```
/*
 * 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 16, 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 a simple binary tree
 *              construction, traversal, and destruction by means of
 *              dynamically allocated tree nodes without any consideration of
 *              height balancing.  Thus, efficiency issue is of no concerned.
 */

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

#define        Normal         0
#define        Succeeded      1
#define        Failed         0
#define        Yes            1
#define        No             0

#define        Item           8
#define        Width          8

/*
 * global variables and definitions
 */
struct  rec
{
       int            val;
       struct  rec    *left;
       struct  rec    *right;
};

typedef struct  rec    NODE;
typedef NODE           *LINK;

int    counter = 0;

/*
 * function prototypes
 */
void   free_all(LINK);
void   print_tree(LINK);
int    proc_loop(LINK *);
void   insert_node(LINK, LINK);
LINK   get_node(void);

/*
 * Set up root pointer for the binary tree and invoke the main processing loop.
 * To read input values from a file instead of the keyboard, simply use command
 * line redirection, i.e.,
 *      a.out  <  infile_name
 *
 * A caveat was discovered regarding recursive call to print, whereby a newline
 * must be outputed at the end to flush the print buffer, a puzzling 'bug' indeed.
 */
int
main(void)
{
       LINK    rootn = NULL;
       int     rt;

       if ((rt = proc_loop(&rootn)) == Succeeded)
       {
              print_tree(rootn);
       }
       /*
        * caveat: this 'printf' line must be retained for the output
        * produced by 'print_tree' to be flushed (printed).
        */
       printf("\n\n");
```

```
	free_all(rootn);
	return Normal;
}

/*
 * description: the main processing module performs the following functions:
 *	1) read input number
 *	2) allocate a new node to store the value
 *	3) insert the new node into the proper place
 * input: address of root node.
 */
int
proc_loop(LINK *root)
{
	LINK	p, q;
	int	rc = Succeeded;
	int	tmp;

	printf("\nEnter input values (numbers) separated by blanks/newline, ");
	printf("press ctrl-d to quit\n");
	p = *root;
	while (scanf("%d", &tmp) != EOF)
	{
		if ((q = get_node()) == NULL)
		{
			rc = Failed;
			break;
		}
		q->val = tmp;
		/*
		 * insert new node into the right place
		 */
		if (p == NULL)
			p = q;
		else
			insert_node(p, q);
	}
	*root = p;
	return rc;
}

/*
 * description: this module recursively finds the right place to insert
 *		the new node in the tree, i.e., left < root < right.
 * input: addresses of root and new nodes.
 */
void
insert_node(LINK p, LINK q)
{
	if (p->val >= q->val)
	{
		if (p->left  == NULL)
		{
			p->left = q;
		}
		else
		{
			insert_node(p->left, q);
		}
	}
	else
	{
		if (p->right == NULL)
		{
			p->right = q;
		}
		else
		{
			insert_node(p->right, q);
		}
	}
}
```

```
/*
 * description: this function allocates memory space to hold a new node.
 * output: pointer to the new node or NULL if memory shortage is encountered.
 */
LINK
get_node(void)
{
	LINK	ptr;

	if ((ptr = (LINK)malloc(sizeof(NODE))) == (LINK)NULL)
		perror("Out of memory");
	else
	{
		ptr->left  = NULL;
		ptr->right = NULL;
	}
	return ptr;
}

/*
 * description: this function prints the tree 'in-order' of the values
 *              stored, i.e., left < root < right, with 'Item' per line.
 * input: address of the root node.
 */
void
print_tree(LINK root)
{
	LINK	p, l, r;

	p = root;
	if (p != NULL)
	{
		l = p->left;
		r = p->right;
		print_tree(l);
		printf("%-*d", Width, p->val);
		if (++counter % Item == 0)
		{
			printf("\n");
			counter = 0;
		}
		print_tree(r);
	}
}

/*
 * description: this function recursively frees the entire tree starting
 *              from the root node.
 * input: address of the root node.
 */
void
free_all(LINK root)
{
	LINK	p, l, r;

	p = root;
	if (p != NULL)
	{
		l = p->left;
		r = p->right;
		free((void *)p);
		free_all(l);
		free_all(r);
	}
}
```