Chapter 6

Data Structure

Introductory notes are included in PowerPoint file. The programs illustrated are as follows:

//elephnt1.c
// This example is from C for Scientists and Engineers

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

/* declare a self-referential structure */
typedef struct elephant
{
    char name[10];
    struct elephant *next;
} ELEPHANT;

void print_elephants( const ELEPHANT *ptr );

main()
{
    /* define three ELEPHANT variables and one pointer to ELEPHANT */
    ELEPHANT  elephant1, elephant2, elephant3, *start;

    /* store elephants' names */
    strcpy( elephant1.name, "Edna" );
    strcpy( elephant2.name, "Elmer" );
    strcpy( elephant3.name, "Eloise" );

    /* link elephants */
    elephant1.next = &elephant2; /* Edna points to Elmer */
    elephant2.next = &elephant3; /* Elmer points to Eloise */
    elephant3.next = NULL; /* Eloise is last */

    /* start contains the address of the first node */
    start = &elephant1;
    print_elephants( start );

    return EXIT_SUCCESS;
}

void print_elephants( const ELEPHANT *ptr )
{
    int count = 1;

    printf( "\n\n\n\n" );
    while ( ptr != NULL )
    {
        printf( "Elephant number %d is %s.\n", count++, ptr -> name );
        count++;
        ptr = ptr -> next;
    }
}
//elephnt2.c

// This example is from C for Scientists and Engineers

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

/* declare a self-referential structure */
typedef struct elephant
{
   char name[ 10 ];
   struct elephant *next;
} ELEPHANT;

void print_elephants( const ELEPHANT *ptr );

ELEPHANT *get_elephants( void ), *start;

main()
{
   clrscr();
   start = get_elephants();
   print_elephants( start );
   getch();
   return EXIT_SUCCESS;
}

ELEPHANT *get_elephants( void )
{
   ELEPHANT *current, *first;
   int response;

   /* allocate first node */
   current = first = malloc( sizeof ( ELEPHANT ) );

   /* store name of first elephant */
   printf( "NAME:	" );
   scanf( "%s", current -> name );

   /* prompt user about another elephant */
   printf( "Add another? (1 == yes, 0 == no)\t" );
   scanf( "%d", &response );

   /* try to allocate another elephant node */
   if ( ( current -> next = malloc( sizeof ( ELEPHANT ) ) ) == NULL )
   {
      printf( "Out of memory\nCan't add more elephants\n" );
      return first;
   }

   current = current -> next;

   /* store name of next elephant */
   printf( "NAME:\t" );
   scanf( "%s", current -> name );

   /* prompt user about another elephant */
   printf( "Add another? (1 == yes, 0 == no)\t" );
   scanf( "%d", &response );

   /* set link field in last node to NULL */
   current -> next = NULL;
   return first;
}

/* get_elephants allocates run-time storage for nodes. It builds
the linked list and stores user-supplied names in the name
fields of the nodes. It returns a pointer to the first such
node. */
/* print_elephants steps through the linked list pointed to by ptr and prints the name field in each node as well as the position of the node in the list */

void print_elephants( const ELEPHANT *ptr )
{
    int count = 1;
    printf( "\n\n\n" );
    while ( ptr != NULL )
    {
        printf( "Elephant number %d is %s.\n", count++, ptr -> name );
        ptr = ptr -> next;
    }
}

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

#define SIZE 100

typedef struct customer
{
    char lname[25]; /* last name */
    char fname[15]; /* first name */
    int account_no; /* account number */
    long int balance; /* balance */
} CUSTOMER;

CUSTOMER *customers[ SIZE ];

int front = 0, rear = 0; /* exit and entry positions in queue */
int count = 0; /* count of items in queue */

CUSTOMER *insert( CUSTOMER cust ), *remov( void );

void get_data( CUSTOMER *ptr ), put_data( const CUSTOMER *ptr );
main()
{
    int ans, flag;
    CUSTOMER cu, *ptr;
    /* do queue operations until user signals halt */
    clrscr();
    do
    {
        do
        {
            printf( "Enter 1 to insert, 2 to remove: " );
            scanf( "%d", &ans );
            printf( "\n" );
            switch ( ans ) {
            case 1: /* get a CUSTOMER and add to queue */
                get_data( &cu );
                if ( insert( cu ) == NULL )
                    printf( "\nQUEUE FULL\n\n" );
                break;
            case 2: /* delete a CUSTOMER from queue and print */
                if ( ( ptr = remov() ) != NULL )
                    put_data( ptr );
                else
                    printf( "\nQUEUE EMPTY\n\n" );
                break;
            default:
                printf( "\nIllegal response\n" );
                break;
            } //case
        } while ( ans != 1 && ans != 2 );
        printf( "1 to continue, 0 to quit: " );
        scanf( "%d", &flag );
    } while ( flag );
    return EXIT_SUCCESS;
}

/* get_data prompts the user for a CUSTOMER's last name, first name, account_no, and balance stores the data at the address passed. */
void get_data( CUSTOMER *ptr )
{
    printf( "Enter the customer's last name: ");
    scanf( "%s", ptr -> lname );
    printf( "Enter the customer's first name: ");
    scanf( "%s", ptr -> fname );
    printf( "Enter the customer's account number: ");
    scanf( "%d", ( ptr -> account_no ) );
    printf( "Enter the customer's balance: ");
    scanf( "%ld", ( ptr -> balance ) );
    printf( "\n" );
}

/* put_data writes the last name, first name, account_no, and balance of the CUSTOMER whose address is passed. */
void put_data( const CUSTOMER *ptr )
{
    printf( "Deleted Record: ");
    printf( "Customer's name: %s, %s\n", ptr -> lname,
            ptr -> fname );
    printf( "Customer's account number: %d\n",
            ptr -> account_no );
    printf( "Customer's balance: %ld\n", ptr -> balance );
    printf( "\n" );
}

/* If the queue is full, insert returns NULL. Otherwise, insert allocates storage for a CUSTOMER, copies the data passed into the allocated storage, adds a pointer to the CUSTOMER to the array, and returns the address of the CUSTOMER added. */
CUSTOMER *insert( CUSTOMER cust )
{
    CUSTOMER *ptr;
    
    if ( count >= SIZE ) /* queue full? */
        return NULL;
    
    ptr = malloc( sizeof( CUSTOMER ) ); /* new CUSTOMER */
    *ptr = cust; /* store data */
    customers[ rear ] = ptr; /* add CUSTOMER to queue */
    
    rear = ++rear % SIZE; /* update rear */
    ++count; /* update count */
    return ptr;
}

/* If the queue is empty, remov returns NULL. Otherwise, remov copies the CUSTOMER at the front to permanent storage, frees the queue storage, updates front, and returns the address of the CUSTOMER. */

CUSTOMER *remov( void )
{
    static CUSTOMER removed_cust;
    
    if ( count == 0 ) /* empty queue? */
        return NULL;
    
    removed_cust = *customers[ front ]; /* copy CUSTOMER at front */
    free( customers[ front ] ); /* collect garbage */
    front = ++front % SIZE; /* update front */
    --count;
    return &removed_cust;
}

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

#define SIZE 100
typedef struct customer
{
    char lname[25]; /* last name */
    char fname[15]; /* first name */
    int account_no; /* account number */
    long int balance; /* balance */
    struct customer *succ; /* successor on the queue */
} CUSTOMER;

CUSTOMER *front, *rear; /* exit entry positions in queue */
int count = 0; /* count of items in queue */

void get_data( CUSTOMER *ptr ), put_data( const CUSTOMER *ptr );
CUSTOMER *insert( CUSTOMER cust ), *remov( void );
main()
{
    int ans, flag;
    CUSTOMER cu, *ptr;

    /* do queue operations until user signals halt */
    do {
        do {
            printf( "Enter 1 to insert, 2 to remove: " );
            scanf( "%d", &ans );
            printf( "\n" );
            switch ( ans )
            {
                case 1: /* get a CUSTOMER and add to queue */
                    get_data( &cu );
                    if ( insert( cu ) == NULL )
                        printf( "\n\nQUEUE FULL\n\n" );
                    break;
                case 2: /* delete a CUSTOMER from queue and print */
                    if ( ( ptr = remov() ) != NULL )
                        put_data( ptr );
                    else
                        printf( "\n\nQUEUE EMPTY\n\n" );
                    break;
                default:
                    printf( "\nIllegal response\n" );
                    break;
            }
        } while ( ans != 1 && ans != 2 );
        printf( "\n1 to continue, 0 to quit: " );
        scanf( "%d", &flag );
        printf( "\n" );
    } while ( flag );

    return EXIT_SUCCESS;
}

/* get_data prompts the user for a CUSTOMER's last name, first name, account_no, and balance stores the data at the address passed. */
void get_data( CUSTOMER *ptr )
{
    printf( "Enter the customer's last name: " );
    scanf( "%s", ptr -> lname );
    printf( "Enter the customer's first name: " );
    scanf( "%s", ptr -> fname );
    printf( "Enter the customer's account number: " );
    scanf( "%d", &( ptr -> account_no ) );
    printf( "Enter the customer's balance: " );
    scanf( "%ld", &( ptr -> balance ) );
    printf( "\n" );
}

/* put_data writes the last name, first name, account_no, and balance of the CUSTOMER whose address is passed. */
void put_data( const CUSTOMER *ptr )
{
    printf( "Customer's name: %s, %s\n", ptr -> lname, ptr -> fname );
    printf( "Customer's account number: %d\n", ptr -> account_no );
    printf( "Customer's balance: %ld\n", ptr -> balance );
}

/* If the queue is full, insert returns NULL. Otherwise, insert allocates storage for a CUSTOMER, copies the data passed into the allocated storage, adds the node to the rear (last node in the linked list), updates rear, NULLs the link field of the new node, updates count, and returns the address of the CUSTOMER added. */
CUSTOMER *insert( CUSTOMER cust )
{
    CUSTOMER *ptr;
    if ( count >= SIZE ) /* queue full? */
        return NULL;
    ptr = malloc( sizeof ( CUSTOMER ) ); /* new CUSTOMER */
    *ptr = cust; /* store data */
    if ( count == 0 ) /* empty queue? */
        front = ptr; /* front points to first node in list */
    else
        rear -> succ = ptr; /* if queue not empty, add at end */
    rear = ptr; /* update rear */
    ptr -> succ = NULL; /* NULL last succ field */
    ++count; /* update count */
    return ptr;
}

/* If the queue is empty, remov returns NULL. Otherwise, remov
 copies the CUSTOMER at the front (first node in the linked list)
 to permanent storage, updates front, frees the node, updates
 count, and returns the address of the CUSTOMER. */

CUSTOMER *remov( void )
{
    static CUSTOMER removed_cust;
    CUSTOMER *next;
    if ( count == 0 ) /* empty queue? */
        return NULL;
    removed_cust = *front; /* copy CUSTOMER at front */
    next = front; /* save front node's address for freeing */
    front = front -> succ; /* remove front node */
    free( next ); /* deallocate storage */
    --count; /* update count */
    return &removed_cust;
}
main()
{
    int ans, flag;
    TRAY t, *ptr;

    /* do stack operations until user signals halt */
    do {
        do {
            printf( "Enter 1 ... &ans );
            scanf( "%d", &ans );

            switch ( ans ) {
                case 1: /* get a TRAY and add it to stack */
                    get_data( &t );
                    if ( push( t ) == NULL )
                        printf( "STACK FULL

" );
                    break;
                case 2: /* delete a TRAY from stack and print it */
                    if ( ( ptr = pop() ) != NULL )
                        put_data( ptr );
                    else
                        printf( "STACK EMPTY

" );
                    break;
                default:
                    printf( "nIllegal response
" );
                    break;
            }
        } while ( ans != 1 && ans != 2 );
        printf( "n1 to continue, 0 to quit: ");
        scanf( "%d", &flag );
    } while ( flag );

    return EXIT_SUCCESS;
}

/* get_data prompts the user for a TRAY's color and id and stores it at the address passed. */
void get_data( TRAY *ptr )
{
    printf( "Enter the tray's color: ");
    scanf( "%s", ptr -> color );
    printf( "Enter the tray's id: ");
    scanf( "%d", &ptr -> id );
}

/* put_data writes the color and id of the TRAY whose address is passed. */
void put_data( const TRAY *ptr )
{
    printf( "tray's color: %s\n", ptr -> color );
    printf( "tray's id: %d\n", ptr -> id );
}

/* If the stack is full, push returns NULL. Otherwise, push allocates storage for a TRAY, copies the data passed into the allocated storage, pushes a pointer to the TRAY onto the stack, and returns the address of the TRAY added. */
TRAY *push( TRAY tr )
{
    TRAY *ptr;

    if ( top >= SIZE - 1 ) /* stack full */ return NULL;

    ptr = malloc( sizeof( TRAY ) ); /* new TRAY */
    *ptr = tr; /* store data */
    trays[ ++top ] = ptr; /* push it and update top */
    return ptr;
}
If the stack is empty, pop returns NULL. Otherwise, pop copies the top TRAY to permanent storage, frees the stack storage, updates top, and returns the address of the TRAY.

```c
TRAY *pop( void )
{
    static TRAY popped_tray;

    if ( top < 0 ) /* empty stack? */
        return NULL;

    popped_tray = *trays[ top ]; /* copy top TRAY */
    free( trays[ top-- ] ); /* collect garbage */
    return &popped_tray;
}
```

// This program is from C for Scientists and Engineers

// Implementation of Stack by linked list

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

#define SIZE 100

typedef struct tray {
    char color[10]; /* its color */
    int id;          /* its unique id number */
    struct tray *below; /* pointer to successor on stack */
} TRAY;

TRAY *top = NULL; /* pointer to top TRAY on stack */
int currsize = 0; /* number of items on stack */

void get_data( TRAY *ptr ), put_data( const TRAY *ptr );
TRAY *pop( void ), *push( TRAY tr );
```
main()
{  
    int ans, flag;
    TRAY t, *ptr;

    /* do stack operations until user signals halt */  
    do {
        do {
            printf( "Enter 1 to push, 2 to pop: ");
            scanf( "%d", &ans );

            switch ( ans ) {
                case 1: /* get a TRAY and add it to stack */
                    get_data( &t );
                    if ( push( t ) == NULL )
                        printf( "STACK FULL\n\n" );
                    break;
                case 2: /* delete a TRAY from stack and print it */
                    if ( ( ptr = pop() ) != NULL )
                        put_data( ptr );
                    else printf( "STACK EMPTY\n\n" );
                    break;
                default:
                    printf( "Illegal response\n" );
                    break;
            }
        } while ( ans != 1 && ans != 2 );
        printf( "1 to continue, 0 to quit: ");
        scanf( "%d", &flag );
    } while ( flag );

    return EXIT_SUCCESS;
}  

/* get_data prompts the user for a TRAY's color and id and stores it at the address passed. */

void get_data( TRAY *ptr ){
    printf( "Enter the tray's color: ");
    scanf( "%s", ptr -> color );
    printf( "Enter the tray's id: ");
    scanf( "%d", & ( ptr -> id ) );
}

/* put_data writes the color and id of the TRAY whose address is passed. */

void put_data( const TRAY *ptr ){
    printf( "Tray's color: %s\n", ptr -> color );
    printf( "Tray's id: %d\n", ptr -> id );
}
/* If the stack is full, push returns NULL. Otherwise, push allocates storage for a TRAY, copies the data passed into the allocated storage, adds the node to the linked list, updates top and the current size of the stack, and returns the address of the TRAY added. */

TRAY *push( TRAY tr )
{
    TRAY *ptr;
    if ( currsize >= SIZE ) /* stack full? */
        return NULL;
    ptr = malloc( sizeof ( TRAY ) ); /* new TRAY */
    *ptr = tr; /* store data */
    ptr -> below = top; /* push it on stack */
    top = ptr; /* update top */
    ++currsize; /* update current stack size */
    return ptr;
}

/* If the stack is empty, pop returns NULL. Otherwise, pop copies the top TRAY to permanent storage, updates top, frees the stack storage, updates the current size of the stack, and returns the address of the TRAY. */

TRAY *pop( void )
{
    static TRAY popped_tray;
    TRAY *ptr;
    if ( currsize < 1 ) /* empty stack? */
        return NULL;
    popped_tray = *top; /* copy data to return */
    ptr = top; /* save address of 1st node for garbage collection */
    top = top -> below; /* update top */
    free( ptr ); /* collect garbage */
    --currsize; /* update current size */
    return &popped_tray;
}