Example 13.1
Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig.13.2. It tests for availability of memory space of required size. If it is available, then the required space is allocated and the address of the first byte of the space allocated is displayed. The program also illustrates the use of pointer variable for storing and accessing the table values.

USE OF **malloc** FUNCTION

Program

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

main()
{
   int *p, *table;
   int size;

   printf("\nWhat is the size of table?");
   scanf("%d",size);

   printf("\n")

      /*------------Memory allocation --------------*/

   if((table = (int*)malloc(size *sizeof(int))) == NULL)
     {
        printf("No space available \n");
        exit(1);
     }
   printf("\n Address of the first byte is  %u\n", table);

   /* Reading table values*/

   printf("\nInput table values\n");

   for (p=table; p<table + size; p++)
        scanf("%d",p);

   /* Printing table values in reverse order*/

   for (p = table + size -1; p >= table; p --)
      printf("%d is stored at address %u \n",*p,p);

}
```

Output

```
What is the size of the table? 5
Address of the first byte is 2262

Input table values
11 12 13 14 15

15 is stored at address 2270
14 is stored at address 2268
13 is stored at address 2266
12 is stored at address 2264
11 is stored at address 2262
```

Fig.13.2** Memory allocation with **malloc

Example 13.2

Write a program to store a character string in a block of memory space created by **malloc** and then modify the same to store a larger string.

The program is shown in Fig. 13.3. The output illustrates that the original buffer size obtained is modified to contain a larger string. Note that the original contents of the buffer remains same even after modification of the original size.

USE OF realloc AND free FUNCTIONS

Program

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

main()
{
  char *buffer;

  /* Allocating memory */

  if((buffer = (char *)malloc(10)) == NULL)
  {
     printf("malloc failed.\n");
     exit(1);
  }

  printf("Buffer of size %d created \n",_msize(buffer));
  strcpy(buffer, "HYDERABAD");
  printf("\nBuffer contains: %s \n ", buffer);

  /* Realloction */

  if((buffer = (char *)realloc(buffer, 15)) == NULL)
  {
     printf("Reallocation failed. \n");
     exit(1);
  }
  printf("\nBuffer size modified. \n");
  printf("\nBuffer still contains: %s \n",buffer);

  strcpy(buffer, "SECUNDERBAD");
  printf("\nBuffer now contains: %s \n",buffer);

/* Freeing memory */

free(buffer);
}
```

Output

```
Buffer of size 10 created
Buffer contains: HYDERABAD
Buffer size modified
Buffer still contains: HYDERABAD
Buffer now contains: SECUNDERABAD
```

Fig . 13.3 Reallocation and release of memory space

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).

CREATING A LINEAR LINKED LIST

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);
   }
   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);

        printf(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*

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;
```

FUNCTION INSERT

```
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*

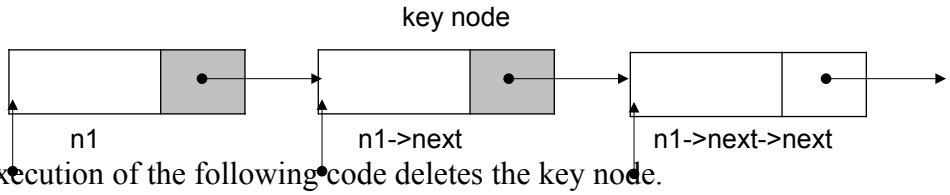
Example 13.5

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

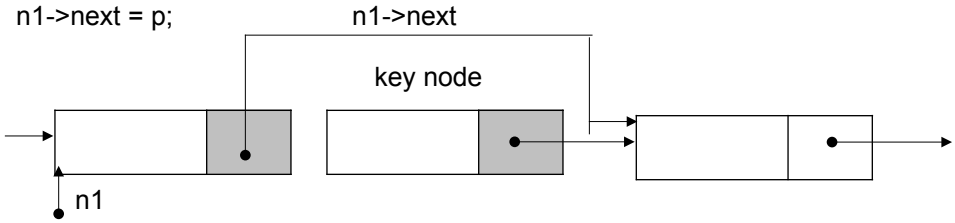
location of the second node is assigned to **head**. Thus, the previous second node becomes the first node of the new list.

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;
```



FUNCTION DELETE

```
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*