $y$ and $c_i$ is the coefficient.

$$e_1 d_1 > e_2 d_2 > e_2 d_2 > \ldots > e_n d_n$$

This means that the power of $x$ in $e_1$ is greater than power of $x$ in $e_2$. Suppose if the powers are same in $e_1$, then $d_1 > d_2$. That is, power of $y$ is $d_1$ should be greater than the power of $y$ in $d_2$.

An example will clarify the above said fact. Consider the polynomial,

$$x + 8y^2 + 7xy^2 - 9y + 15$$

This polynomial is not in a descending order, hence by following, the above method we have

$$7xy^2 + x + 8y^2 - 9y + 15$$

The descending order of exponents of $x$ and $y$ are shown below.

| *Coeff* | *pow - x* | *pow – y* |
|---|---|---|
| 7 | 1 | 2 |
| 1 | 1 | 0 |
| 8 | 0 | 2 |
| -9 | 0 | 1 |
| 15 | 0 | 0 |

Notice that in terms $7xy^2$ and $x$, power of $x$ is same. Which one should occur first? In these circumstances, check the power of $y$. $7xy^2$ is greater than $x$, because in the second term, the power of $y$ is zero. Hence, $7xy^2$ precedes $x$.

In general, the term $c_1 e_1 d_1$ will precede $c_2 e_2 d_2$ provided,

$e_1 > e_2$

or $(e_1 = e_2)$ and $(d_1 > d_2)$

or $(e_1 = e_2)$ and $(d_1 = d_2)$.

Once the two polynomials are stored in the descending order of the powers of $x$ and/or $y$, addition becomes straightforward and the corresponding algorithm is shown in Figure 6.38.

```
Algorithm PolyAdd (p,q)
{
     // add two polynomials pointed by p and q
     // the resultant polynomial to be returned through r.
     while (p && q)
     {
          // pterm = (c1, e1,d1) and qterm =(c2,e2,d2)
          // info includes the power of x and y plus the
          // coefficients.
          pterm = info(p);
          qterm = info(q);
```

```
            if(e1 = e2 && d1 = d2)  // same powers
                  if(c1 + c2 != 0)  // terms whose sum in zero
                                    // need not be added to r.
            {
                  pterm = copy e1,d1,(c1+c2)
                  // put (c1 + c2),e1 and d1 to r.
                  PolyLast(pterm);
                  p = link(p);      // advance to next term.
                  q = link(q);
            }
            else if((e1 > e2) OR (e1 =e2 AND d1 > d2))
            {
                  r = PolyLast(pterm);
                  p = link(p);
            }
            else
            {
                  r = PolyLast(qterm);
                  q = link(q);
            }
      }

      // copy remaining elements of p, if any
      while (P != NULL)
      {
            r = PolyLast(pterm);
            p = link(p);
      }

      // copy remaining elements of p, if any
      while (q != NULL)
      {
            r = PolyLast(qterm);
            q = link(q);
      }
      return r;
}
```

**Fig. 6.38 Algorithm for polynomial addition**

Building the polynomial using a linked list in the descending order of powers of $x$ and/or $y$ is another big task. The knowledge of ordered linked list should be helpful. The complete source code for polynomial addition is given in chapter.10. Here, we shall show only `PolyAdd()` and `PolyLast()` functions in Program 6.29.

*Program 6.29*
*Addition of two Polynomials*

```
NODE PolyAdd(NODE p, NODE q)
{
     NODE r = NULL;
     int sum;
     while (p && q)
     {
          sum = p->coeff + q->coeff;
          if ((p->px == q->px) && (p->py == q->py))
               if (sum != 0)
               {
                    r = PolyLast(r, p->px, p->py, sum);
                    p = p->link;
                    q = q->link;
               }
               else
               {
                    p = p->link;
                    q = q->link;
               }
          else if ((p->px > q->px)
                    ||(p->px == q->px && p->py > q->py)
                    ||(p->px == q->px)
                    && (p->py == q->py))
               {
                    r = PolyLast(r,p->px,p->py,p->coeff);
                    p = p->link;
               }
               else
               {
                    r = PolyLast(r,q->px,q->py,q->coeff);
                    q = q->link;
               }
     }
     while (p)
     {
          r = PolyLast(r, p->px, p->py, p->coeff);
          p = p->link;
     }
     while (q)
     {
          r = PolyLast(r, q->px, q->py, q->coeff);
          q = q->link;
```

```
    }
    return r;
}

NODE PolyLast(NODE p, int fx, int fy, int fc)
{
    NODE q;
    q = (NODE) malloc (sizeof(struct List));
    q->px = fx;
    q->py = fy;
    q->coeff = fc;
    q->link = NULL;
    if (!p)
    {
        p = q;
        last = p;
        return p;
    }
    last->link = q;
    last = q;
    return p;
}
```

---

The function `PolyAdd()` is written based on the algorithm just shown. It uses another function `PolyLast()` whose purpose is to add the terms to the end of resultant polynomial `r`. A sample execution of the program appears below:

*Example Run*

```
Enter your Choice: 3
  7x^1 y^2 + 1x^1 y^0 + 8x^0 y^2   -9x^0 y^1 + 15x^0 y^0
        1 Poly1 Insert
        2 Poly2 Insert
        3 Display Poly1
        4 Display Poly2
        5 PolyAdd
        6 Exit
Enter your Choice: 4
  5x^2 y^2 + 8x^2 y^1 + 10x^0 y^2 + 9x^0 y^1 + 3x^0 y^0
        1 Poly1 Insert
        2 Poly2 Insert
        3 Display Poly1
        4 Display Poly2
        5 PolyAdd
```

```
                6 Exit
        Enter your Choice: 5
5x^2 y^2 + 8x^2 y^1 + 7x^1 y^2 + 1x^1 y^0 + 18x^0 y^2 + 18x^0 y^0
```

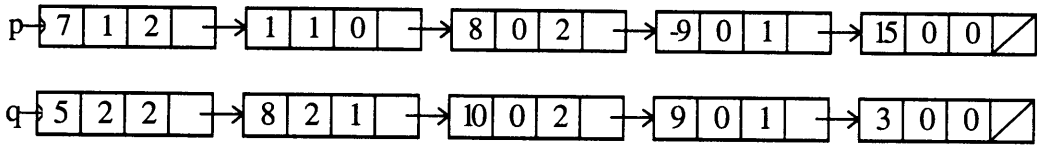
*Example*

To validate the working of `PolyAdd()`, let us consider two polynomials and write the steps in tracing it.

Poly1 = $x + 8y^2 + 7xy^2 - 9y + 15$
Poly2 = $8x^2y + 5x^2y^2 + 9y + 10y^2 + 3$

The input function stores these two polynomials in the descending order and is shown in Figure 6.39.

p = $7xy^2 + x + 8y^2 - 9y + 15$
q = $5x^2y^2 + 8x^2y + 10y^2 + 9y + 3$




| Sl. No | Terms considered | | Term to be added | Result r |
|---|---|---|---|---|
| | p | q | | |
| 1. | [7, 1, 2] | [5, 2, 2] | [5, 2, 2] | [5, 2, 2] |
| 2. | [7, 1, 2] | [8, 2, 1] | [8, 2, 1] | [5, 2, 2], [8, 2, 1] |
| 3. | [7, 1, 2] | [10, 0, 2] | [7, 1, 2] | [5, 2, 2], [8, 2, 1], [7, 1, 2] |
| 4. | [1, 1, 0] | [10, 0, 2] | [1, 1, 0] | [5, 2, 2], [8, 2, 1], [7, 1, 2] [1, 1, 0] |
| 5. | [8, 0, 2] | [10, 0, 2] | [18, 0, 2] | [5, 2, 2], [8, 2, 1], [7, 1, 2],[1, 1, 0], [18, 0, 2] |
| 6. | [-9, 0, 1] | [9, 0, 1] | - | [5, 2, 2], [8, 2, 1], [7, 1, 2],[1, 1, 0], [18, 0, 2] |
| 7. | [15, 0, 0] | [3, 0, 0] | [18, 0, 0] | [5, 2, 2], [8, 2, 1], [7, 1, 2],[1, 1, 0], [18, 0, 2], [18, 0, 0] |

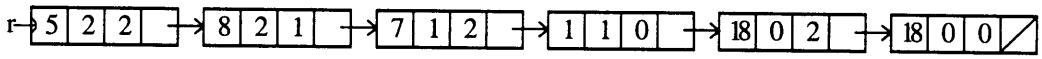



**Fig. 6.39 Snapshot of Polynomial addition**

The resultant polynomial `r` is,

r = $5x^2y^2 + 8x^2y + 7xy^2 + x + 18y^2 + 18$

The group of elements represented in angle brackets [and ] are the [*coeff*, *power of x*, *power of y*]. In serial number 6, the addition of coefficients with the same powers yields