```
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: October 15, 2003
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th
 * http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Data Structures (2301263) classnote.
 * Description: This sample program demonstrates the use of singly linked
 *              list that arranges the input data (words) in alphabetic
 *              order.  Each node is dynamically created and inserted into
 *              the list where it belongs.
 */

#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>

#define         Normal          0
#define         Succeeded       1
#define         Failed          0
#define         OS_code1        1

#define         End_str         "FFFF"                  /* can be changed            */
#define         Size            100
#define         Huge            BUFSIZ
#define         Item            5


/*
 * global variables and definitions
 */
struct  rec
{
        char            name[Size];
        struct  rec     *next;
};

typedef struct  rec     REC;
typedef REC             *LINK;


/*
 * function prototypes
 */
void    free_all(LINK);
void    print_all(LINK);
int     processing(LINK *);
LINK    get_node(void);
LINK    traverse_list(LINK, char *);

/*
 * main processing
 * read input string (word) from keyboard; quit when 'End_str' is encountered.
 */
int
main(void)
{
        int     rcode = Normal;
        LINK    head  = NULL;

        if (processing(&head) == Failed)
        {
                printf("Out of memory\n");
                rcode = OS_code1;
        }
        print_all(head);
        free_all(head);
        return rcode;
}
```

```c
/*
 * p and q are temp pointers of type LINK.  q is used to point at the newly allocated
 * node REC.  p, on the other hand, is used to mark the node prior to the insertion point.
 * input:  head pointer of the list.
 * output: processing returned code (normal or error).
 */
int
processing(LINK *h)
{
       int     rt_code = Succeeded;
       LINK    p, q;
       char    tmp[Huge];

       p = q = NULL;
       printf("enter a word no longer than %d chars, type '%s' to quit\n",
               Size, End_str);
       (void)scanf("%s", tmp);
       while(strcmp(End_str, tmp) != 0)
       {
               if ((q = get_node()) == NULL)
               {
                       rt_code = Failed;
                       break;
               }
               /*
                * look for the position to insert new item into the list
                */
               if (*h == NULL)
                       *h = q;
               else
               {
                       p = traverse_list(*h, tmp);
                       if (p == NULL)                /* head of the list         */
                       {
                               q->next = *h;
                               *h      = q;
                       }
                       else                          /* somewhere along the list */
                       {
                               q->next = p->next;
                               p->next = q;
                       }
               }

               /*
                * copy the input value to the newly allocated node;
                * truncate if the string is longer than 'Size'
                */
               if (strlen(tmp) > Size-1)
               {
                       strncpy(q->name, tmp, Size-1);
                       q->name[Size] = '\0';
               }
               else
               {
                       strcpy(q->name, tmp);
               }
               (void)scanf("%s", tmp);
       }
       return rt_code;
}

/*
 * traverse the list from the head (start) node until finding the right position.
 * input:  head pointer and the input string.
 * output: pointer to the node just before the insertion point, NULL if head
 *        position is called for.
 */
LINK
traverse_list(LINK h, char *s)
{
       LINK    p, t;

       p = h;
       t = NULL;
       while (p != NULL)
       {
```

```
		if (strcmp(p->name, s) > 0)
			break;
		t = p;
		p = p->next;
	}
	return t;
}

/*
 * allocate a new node and return a pointer to that node.
 * input:  none
 * output: pointer to new node or NULL (if no space left on the system).
 */
LINK
get_node(void)
{
	LINK	hptr;

	hptr = (LINK)malloc(sizeof(REC));
	if (hptr == NULL)
		perror("Out of memory");
	else
		hptr->next = NULL;
	return hptr;
}

/*
 * free all the allotted memories.
 * input: head pointer to the list.
 */
void
free_all(LINK h)
{
	LINK	p, q;

	p = q = h;
	while (p != NULL)
	{
		q = p->next;
		free((void *)p);
		p = q;
	}
}

/*
 * print the entire list, 'Item' per line.
 * input: head pointer to the list.
 */
void
print_all(LINK h)
{
	LINK	p;
	int	cnt = 0;

	printf("\noutput:\n");
	p = h;
	while (p != NULL)
	{
		printf("%s", p->name);
		cnt++;
		if (cnt % Item == 0)
			printf("\n");
		else
			printf("\t");
		p = p->next;
	}
	printf("\n");
}
```