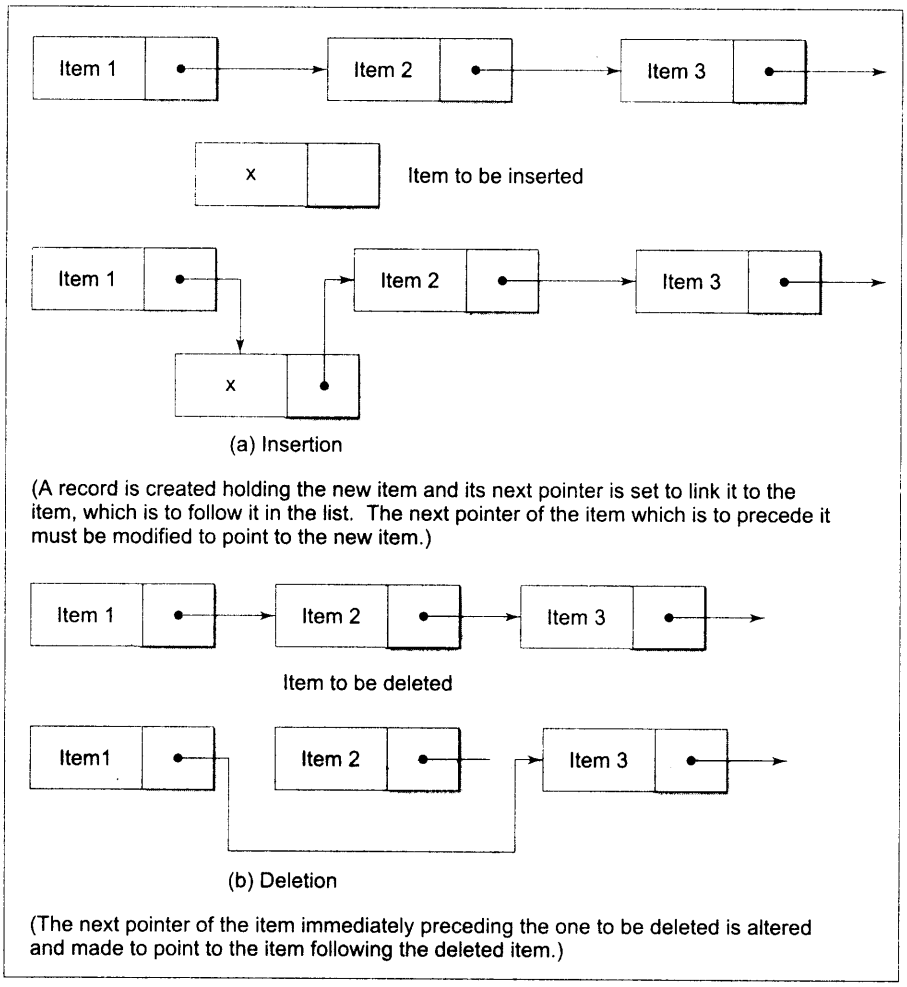


Fig. 13.5 *Insertion into and deletion from a linked list*

13.9 TYPES OF LINKED LISTS

There are different types of lined lists. The one we discussed so far is known as *linear singly* linked list. The other linked lists are:

- Circular linked lists
- Two-way or doubly linked lists
- Circular doubly linked lists

The circular linked lists have no beginning and no end. The last item points back to the first item. The doubly linked list uses double set of pointers, one pointing to the next item and other pointing to the preceding item. This allows us to traverse the list in either direction. Circular doubly linked lists employs both the forward pointer and backward pointer in circular form. Figure 13.6 illustrates various kinds of linked lists.

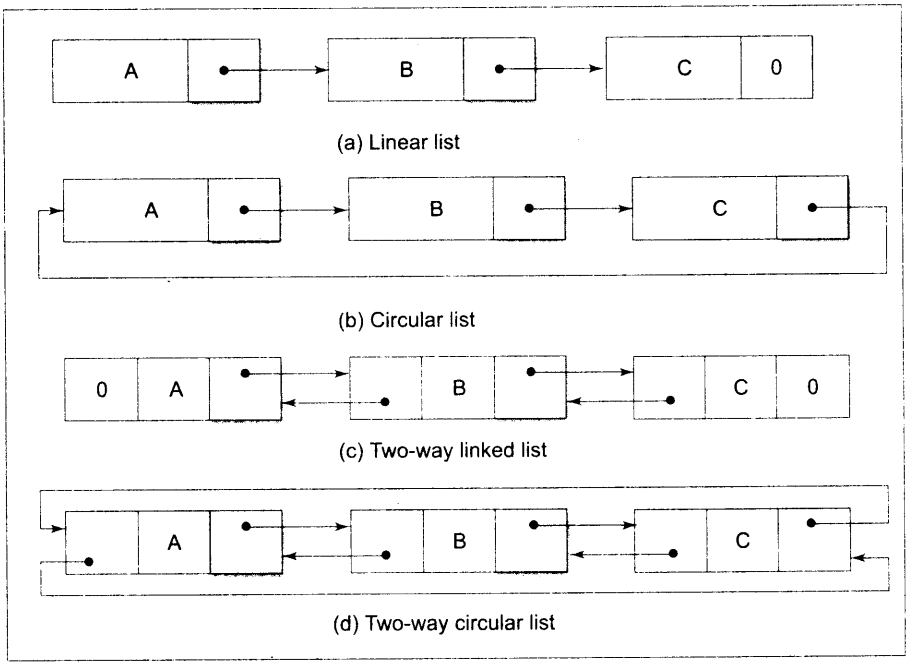


Fig. 13.6 *Different types of linked lists*

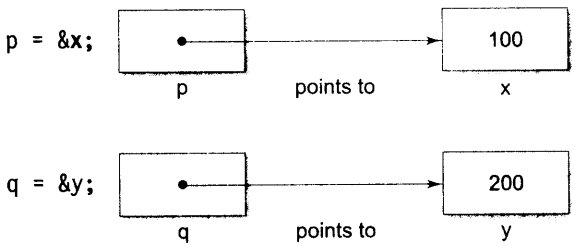
13.10 POINTERS REVISITED

The concept of pointers was discussed in Chapter 11. Since pointers are used extensively in processing of the linked lists, we shall briefly review some of their properties that are directly relevant to the processing of lists.

We know that variables can be declared as pointers, specifying the type of data item they can point to. In effect, the pointer will hold the address of the data item and can be used to access its value. In processing linked lists, we mostly use pointers of type structures.

It is most important to remember the distinction between the pointer variable **ptr**, which contain the address of a variable, and the referenced variable ***ptr**, which denotes the value of variable to which **ptr**'s value points. The following examples illustrate this distinction. In these illustrations, we assume that the pointers **p** and **q** and the variables **x** and **y** are declared to be of same type.

(a) Initialization

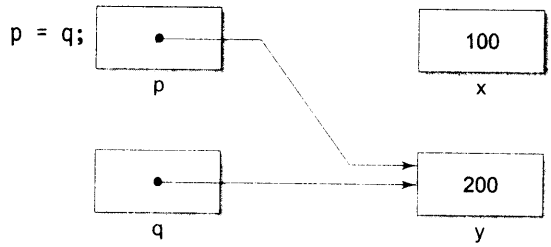


The pointer **p** contains the address of **x** and **q** contains the address of y.

***p =100 and *q = 200 and p<>q**

(b) Assignment p = q

The assignment **p = q** assigns the address of the variable **y** to the pointer variable **p** and therefore **p** now points to the variable y.

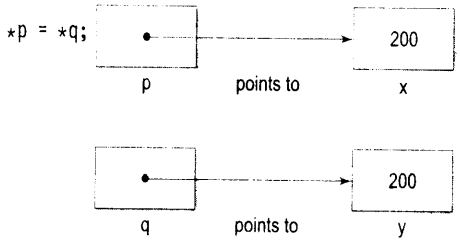


Both the pointer variables point to the same variable.

***p = *q = 200 but x <> y**

(c) Assignment *p = *q

This assignment statement puts the value of the variable pointed to by **q** in the location of the variable pointed to by **p**.



The pointer **p** still points to the same variable **x** but the old value of **x** is replaced by 200 (which is pointed to by **q**).

x = y = 200 but p <> q

(d) NULL pointers

A special constant known as NULL pointer (0) is available in C to initialize pointers that point to nothing. That is the statements

p = 0; (or p = NULL;) p → 0

q = 0; (q = NULL;) q → 0

make the pointers **p** and **q** point to nothing. They can be later used to point any values.

We know that a pointer must be initialized by assigning a memory address before using it. There are two ways of assigning memory address to a pointer.

1. Assigning an existing variable address (static assignment)

```
ptr = &count;
```

2. Using a memory allocation function (dynamic assignment)

```
ptr = (int*) malloc(sizeof(int));
```

13.11 CREATING A LINKED LIST

We can treat a linked list as an abstract data type and perform the following basic operations:

1. Creating a list
2. Traversing the list
3. Counting the items in the list
4. Printing the list (or sub list)
5. Looking up an item for editing or printing
6. Inserting an item
7. Deleting an item
8. Concatenating two lists

In Section 13.7 we created a two-element linked list using the structure variable names **node1** and **node2**. We also used the address operator **&** and member operators **.** and **–>** for creating and accessing individual items. The very idea of using a linked list is to avoid any reference to specific number of items in the list so that we can insert or delete items as and when necessary. This can be achieved by using "anonymous" locations to store nodes. Such locations are accessed not by name, but by means of pointers, which refer to them. (For example, we must avoid using references like **node1.age** and **node1.next –> age**.)

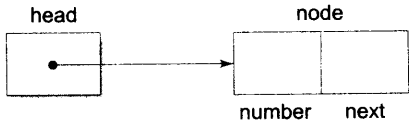
Anonymous locations are created using pointers and dynamic memory allocation functions such as **malloc**. We use a pointer **head** to create and access anonymous nodes. Consider the following:

```
struct linked_list
{
    int number;
    struct linked_list *next;
};
typedef struct linked_list node;
node *head;
head = (node *) malloc(sizeof(node));
```

The **struct** declaration merely describes the format of the nodes and does not allocate storage. Storage space for a node is created only when the function **malloc** is called in the statement

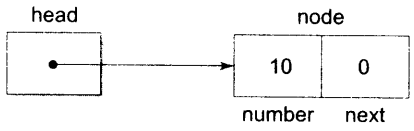
```
head = (node *) malloc(sizeof(node));
```

This statement obtains a piece of memory that is sufficient to store a node and assigns its address to the pointer variable **head**. This pointer indicates the beginning of the linked list.



The following statements store values in the member fields:

```
head -> number = 10;
head -> next = NULL;
```



The second node can be added as follows:

```
head -> next = (node *)malloc(sizeof(node));
head -> next ->number = 20;
head->next->next = NULL;
```

Although this process can be continued to create any number of nodes, it becomes cumbersome and clumsy if nodes are more than two. The above process may be easily implemented using both recursion and iteration techniques. The pointer can be moved from the current node to the next node by a self-replacement statement such as

```
head = head -> next;
```

The Example 13.3 shows creation of a complete linked list and printing of its contents using recursion.

Example 13.3 Write a program to create a linear linked list interactively and print out the list and the total number of items in the list.

The program shown in Fig. 13.7 first allocates a block of memory dynamically for the first node using the statement

```
head = (node *)malloc(sizeof(node));
```

which returns a pointer to a structure of type **node** that has been type defined earlier. The linked list is then created by the function **create**. The function requests for the number to be placed in the current node that has been created. If the value assigned to the current node is –999, then null is assigned to the pointer variable **next** and the list ends. Otherwise, memory space is allocated to the next node using again the **malloc** function and the next value is placed into it. Not that the function **create** calls itself recursively and the process will continue until we enter the number –999.

The items stored in the linked list are printed using the function **print** which accept a pointer to the current node as an argument. It is a recursive function and stops when it receives a NULL pointer. Printing algorithm is as follows;

1. Start with the first node.
2. While there are valid nodes left to print
   (a) print the current item and
   (b) advance to next node

Similarly, the function **count** counts the number of items in the list recursively and return the total number of items to the **main** function. Note that the counting does not include the item –999 (contained in the dummy node).

Program

```
#include  <stdio.h>
#include  <stdlib.h>
#define   NULL 0

struct linked_list
{
            int number;
            struct linked_list *next;
};
typedef struct linked_list node;  /* node type defined */

main()
{
            node *head;
            void create(node *p);
            int count(node *p);
            void print(node *p);
            head = (node *)malloc(sizeof(node));
            create(head);
            printf("\n");
            printf(head);
            printf("\n");
            printf("\nNumber of items = %d \n", count(head));
}
void create(node *list)
{
            printf("Input a number\n");
            printf("(type -999 at end): ");
            scanf("%d", &list -> number); /* create current node */

            if(list->number == -999)
            {
              list->next = NULL;
            }
            else /*create next node */
            {
              list->next = (node *)malloc(sizeof(node));
              create(list->next);  */ Recursion occurs */
            }
            return;
}
void print(node *list)
{
              if(list->next != NULL)
```

```
    {
      printf("%d--->",list ->number);  /* print current item */

       if(list->next->next == NULL)
         printf("%d", list->next->number);

      print(list->next);   /* move to next item */
    }
    return;
}

int count(node *list)
{
  if(list->next == NULL)
         return (0);
  else
         return(1+ count(list->next));
}
```

Output

```
Input a number
(type -999 to end); 60
Input a number
(type -999 to end); 20
Input a number
(type -999 to end); 10
Input a number
(type -999 to end); 40
Input a number
(type -999 to end); 30
Input a number
(type -999 to end); 50
Input a number
(type -999 to end); -999

60 -->20 -->10 -->40 -->30 -->50 --> -999

Number of items = 6
```

Fig. 13.7 *Creating a linear linked list*

13.12 INSERTING AN ITEM

One of the advantages of linked lists is the comparative case with which new nodes can be inserted. It requires merely resetting of two pointers (rather than having to move around a list of data as would be the case with arrays).

Inserting a new item, say X, into the list has three situations:

1. Insertion at the front of the list.
2. Insertion in the middle of the list.
3. Insertion at the end of the list.

The process of insertion precedes a search for the place of insertion. The search involves in locating a node after which the new item is to be inserted.

A general algorithm for insertion is as follows:

Begin
  if the list is empty or
  the new node comes before the head node *then*,
  insert the new node as the head node,
  else
  if the new node comes after the last node, *then*,
  insert the new node as the end node,
  else
  insert the new node in the body of the list.
End

Algorithm for placing the new item at the beginning of a linked list:

1. Obtain space for new node.
2. Assign data to the item field of new node.
3. Set the *next* field of the new node to point to the start of the list.
4. Change the head pointer to point to the new node.

Algorithm for inserting the new node X between two existing nodes, say, N1 and N2;

1. Set space for new node X.
2. Assign value to the item field of X.
3. Set the *next* field of X to point to node N2.
4. Set the *next* field of N1 to point to X.

Algorithm for inserting an item at the end of the list is similar to the one for inserting in the middle, except the *next* field of the new node is set to NULL (or set to point to a dummy or sentinel node, if it exists).

Example 13.4 Write a function to insert a given item *before* a specified node known as key node.

The function **insert** shown in Fig. 13.8 requests for the item to be inserted as well as the "key node". If the insertion happens to be at the beginning, then memory space is created for the new node, the value of new item is assigned to it and the pointer **head** is assigned to the next member. The pointer **new** which indicates the beginning of the new node is assigned to **head.** Note the following statements:

```
new->number = x;
new->next = head;
head = new;
```

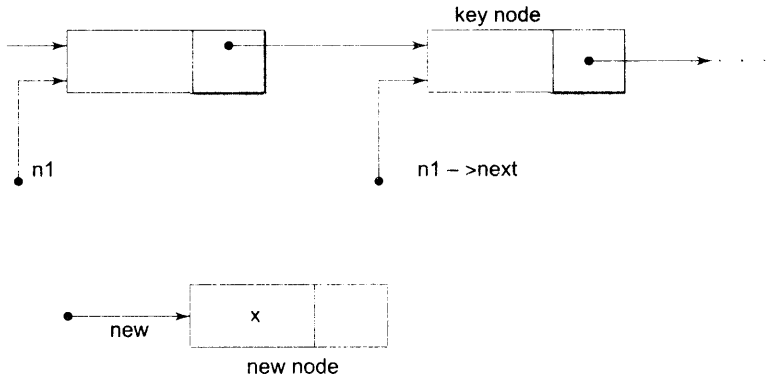
```
node *insert(node *head)
{
            node *find(node *p, int a);
            node *new;    /* pointer to new node */
            node *n1;   /* pointer to node preceding key node */
            int key;
            int x;      /* new item (number) to be inserted */

            printf("Value of new item?");
            scanf("%d", &x);
            printf("Value of key item ? (type -999 if last) ");
            scanf("%d", &key);

            if(head->number == key)  /* new node is first */
            {
                new = (node *)malloc(size of(node));
                new->number = x;
                new->next = head;
                head = new;
            }
            else   /* find key node and insert new node */
            {      /* before the key node */
              n1 = find(head, key);    /* find key node */

              if(n1 == NULL)
                 printf("\n key is not found \n");
              else   /* insert new node */
              {
                  new = (node *)malloc(sizeof(node));
                  new->number = x;
                  new->next = n1->next;
                  n1->next = new;
              }
            }
            return(head);
}
node *find(node *lists, int key)
{
              if(list->next->number == key) /* key found */
                return(list);
            else

              if(list->next->next == NULL)  /* end */
                return(NULL);
            else
              find(list->next, key);
}
```

Fig. 13.8 *A function for inserting an item into a linked list*

However, if the new item is to be inserted after an existing node, then we use the function **find** recursively to locate the 'key node'. The new item is inserted before the key node using the algorithm discussed above. This is illustrated as:

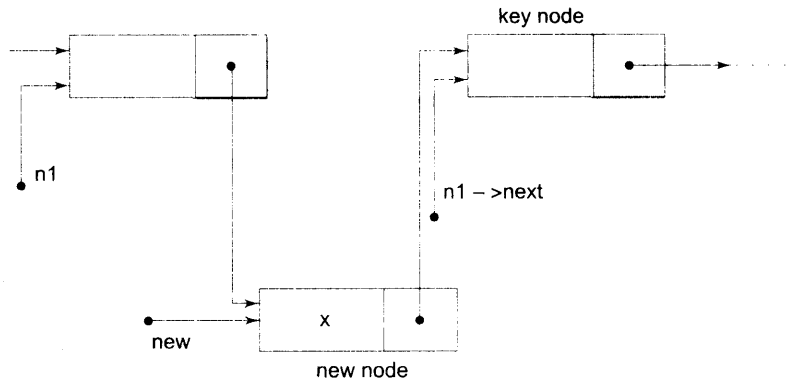
Before insertion

```
new = (node *)malloc(sizeof(node));
new->number = x;
```



After insertion

```
new->next = n1->next;
n1->next = new;
```



13.13 DELETING AN ITEM

Deleting a node from the list is even easier than insertion, as only one pointer value needs to be changed. Here again we have three situations.

1. Deleting the first item
2. Deleting the last item
3. Deleting between two nodes in the middle of the list

In the first case, the head pointer is altered to point to the second item in the list. In the other two cases, the pointer of the item immediately preceding the one to be deleted is altered to point to the item following the deleted item. The general algorithm for deletion is as follows:

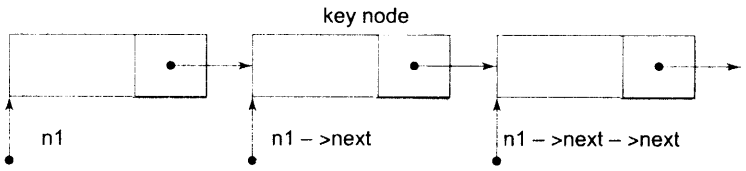
Begin
if the list is empty, *then*,
node cannot be deleted
else
if node to be deleted is the first node, *then*,
make the head to point to the second node,
else
delete the node from the body of the list.
End

The memory space of deleted node may be released for re-use. As in the case of insertion, the process of deletion also involves search for the item to be deleted.

Example 13.5 Write a function to delete a specified node.

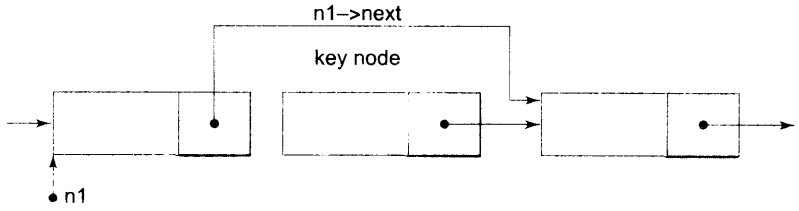
A function to delete a specified node is given in Fig. 13.9. The function first checks whether the specified item belongs to the first node. If yes, then the pointer to the second node is temporarily assigned the pointer variable **p**, the memory space occupied by the first node is freed and the location of the second node is assigned to **head**. Thus, the previous second node becomes the first node of the new list.

If the item to be deleted is not the first one, then we use the **find** function to locate the position of 'key node' containing the item to be deleted. The pointers are interchanged with the help of a temporary pointer variable making the pointer in the preceding node to point to the node following the key node. The memory space of key node that has been deleted if freed. The figure below shows the relative position of the key node.



The execution of the following code deletes the key node.

```
p = n1->next->next;
free (n1->next);
n1->next = p;
```



```
node *delete(node *head)
{
            node *find(node *p, int a);
            int  key;   /* item to be deleted */
            node *n1;   /* pointer to node preceding key node */
            node *p;    /* temporary pointer */
            printf("\n What is the item (number) to be deleted?");
            scanf("%d", &key);
            if(head->number == key)/* first node to be deleted) */
            {
              p = head->next;    /* pointer to 2nd node  in list */
              free(head);        /* release space of key node */
              head = p;          /* make head to point to 1st node */
            }
            else
            {
              n1 = find(head, key);
              if(n1 == NULL)
                 printf("\n key not found \n");
              else                        /* delete key node */
              {
                p = n1->next->next;     /*  pointer to the node
                                            following the keynode */

                free(n1->next);         /* free key node */
                n1->next = p;           /* establish link */
              }
            }
            return(head);
}
               /* USE FUNCTION find() HERE */
```

Fig. 13.9 *A function for deleting an item from linked list*

13.14 APPLICATION OF LINKED LISTS

Linked list concepts are useful to model many different abstract data types such as queues, stacks and trees.

If we restrict the process of insertion to one end of the list and deletions to the other end, then we have a model of a *queue*. That is, we can insert an item at the rear and remove an item at the front (see Fig. 13.10a). This obeys the discipline of "first in, first out" (FIFO). There are many examples of queues in real-life applications.

If we restrict insertions and deletions to occur only at one end of .lst, the beginning, then we model another data structure known as *stack*. Stacks are also referred to as *push-down* lists. An example of a stack is the "in" tray of a busy executive. The files pile up in the tray, and whenever the executive

has time to clear the files, he takes it off from the top. That is, files are added at the top and removed from the top (see Fig. 13.10b). Stacks are sometimes referred to as "last in, first out" (LIFO) structure.

Lists, queues and stacks are all inherently one-dimensional. A *tree* represents a two-dimensional linked list. Trees are frequently encountered in everyday life. One example is the organizational chart of a large company. Another example is the chart of sports tournaments.

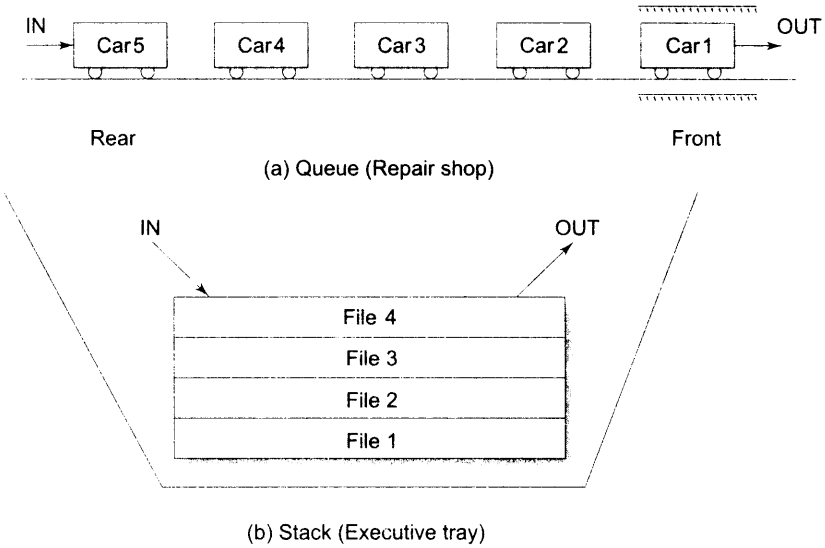


Fig. 13.10 *Application of linked lists*

Just Remember

- Use the **sizeof** operator to determine the size of a linked list.
- When using memory allocation functions **malloc** and **calloc,** test for a NULL pointer return value. Print appropriate message if the memory allocation fails.
- Never call memory allocation functions with a zero size.
- Release the dynamically allocated memory when it is no longer required to avoid any possible "memory leak".
- Using **free** function to release the memory not allocated dynamically with **malloc** or **calloc** is an error.
- Use of a invalid pointer with **free** may cause problems and, sometimes, system crash.
- Using a pointer after its memory has been released is an error.
- It is an error to assign the return value from **malloc** or **calloc** to anything other than a pointer.
- It is a logic error to set a pointer to NULL before the node has been released. The node is irretrievably lost.

- It is an error to declare a self-referential structure without a structure tag.
- It is an error to release individually the elements of an array created with **calloc**.
- It is a logic error to fail to set the link filed in the last node to null.

CASE STUDIES

1. Insertion in a Sorted List

The task of inserting a value into the current location in a sorted linked list involves two operations:

1. Finding the node before which the new node has to be inserted. We call this node as 'Key node'.
2. Creating a new node with the value to be inserted and inserting the new node by manipulating pointers appropriately.

In order to illustrate the process of insertion, we use a sorted linked list created by the create function discussed in Example 13.3. Figure 13.11 shows a complete program that creates a list (using sorted input data) and then inserts a given value into the correct place using function insert.

Program

```
#include <stdio.h>
#include <stdio.h>
#define NULL 0

struct linked_list
{
            int number;
            struct linked-list *next;
};
typedef struct linked_lit node;

main()
{
            int n;
            node *head;
            void create(node *p);
            node *insert(node *p, int n);
            void print(node *p);
            head = (node *)malloc(sizeof(node));
            create(head);
            printf("\n");
            printf("Original list: ");
            print(head);
            printf("\n\n");
            printf("Input number to be inserted: ");
            scanf("%d", &n);
```

```
            head = inert(head,n);
            printf("\n");
            printf("New list:  ");
            print(head);
}
void create(node *list)
{
            printf("Input a number \n");
            printf("(type -999 at end): ");
            scanf("%d", &list->number);

            if(list->number == -999)
            {
              list->next = NULL;
            }
            else /* create next node */
            {
              list->next = (node *)malloc(sizeof(node));
              create(list->next);
            }
            return;
}

void print(node *list)
{
            if(list->next != NULL)
            {
              printf("%d -->", list->number);

            if(list ->next->next = = NULL)
              printf("%d", list->next->number);

            print(list->next);
            }
            return;
}

node *insert(node *head, int x)
{
            node *p1, *p2, *p;
            p1 = NULL;
            p2 = head;  /* p2 points to first node */

            for( ; p2->number < x; p2 = p2->next)
            {
```

```
            p1 = p2;

            if(p2->next->next == NULL)
               {
                  p2 = p2->next;    /* insertion at end */
                  break;
               }
         }

         /*key node found and insert new node */

            p = (node )malloc(sizeof(node));  / space for new node */

            p->number = x; /* place value in the new node */

            p->next = p2; /*link new node to key node */

            if (p1 == NULL)
              head = p; /* new node becomes the first node */
            else
              p1->next = p; /* new node inserted in middle */

            return (head);
}
```

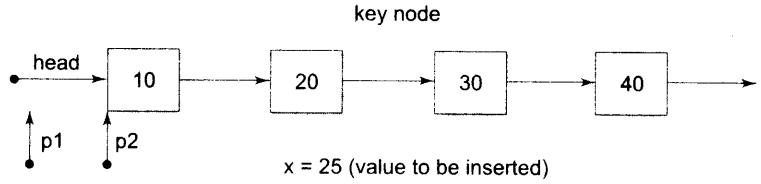
Output

```
  Input a number
  (type -999 at end ); 10

  Input a number
  (type -999 at end ); 20

  Input a number
  (type -999 at end ); 30

  Input a number
  (type -999 at end ); 40

  Input a number
  (type -999 at end ); -999

  Original list:  10 -->20-->30-->40-->-999

  Input number to be inserted: 25
  New list: 10-->20-->25-->30-->40-->-999
```

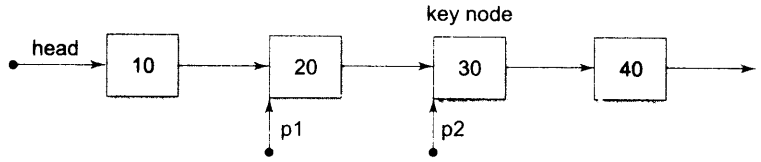
Fig. 13.11 *Inserting a number in a sorted linked list*

The function takes two arguments, one the value to be inserted and the other a pointer to the linked list. The function uses two pointers, **p1** and **p2** to search the list. Both the pointers are moved down the list with **p1** trailing **p2** by one node while the value **p2** points to is compared with the value to be inserted. The 'key node' is found when the number **p2** points to is greater (or equal) to the number to be inserted.

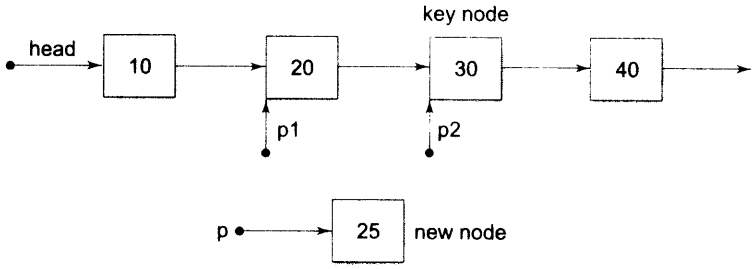
Once the key node is found, a new node containing the number is created and inserted between the nodes pointed to by **p1** and **p2**. The figures below illustrate the entire process.



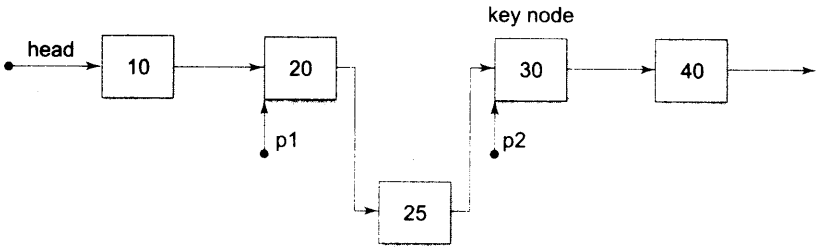
At the start of the search



When key node is found



When new node is created



When new node is inserted

2. Building a Sorted List

The program in Fig. 13.11 can be used to create a sorted list. This is possible by creating 'one item' list using the create function and then inserting the remaining items one after another using insert function.

A new program that would build a sorted list from a given list of numbers is shown in Fig. 13.12. The **main** function creates a 'base node' using the first number in the list and then calls the function **insert_sort** repeatedly to build the entire sorted list. It uses the same sorting algorithm discussed above but does not use any dummy node. Note that the last item points to NULL.

Program

```
#include <stdio.h>
#include <stdlib.h>
#define NULL 0

struct linked_list
{
        int number;
        struct linked_list *next;
};
typedef struct linked_list node;

main ()
{
        int n;
        node *head = NULL;
        void print(node *p);
        node *insert_Sort(node *p, int n);

        printf("Input the list of numbers.\n");
        printf("At end, type -999.\n");
        scanf("%d",&n);

        while(n != -999)
        {
          if(head == NULL)       /* create 'base' node */
          {
            head = (node *)malloc(sizeof(node));
            head ->number = n;
            head->next = NULL;

          }

          else       /* insert next item */
          {
```

```
                head = insert_sort(head,n);
              }
                scanf("%d", &n);
            }
            printf("\n");
            print(head);
            print("\n");
}
node *insert_sort(node *list, int x)
{
            node *p1, *p2, *p;
            p1 = NULL;
            p2 = list; /* p2 points to first node */

            for( ; p2->number < x ; p2 = p2->next)
            {
              p1 = p2;

              if(p2->next == NULL)
              {
                p2 = p2->next;          /* p2 set to NULL */
                break;          /* insert new node at end */
              }
            }

    /* key node found */
    p = (node *)malloc(sizeof(node)); /* space for new node */
    p->number = x;      /* place value in the new node */
    p->next = p2;       /* link new node to key node */
    if (p1 == NULL)
              list = p;       /* new node becomes the first node */
    else
              p1->next = p;  /* new node inserted after 1st node */


    return (list);
}
void print(node *list)
{
            if (list == NULL)
              printf("NULL");
            else
            {
              printf("%d-->",list->number);
              print(list->next);
```

```
            }
            return;
}

Output
        Input the list of number.
        At end, type -999.
        80 70 50 40 60 -999
        40-->50-->60-->70-->80 -->NULL
        Input the list of number.
        At end, type -999.
        40 70 50 60 80 -999
        40-->50-->60-->70-->80-->NULL
```

Fig. 13.12 *Creation of sorted list from a given list of numbers*

REVIEW QUESTIONS

13.1 State whether the following statements are *true* or *false*
- (a) Dynamically allocated memory can only be accessed using pointers.
- (b) **calloc** is used to change the memory allocation previously allocated with **malloc**.
- (c) Only one call to free is necessary to release an entire array allocated with **calloc**.
- (d) Memory should be freed when it is no longer required.
- (e) To ensure that it is released, allocated memory should be freed before the program ends.
- (f) The link field in a linked list always points to successor.
- (g) The first step in a adding a node to a linked list is to allocate memory for the next node.

13.2 Fill in the blanks in the following statements
- (a) Function _____ is used to dynamically allocate memory to arrays.
- (b) A_____ is an ordered collection of data in which each element contains the location of the next element.
- (c) Data structures which contain a member field that points to the same structure type are called _____ structures.
- (d) A_____identifies the last logical node in a linked list.
- (e) Stacks are referred to as _____

13.3 What is a linked list? How is it represented?

13.4 What is dynamic memory allocation? How does it help in building complex programs?

13.5 What is the principal difference between the functions **malloc** and **calloc**

13.6 Find errors, if any, in the following memory management statements:
- a. `*ptr = (int *)malloc(m, sizeof(int));`
- b. `table = (float *)calloc(100);`
- c. `node = free(ptr);`

13.7 Why a linked list is called a dynamic data structure? What are the advantages of using linked lists over arrays?

13.8 Describe different types of linked lists.

13.9 Identify errors, if any, in the following structure definition statements:

```
struct
{
  char name[30]
  struct *next;
};
typedef struct node;
```

13.10 The following code is defined in a header file *list.h*

```
typedef struct
{
     char name[15];
     int age;
     float weight;
}DATA;

struct linked_list
{
     DATA person;
     Struct linked_list *next;
};
typedef struct linked_list NODE;
typedef NODE *NDPTR;
```

Explain how could we use this header file for writing programs.

PROGRAMMING EXERCISES

13.1 In Example 13.3, we have used print() in recursive mode. Rewrite this function using iterative technique in for loop.

13.2 Write a menu driven program to create a linked list of a class of students and perform the following operations:

a. Write out the contents of the list.
b. Edit the details of a specified student.
c. Count the number of students above a specified age and weight.

Make use of the header file defined in Exercise 13.7.

13.3 Write recursive and non-recursive functions for reversing the elements in a linear list. Compare the relative efficiencies of them.

13.4 Write an interactive program to create linear linked lists of customer names and their telephone numbers. The program should be menu driven and include features for adding a new customer and deleting an existing customer.

13.5 Modify the above program so that the list is always maintained in the alphabetical order of customer names.

13.6 Develop a program to combine two sorted lists to produce a third sorted lists which contains one occurrence of each of the elements in the original lists.

13.7 Write a program to create a circular linked list so that the input order of data item is maintained. Add function to carry out the following operations on circular linked list.

a. Count the number of nodes
b. Write out contents
c. Locate and write the contents of a given node

13.8 Write a program to construct an ordered doubly linked list and write out the contents of a specified node.

13.9 Write a function that would traverse a linear singly linked list in reverse and write out the contents in reverse order.

13.10 Given two ordered singly linked lists, write a function that will merge them into a third ordered list.