Linear lists

The logical level

- What is a list?
  - Homogeneous collection of elements with a linear relationship between elements
  - Each element in the list, except the first one, has a unique predecessor
  - Each element in the list, except the last one, has a unique successor
  - The elements in the list need not be in any particular order, but they can also be ordered in different ways
    - Stacks and queues are special types of lists that are ordered by the time when the elements were added
    - Lists can also be ordered by value
      - List of grades could be ordered numerically
      - Such lists are called *value-ordered lists*, or *sorted lists*
    - If the elements of the list are records, their order is determined by one (or more) field, known as the *key field*, leading to *key-ordered lists*
  - If a list cannot contain records with duplicate keys, it is considered to have *unique keys*
  - **Key** – Field in a record whose value is used to determine the logical (and/or physical) order of the records in a list
  - We will consider lists with unique keys, ordered from smallest to largest key values
- List ADT specification

  Structure of the list

  - All the elements are of type `list_element_type`
  - Each element contains a key field, called `key`
  - List is logically ordered from smallest unique element key value to the largest

  Operations on the list ADT

  Create a list

  Function. Initializes list to empty state
  Input. None
  Preconditions. None
  Output. A list of `list_type`
  Postconditions. Lists exists and is empty

  Destroy list

  Function. Removes all elements from list, leaving the list empty
  Input. A list
  Preconditions. The list exists
  Output. List
  Postconditions. The list is empty

  Empty list? Checks to see if the list is empty

  Function. Tests whether the list is empty
  Input. A list
  Preconditions. List has been created
  Output. True or False (1 or 0)
  Postconditions. True if list is empty, false otherwise

  Full list? Checks to see if the list is full

  Function. Tests whether the list is full
  Input. A list
  Preconditions. List has been created
  Output. True or False (1 or 0)
  Postconditions. True if list is full, false otherwise

Retrieve an element

- **Function.** If found, a copy of the element that contains the key `key_value` is returned
- **Input.** List as well as the `key_value`
- **Preconditions.** List has been created
- **Output.** Element, and a logical flag `found`
- **Postconditions.** `found` indicates if an element with key `key_value` exists in the list; if it does, `element` contains a copy of the element; list itself is unchanged

Insert an element into the list

- **Function.** Adds new element to the list
- **Input.** List and new element
- **Preconditions.** List is not full and new element is not in the list
- **Output.** List
- **Postconditions.** List contains the new element

Modify an element in the list

- **Function.** Replace existing element with same key as modified element
- **Input.** List and the modified element
- **Preconditions.** Element with the same key as the modified element exists in the list
- **Output.** List
- **Postconditions.** List, with the value of the modified element replaced

Delete an element from the list

- **Function.** Deletes the element containing the key `delete_val` from the list
- **Input.** List and the key `delete_val`
- **Preconditions.** Element with key `delete_val` is in the list and appears only once
- **Output.** List
- **Postconditions.** List with the element specified by the key `delete_val` removed

Print the list

- **Function.** Prints all the elements in the list in order from smallest to the largest value
- **Input.** List
- **Preconditions.** List has been created
- **Output.** List elements (to standard output)
- **Postconditions.** List elements have been printed in order from smallest to largest key value; list itself is unchanged

The user level

- Enhance the basic ADT as specified with additional operations
- Add the operation `change_key`
  - Retrieve the record with first key
  - Delete the record with first key
  - Change the value of the key in the retrieved record
  - Insert the modified record in the list
- Add the operation to create the list by reading records from a file
  - Create an empty list
  - Using a while loop, read a record and insert it into the list till all the records have been read and inserted

The implementation level

- Logical order of the elements in a list may or may not be mirrored in the way they are physically stored in a data structure

- The physical arrangement of the list elements affects the way they will be accessed
- The specification of the list ADT does not require the elements to be stored in order
- We could implement the list operations by storing the elements as completely unordered in an array
  - The algorithm to insert elements will be $O(1)$ but the algorithm to print them in sorted order will be $O(N^2)$, or $O(N \lg N)$
  - If the elements of a list are physically ordered, the two operations will be $O(N)$
- Sequential list representation in an array
  - Stores the elements in the list sequentially
  - Order of elements is implicit in their placement in the array
- Linked list representation
  - Data elements are not constrained to be stored in physically contiguous, sequential locations
  - Individual elements are stored "somewhere" in the memory and order is maintained by explicit links between them
- List design jargon (more abstractions)

  node(location) refers to all the data at the location
  info(location) refers to the user's data at the location
  key(location) refers to the key of the user's data at location
  next(location) refers to the node following node(location)
  location itself depends on the implementation type
  - In an array-based implementation, location is an index
  - In a linked implementation where each element is dynamically allocated, location is an address (pointer)

  Each of these abstractions can be implemented as a separate function in addition to the regular way of coding

Sequential list implementations

- The elements in the sequential list implementation will be stored as an array of structures

```
#define MAX_ELEMENTS 100

...

    typedef char list_element_type;
    list_element_type info[MAX_ELEMENTS];
```

- The first element (smallest key value) will be stored in the first slot `info[0]`, the next element in second slot, and so on
- The array ends with the slot at `MAX_ELEMENTS - 1` but the list may not fill the entire array
  - We'll keep track of the number of elements in the list by a separate variable `length`
  - This will also tell us that the largest value in the list is in the slot `length - 1`
- The entire structure is put together as

```
#define MAX_ELEMENTS 100     /* Maximum number of elements in the list */

typedef char key_type;       /* Type of the key field                  */
typedef struct
{
    key_type   key;          /* Key field                              */
    ...                      /* Other fields in the structure as needed */
} list_element_type;         /* Type of the structure of each element  */

typedef struct
{
    int        length;       /* Number of elements in the list         */
    list_element_type info[MAX_ELEMENTS]; /* The data in the list      */
} list_type;                 /* The list itself                        */

list_type * list;
```

- Operations on the lists
  - Creating the list

    ```
    list_type * create_list ( list_type * list )
    {
        /* Allocate memory for list                                */

        list = ( list_type * ) malloc ( sizeof ( list_type ) );

        /* Initialize the number of items in the list to zero     */

        list -> length = 0;

        return ( list );
    }
    ```
  - Destroying the list

    ```
    list_type * destroy_list ( list_type * list )
    {
        /* Change the length of the list to zero                   */

        list -> length = 0;

        return ( list );
    }
    ```
  - Checking for the list being empty

    ```
    int empty_list ( list_type * list )
    {
        /* Return true if list is empty, false otherwise          */

        return ( list -> length == 0 );
    }
    ```
  - Checking for the list being full

    ```
    int full_list ( list_type * list )
    {
        /* Return true if list is full, false otherwise           */

        return ( list -> length == MAX_ELEMENTS );
    }
    ```
  - Printing the contents of the list

- Need to process all the elements in the list
- Pseudocode

```
location ← start of list
while more elements in the list
    print ( info(location) )
    location ← next(location)
```

- Since this is an array-based implementation, the first element is in location zero
- On entrance to each iteration of the loop, all the elements up to, but not including, `list->info[location-1]` have been printed
- First loop invariant is:

  `list->info[0]` ... `list->info[location-1]` *have been printed*

- We want to stop looping when the whole list has been printed which occurs when `location` equals `list -> length`
- Second loop invariant is:

  $0 \leq$ `location` $\leq$ `list -> length`

- We also assume the existence of a local function `print_info` that will print the information field of one element

```
void print_list ( list_type * list )
{
    /* Prints the contents of an entire list                */

    for ( location = 0; location < list -> length; location++ )
        print_info ( list -> info[location] );
}
```

– Finding a list element

- All the remaining operations on list require the ability to search the list
- We will develop a function `find_element` that will return the array index of the element, if it exists, and -1 if it does not
- If the element is found, and we need to make any modifications to it (such as deleting it), we may also want to get the address of its *logical predecessor*
- The complete ADT specification of the operation is

  Function. Search the list for the element whose key is `key_val`
  Input. List and a key value
  Preconditions. List has been created
  Output. Location and predecessor location
  Postconditions. If key value is found in the list, location contains the index number of the found element and predecessor location contains the index number of the previous location; if the key value is not found, location contains -1 and the predecessor location contains the index number of the logical predecessor of the element
- For simplicity, we'll use a linear search

```
#define TRUE  1
#define FALSE 0

int find_element ( list_type * list, key_type key_val, int * pred_loc )
{
    /* Find the element with key key_val                          */
    /* If the element exists, return the location and set pred_loc */
    /*    to logical predecessor location                          */
    /* If the element does not exist, return -1 for location and   */
    /*    set pred_loc to logical predecessor location             */

    int index;               /* Keep track of the element being looked at  */
    int more_to_search;      /* Make sure that we do not go too far        */

    /* Set up the search                                                   */

    index = 0;               /* First element in the array                 */
    more_to_search = TRUE;    /* There are more elements to be searched     */

    while ( more_to_search && ( index < list -> length ) )
        if ( (list -> info[index]).key < key_val )
            index++;
        else
            more_to_search = FALSE;

    /* Set output parameters                                                */

    *pred_loc = index - 1;

    if ( index > list -> length )   /* Key value not found                  */
        return ( -1 );
    else
        if ( (list -> info[index]).key == key_val )   /* Key found          */
            return ( index );
        else
            return ( -1 );
}
```

  * This function is not found in the list ADT specifications as it is *private* or *logically internal* for the use of list operations, and not intended for the list user
  * The reason that it is private is because ewe do not want the list user to know the way we store the data in the list (the function will be created differently for the dynamic allocation strategy)
  * This fits in well with the goal of information hiding

- Retrieving an element
  * Allows the user to access the list element with a specified key, if the element exists in the list
  * The basic algorithm is

```
find location of key value in the list
found ← key value was found in the list
if found
   element ← info(location)
```

  * The code can be given as

```
list_element_type * retrieve_element ( list_type * list, key_type key_val, \
                                       int * found )
{
    int location;     /* Location of element in the array                   */
    int pred_loc;     /* Previous location; not applicable to this operation */
    list_element_type * tmp; /* Location to hold the list element to retrieve*/

    location = find_element ( list, key_val, &pred_loc );

    *found = ( location >= 0 );

    if ( *found )
    {
        tmp = ( list_element_type * ) malloc ( sizeof ( list_element_type ) );
        *tmp = list -> info[location];
        return ( tmp );
    }
    else
        return ( NULL );
}
```

- Modify an element
  * To modify, we need to find the location of the element to be modified, and then, replace it with the modified copy of the element
  * The key of the modified element stays the same as the original element
  * The code can be given as

```
list_type * modify_element ( list_type * list, list_element_type * mod_element )
{
    int location;     /* Location of element in the array                   */
    int pred_loc;     /* Previous location; not applicable to this operation */

    location = find_element ( list, key_val, &pred_loc );

    if ( location >= 0 )
        return ( list -> info[location] = *mod_element );
    else
        return ( NULL );
}
```

- Inserting an element
  - We need to find the place where the element belongs
  - If the element exists in the list, we do not add the new element
  - Otherwise, we create space for the new element and put it in there
  - The code can be given as

```
list_type * insert_element ( list_type * list, list_element_type * new_element,\
                             int * error )
{
    int location;     /* Location of element in the array                  */
    int pred_loc;     /* Previous location                                 */
    int i;            /* Just a counter (index counter)                    */

    *error = 0;       /* Error is initially false                         */

    if ( location = find_element ( list, new_element -> key, &pred_loc ) >= 0 )
    {
        *error = TRUE;
        return ( list );    /* Error, list is returned unchanged           */
    }

    /* Make room for the new element by moving all the other elements      */
    /* forward by one                                                      */

    for ( i = list -> length; i > pred_loc; i-- )
        list -> info[i+1] = list -> info[i];

    /* Put new element in the proper place                                 */

    list -> info[pred_loc+1] = *new_element;
    list -> length++;

    return ( list );
}
```

  - Check to make sure that the function handles the special cases when the element needs to be inserted at the beginning or end of the list
- Deleting an element
  - Again, we need to find out the location of the element
  - If the element is found, delete it and move the rest of the list to reflect the new structure
  - The code can be given as

```
list_type * delete_element ( list_type * list, key_type delete_val, int * error)
{
    int location;     /* Location of element in the array                  */
    int pred_loc;     /* Previous location; not applicable to this operation */
    int i;            /* Just a counter (index counter)                    */

    *error = 0;       /* Error is initially false                         */

    /* Error if the element does not exist in the list                    */

    if ( location = find_element ( list, delete_val, &pred_loc ) == -1 )
    {
        *error = TRUE;
        return ( list );    /* Error, list is returned unchanged           */
    }

    /* Remove the element by moving the rest of the elements up the list    */

    for ( i = location; i < list -> length; i++ )
        list -> info[i] = list -> info[i+1];

    list -> length--;

    return ( list );
}
```

- Notes on the sequential list implementation
  - In many operations, a local variable `location` is declared which contains the array index of the list element being processed
  - This information is not visible outside the function and is *internal* to the list ADT
  - Even when a user wants a record in the list, he gets the complete record instead of getting a pointer (index) to it

  - The list user never gets to see the internal structure of the list
  - These features make the implementation encapsulated by the ADT

Linked list implementations

- In linked list, elements are not constrained to be stored in sequential order
- The elements may be scattered anywhere in the memory with explicit links to connect them
- In the linked list implementation, the last node of the list does not point anywhere and therefore, its link field contains `NULL`
- An external pointer tells us the location where the list starts
- The nodes in the linked implementation cannot be accessed directly but only through other nodes (or links)
- One implementation of this technique will store the list in an array and connect the elements through explicit indices
- The main advantage with linked list is that the existing elements stay in place when new elements are added or deleted
- Implementing a linked list
  - Structure of each node
    * Each element contains at least two fields

      info to keep the data at the node, including the key
      next to keep the link to the next node
    * The structure can be coded as

```
typedef char   key_type;    /* Type of the key field                   */
typedef struct node_type
{
    key_type   key;         /* Key field                               */
    ...                     /* Other fields in the structure as needed */
    struct node_type next;  /* Pointer to the next element in the list */
} list_element_type;        /* Type of the structure of each element   */

typedef list_element_type list_type;

list_type * list;
```

  - Creating the list

```
list_type * create_list ( list_type * list )
{
    /* Initialize the list header to point to nothing         */

    list = NULL;

    return ( list );
}
```

  - Checking for the list being empty

```
int empty_list ( list_type * list )
{
    /* Return true if list is empty, false otherwise         */

    return ( list == NULL );
}
```

  - Checking for the list being full

```
int full_list ( list_type * list )
{
    /* Return true if list is full, false otherwise          */
    /* In linked implementation, list is never full         */

    return ( 0 );
}
```

- You can also check for the amount of free memory available and make the function more complicated
- Destroying the list

```
list_type * destroy_list ( list_type * list )
{
    list_element_type * tmp;

    while ( list )             /* is not empty                                  */
    {
        /* Copy the address of the first node in the list                       */

        tmp = list;

        /* Make the list point to the next node                                 */

        list = list -> next;

        /* Free the space allocated to the node                                 */

        free ( tmp );
    }

    return ( list );
}
```

- Printing the contents of the list

```
void print_list ( list_type * list )
{
    list_element_type * location;     /* Temporary pointer within the list  */

    /* Prints the contents of the entire list                                */

    for ( location = list; location /* is not empty */; location = location->next)
        print_info ( location -> info );
}
```

- Finding a list element
  - We will develop a function `find_element` that will return the pointer to the element, if it exists, and `NULL` if it does not
  - If the element is found, and we need to make any modifications to it (such as deleting it), we may also want to get the address of its *logical predecessor*
  - We'll use the same ADT specifications that were developed earlier
  - For simplicity, we'll use a linear search

```
#define TRUE  1
#define FALSE 0

list_element_type * find_element ( list_type * list, key_type key_val, \
        list_element_type * pred_loc )
{
    /* Find the element with key key_val                               */
```

```
        /* If the element exists, return the location and set pred_loc  */
        /*    to logical predecessor location                            */
        /* If the element does not exist, return NULL for location and  */
        /*    set pred_loc to logical predecessor location               */

        list_element_type *location; /* Keep track of the element being looked at */
        int more_to_search;       /* Make sure that we do not go too far        */

        /* Set up the search                                                    */

        location = list;          /* First element in the list                  */
        pred_loc = NULL;          /* No previous location yet                   */
        more_to_search = TRUE;    /* There are more elements to be searched     */

        while ( more_to_search && location /* is not NULL */ ) )
            if ( location -> key < key_val )
            {
                pred_loc = location;
                location = location -> next;
            }
            else
                more_to_search = FALSE;

        /* Set output parameters                                                */

        if ( location && ( location -> key != key_val ) ) /* Key value not found */
            location = NULL;

        return ( location );
    }
```

– Retrieving an element

```
list_element_type * retrieve_element ( list_type * list, key_type key_val, \
                                        int * found )
{
    int location;     /* Location of element in the array                   */
    int pred_loc;     /* Previous location; not applicable to this operation */

    location = find_element ( list, key_val, pred_loc );

    *found = ( location != NULL );

    if ( *found )
        return ( location -> info );
}
```

– Modify an element
  * To modify, we need to find the location of the element to be modified, and then, replace it with the modified copy of the element
  * The key of the modified element stays the same as the original element
  * The code can be given as

```
list_type * modify_element ( list_type * list, list_element_type * mod_element )
{
    int location;     /* Location of element in the array                   */
```

```
        int pred_loc;     /* Previous location; not applicable to this operation */

        location = find_element ( list, mod_element -> key, pred_loc );

        location -> info = mod_element -> info;
    }
```

- Inserting an element

```
list_type * insert_element ( list_type * list, list_element_type * new_element,\
                             int * error )
{
    list_element_type * location;     /* Location of element in the array     */
    list_element_type * pred_loc;     /* Previous location                     */
    list_element_type * tmp;

    *error = 0;       /* Error is initially false                              */

    if ( location = find_element ( list, new_element -> key, pred_loc ) )
    {
        *error = TRUE;
        return ( list );    /* Error, list is returned unchanged             */
    }

    /* Make room for the new element by allocating memory                     */

    tmp = ( list_element_type * ) malloc ( sizeof ( list_element_type ) );
    tmp->info = *new_element;

    /* Put new element in the proper place                                   */

    if ( pred_loc )
    {
        tmp -> next = pred_loc->next;
        pred_loc -> next = tmp;
    }
    else
    {
        // No predecessor; this will be first element

        tmp -> next = list;
        list = tmp;
    }

    return ( list );
}
```

- Deleting an element
  * Again, we need to find out the location of the element
  * If the element is found, delete it and modify the links to reflect the new structure
  * The code can be given as

```
list_type * delete_element ( list_type * list, key_type delete_val, int * error)
{
    list_element_type * location; /* Location of element in the array */
    list_element_type * pred_loc; /* Previous location                 */
```

```
        *error = 0;        /* Error is initially false                      */

        /* Error if the element does not exist in the list                  */

        if ( ( location = find_element ( list, delete_val, pred_loc ) ) == NULL )
        {
            *error = TRUE;
            return ( list );    /* Error, list is returned unchanged        */
        }

        /* Remove the element by modifying the links                        */

        if ( pred_loc == NULL )   /* Deleting first element in the list     */
            list = location -> next;
        else
            pred_loc -> next = location -> next;

        free ( location );

        return ( list );
    }
```

- Analysis of list operations (left as homework exercise)

Application level (you will implement this project as your last assignment)

- Your brother-in-law Marc started a hobby magazine, a small business that is finally becoming successful. He uses his personal computer for desktop publishing, but does all his bookkeeping by hand. Now that he has quite a few advertisers, writing up the bills has become a real chore for him. So he has called you with a proposition: Could you write a program to automate this part of his business?

  The Specification. Obviously, Marc is not able to give you a formal specification. He has a pretty good idea of what he wants, however, and you know what kind of information goes into a specification. So the two of you go out for a pizza, and together you come up with a software specification for program `admgr`.

  Marc tells you that he currently does his billing as follows: Every time an advertiser places an ad for the current issue of the magazine, Marc writes down a description of the ad on a piece of paper. The description includes the name of the advertiser, and the billing address (if it is a new advertiser), as well as the size and color of the ad. His magazine has four ad sizes: full-, half-, quarter-, and eighth-page, each with a set price. If an advertiser wants a color – red, blue, green, or yellow – in addition to black, there is a fee. The fee is higher, of course, for ads with full-color photographs.

  All of these slips of paper are kept in a folder. When he is ready to "dummy" the magazine (position all the ads on magazine pages), he goes through the file and writes down a list of the ads by size – all the full-page ads, then all the half-page ads, and so on. After the ads are in place, he positions his magazine articles, photos, and stories on the pages. When they are complete, he sends the pages to a commercial printer.

  Finally, while the magazine is at the printer, Marc gets around to billing. He collects all the slips of paper, and types up the bills, using a calculator to add the charges for the ads in the current issue to the advertiser's previous balance. (He requires first-time advertisers to pay at the time of their order, but good customers get credit.) This is the part of business that Marc hates. It's so *uncreative*. Since he uses a computer in his business, he'd like to automate this billing.

  Now that you have an idea of the general problem, you start a draft of the specifications. You know some of the inputs: all of the information that Marc writes down on slips of paper – advertiser names, addresses, ad sizes, and colors. You know the main output: the bills. The two of you decide that the bills should be written to a file called `bills` and not to the printer; Marc will later print them out by using the operating system command `lpr`.

Since ads for each issue are collected over a period of several weeks, some data must be retained from one execution of the program to the next. This retained data will also be saved in a file. You note the existence of this file under I/O in the specifications. Marc does not plan to use or modify this information directly, so the file format is not important to him. Therefore, you do not need to mention the *format* of this file in the program specifications – its implementation is part of the program's data design.

As Marc is the program's *user*, he is interested in the *user interface*. He wants to know how he, the human being, will interact with your program. Since you do not know how to program with bars and pull-down menus, you suggest writing the initial program with a simple textual command interface, with prompts to tell Marc what to enter. Later, you will try to modify it to have a choice bar and pulldown windows and whatever "bells and whistles" he wants. You note in the **Programming Requirements** part of your specification that the part of the program that handles the user interface should be very modular, so that it will be easy to replace later.

You finish up with a discussion of exactly what commands the program will process: commands to add a new advertiser, to place an ad in the current issue, to generate all the bills, to record an advertiser's payment when the check arrives, and to quit. You go home to draft the program specifications and by midnight, come up with the following.

Function. The program will support the organization and billing of the ads in a small magazine.

Input.

1. File `ads` contains the data created in the previous executions of the program, if it has been run before.
2. The user enters data interactively from the keyboard, as described under Processing Requirements below.

Output.

1. File `ads` is used to save the state of the advertising data between executions of the program. This data includes the descriptions of the current issue's ads (size and color) and each advertiser's address and previous balance.
2. File `bills` is a text file that contains the bills for the current issue. Each bill must contain the advertiser's name and address, and itemized list of the current issue's advertisements, the previous balance, and the total amount due.
3. Input prompts and messages are written to the screen, as described under Processing Requirement below.

Processing Requirements.

1. The program must process the following commands:

   Bill advertisers. Create text file `bills` to contain the advertisers' statements. The statements will be ordered alphabetically by advertiser name.

   New advertiser. Add a new advertiser to the ad data; prompt user for advertiser name and address.

   Accept ad. Add an advertisement to the ad data; prompt user for advertiser name, size, and color.

   Receive payment. Update ad data to reflect a payment from an advertiser. Prompt user for advertiser name and amount of new payment. Advise user of current balance before and after payment.

   Quit. Save state of ad data and terminate program.
2. The program should keep track of the "current advertiser" being processed, so that this name can be used as a default in commands to accept ads or receive payments. The user can press Enter to accept the current advertiser, or can type a new name.
3. The program is to be used interactively; it must supply helpful prompts and messages to the user. The user-input part of the program should be implemented modularly to allow it to be replaced at a later date with a different user interface.

Assumptions.

1. The program will be used on the magazine's Unix machine.
2. The ad data will fit in memory.

The Design. As always, you start with the design phase. You can see from the inputs and outputs that data design is central to this program. The main data object is some kind of *list of advertisers*. This object is called `ad_list`. Each advertiser has associated with it several pieces of information:

1. A name

2. A billing address
3. A list of advertisements for the current issue
4. The previous balance (money owed from ads in previous issues)

The description of an advertiser suggests a structure, with a field for each piece of information. Since the billing statements in the `bills` file will be ordered alphabetically by advertiser name, which seems to be the key field.

You don't really need to decide at this point how the list of advertiser records will be implemented. Assuming that it is a list, however, you know of a set of basic operations that are available to manipulate `ad_list` – specified in the list ADT.

How does `ad_list` related to the program's *retained data*, which is saved in a file between executions of the program? The retained data is the same information in some kind of file format. You can put off deciding how both the file and `ad_list` are implemented until later. For now you can start the top-down design.

```
ad_mgr  -- Level 0

initialize
repeat
    get_command
    command is:
      bill_advertisers      : gen_bills
      new_advertiser        : add_advertiser
      accept_ad             : process_ad
      receive_pmt           : process_pmt
      quit                  : terminate_program
forever
```

Initialization processing. The main job of the `initialize` module is to get the information from the previous execution of the program – the program's retained data – from the file, and to use this data to rebuild the `ad_list`. Creating a list from data in a file is not one of the basic operations specified in the list ADT, so you specify a level 2 operation `read_list_from_file`:

`read_list_from_file ( ad_file, ad_list )` – Level 2

Function. Read advertiser data from the `ad_file`, and store in `ad_list`

Input. `ad_file`, the file of ads

Preconditions. `ad_file` is not yet open; `ad_list` has not yet been created

Output. `ad_list` of `list_type`

Postconditions. `ad_list` contains all of the information stored in `ad_file`; `ad_file` is unchanged

The algorithm for this operation is:

```
create_list ( ad_list )
open ad_file for reading

while ad_file contains more data
    read all the data for one advertiser into ad_rec
    insert_element ( ad_list, ad_rec )
```

How do you "read all the data for one advertiser into `ad_rec`"? The answer depends on how the data is stored in the file. Marc doesn't care how the data is stored between executions of the program, so it is up to you to determine the format of `ad_file`. One approach is to keep the data in a text file, reading and writing all the record fields one at a time. Using a text file requires you to convert back and forth between the way the data is formatted in the text file and the way the data is stored in a record. A simpler alternative is to use a *binary* file.

Following initialization, a big loop is entered: `repeat ... forever`. In each iteration of the loop body, a command is input from the user and processed.

Getting a command. You told Marc that you'd work on a fancy user interface later. Meanwhile, you decide to create a set of *operations* for getting user inputs, rather than putting scattered calls to the same throughout the program. The function to input a command is called `get_command`

```
get_command ()
```

Function. Read a command from user.
Input. From keyboard: command input.
Preconditions. User is awake.
Output.
To screen: prompt to the user
To calling program: `command` of `command_type`
Postconditions. `command` contains valid command value.

Before we can design this function, we need to know more about the output value. A value of type `command_type` indicates the command specified by the user, listed earlier, and is defined using the `enum` types in C.
Of course, the user cannot type such a value directly into the computer. Instead, you must prompt the user to type in a value that your program will convert to `command_type`. To make it easy for the user, you decide to prompt the user to enter a single letter command – "B" for "Bill Advertisers", etc.
To input the command, you read the character and then, convert it to the appropriate `command_type` value (may be by using a `switch` statement). Another way to do the same is by using a *command table* but it may not be efficient. In any case, prompt the user to input the command again if the first input is not valid.
Before you record the `get_command` design, you give some thought to possible errors. What if the user types a lowercase letter?

```
command_type get_command ()

command = unknown
print instruction prompt
repeat
    read character from keyboard
    case character is
    'A' : command = accept_ad; return
    'B' : command = bill_advertisers; return
    'N' : command = new_advertiser; return
    'R' : command = receive_pmt; return
    'Q' : command = quit; return
until command != unknown
```

Processing the command. Now that the command is known, the program must process it. The Level 0 design contains a big selection statement (switch?) to select the appropriate operation for each command. We will go through the design for the `add_advertiser`, `process_ad`, and `terminate_program` commands here, paying close attention to the issues of data representation. We will leave stubs in place of the modules that support the `process_pmt` and `gen_bills` commands, and develop them later in the project.

The new_advertiser command. The `new_advertiser` command requires you to add a new advertiser to `ad_list`. In addition to updating `ad_list`, the `add_new_advertiser` operation also returns the name of the new advertiser. Why? It is likely that the next operation that the user will want to perform is to enter an ad for this advertiser; thus this advertiser's name will act as the "default" unless the user says otherwise. Here is the specification for `add_new_advertiser`:

`add_new_advertiser ( ad_list, advertiser )` – Level 1
Function. Adds new advertiser to `ad_list`. Returns the advertiser name for use in next operation.
Input. `ad_list` of `list_type`.
Preconditions. `ad_list` has been created.
Output. `ad_list` and `advertiser`.
Postconditions. Record for advertiser exists in `ad_list`.

The algorithm follows. Note that the user inputs are encapsulated by operations – `get_advertiser_name` and `get_address` – to make it simpler to modify the user interface later.

```
algorithm add_new_advertiser

get_advertiser_name ( advertiser )
```

```
check with retrieve_element if the advertiser exists in the ad_list

if advertiser not found in ad_list
    /* add this advertiser */
    put the advertiser name in the key field
    get_address ( ad_rec -> address )
    initialize ads field to empty state
    initialize balance field to zero
    insert_element ( ad_list, ad_rec )
    print success message to user
else
    print error message (advertiser already in the list)
```

The accept_ad command.
The bill_advertiser and receiver_pmt commands.
The quit command.

Circular linked lists

- Linear lists do not allow us to access the nodes that precede a given node
- We always need to access the elements of the lists starting with a special node (or head of the list)
- Circular list is one where the last node points to the first node in its `next` field instead of containing a `NULL`
  - Every node in the list has a successor, with the last element succeeded by the first
  - It is more like a ring of elements
- Advantage of circular linked list
  - Both ends of the list can be accessed by just one pointer
  - If `list` points to the last element in the circular list, the first element can be accessed by `list -> next`
- The structure of each node in the list stays the same as the linear linked list ADT
- The empty list is still shown by the value of `list` being `NULL`, so that the functions `create_list` and `empty_list` stay the same as well
- The functions that involve traversal of linked lists do change as the last element in the list is not `NULL` any more
- Printing a circular list
  - Requires a temporary pointer in the function to point to the current element
  - The temporary pointer starts printing with the first element and stops when it has reached the last element
  - We will always print one node ahead of the pointer, and stop when the pointer comes full circle

```
void print_list ( list_type * list )
{
    list_element_type * ptr;          /* Temporary pointer within the list  */

    /* Prints the contents of the entire list                               */

    if ( list )                       /* is not empty                       */
        for ( ptr = list->next; ptr != list; ptr = ptr->next)
            print_info ( ptr -> info );
}
```

- Finding a list element

- We will develop a function `find_element` that will return the pointer to the element, if it exists, and `NULL` if it does not
- If the element is found, and we need to make any modifications to it (such as deleting it), we may also want to get the address of its *logical predecessor*, which may be the last element if the element found is the smallest element in the list and exists at the beginning of the list

```
#define TRUE  1
#define FALSE 0

list_element_type * find_element ( list_type * list, key_type key_val, \
        list_element_type * pred_loc )
{
    /* list is a pointer to the last node in the circular list        */
    /* If key_val is found in the list, return a pointer to the node  */
    /*    with key key_val and set pred_loc to point to the preceding */
    /*    list node.                                                  */
    /* If key_val is not found in any list node, return NULL and set  */
    /*    pred_loc to point to the logical predecessor of a node with */
    /*    the key key_val                                             */

    list_element_type *ptr;    /* Keep track of the element being looked at */

    /* Set up the search                                                    */

    ptr = NULL;               /* Default for return                         */
    pred_loc = list;          /* Pointer to the previous element            */

    /* If not a special case (empty list or key_val > all keys in the list  */
    /*    search for key_val                                                */

    if ( list /* is not empty */ && key_val <= list->key )
    {
        ptr = list->next;
        while ( ( ptr->key < key_val ) && ( ptr != list ) )
        {
            pred_loc = ptr;
            ptr = ptr->next;
        }

    /* Set output parameters                                                */

    if ( ptr -> key != key_val ) /* Key value not found */
        ptr = NULL;

    return ( ptr );
}
```

- Inserting into a circular list
  - Algorithm similar to that for linked list insertion
  - Special cases include
    - ∗ Inserting into an empty list
      - · Make sure that the next field of the new node points back to itself
    - ∗ Inserting at the end of the list

· In addition to modifying the `next` field of the predecessor node, we have to make sure that the `list` points to the newly added node (the last node in the list)

```
/* Add new element to list, leaving key value-ordered structure of list       */
/* intact.  list points to the last node in a circular linked list.           */
list_type * insert_element ( list_type * list, list_element_type * new_element,\
                             int * error )
{
    list_element_type * ptr;          /* Location of element in the array     */
    list_element_type * pred_loc;     /* Previous location                    */
    list_element_type * tmp;

    *error = 0;        /* Error is initially false                          */

    /* Check if the lement already exists, if so, cannot insert new element */

    if ( location = find_element ( list, new_element -> key, pred_loc ) )
    {
        *error = TRUE;
        return ( list );    /* Error, list is returned unchanged            */
    }

    /* Make room for the new element by allocating memory                   */

    tmp = ( list_element_type * ) malloc ( sizeof ( list_element_type ) );
    *tmp = *new_element;

    /* Put new element in the proper place                                  */

    if ( empty_list ( list ) )
    {
        list = tmp;
        list -> next = list;
    }
    else
    {
        tmp -> next = pred_loc -> next;
        pred_loc -> next = tmp;

        /* If this is the last node in the list, reassign the last node      */

        if ( tmp -> key > list -> key )
            list = tmp;
    }

    return ( list );
}
```

- Deleting from a circular linked list
  - In the linear list version, deleting the first (smallest) element is a special case but in the circular linked list, it is not so
  - However, deleting the only node in a circular list is a special case
  - Also, deleting the largest element from the list is a special case

```
/* Remove the node containing the key delete_val from the linked list pointed */
/* to by list.  Assumes that this key is present in the list.               */
/* list is a pointer to the last node in the circular linked list           */

list_type * delete_element ( list_type * list, key_type delete_val, int * error)
{
    list_element_type * ptr;      /* Location of element in the array */
    list_element_type * pred_loc; /* Previous location                */

    *error = 0;       /* Error is initially false                          */

    /* Error if the element does not exist in the list                     */

    if ( ( ptr = find_element ( list, delete_val, pred_loc ) ) == NULL )
    {
        *error = TRUE;
        return ( list );    /* Error, list is returned unchanged            */
    }

    /* Check if this is the only element in the list                       */

    if ( pred_loc == ptr )
        list = NULL;
    else
    {
        pred_loc -> next = ptr -> next;
        if ( ptr == list )     /* Deleting largest list node    */
            list = pred_loc;
    }

    free ( ptr );

    return ( list );
}
```

- Other operations in the list ADT are left as an exercise

Linked lists with headers and trailers

- Special cases arise when we are dealing with first node or last node in the linked list ADT
- Problem can be simplified if we make sure that we *never* insert or delete at the ends of the list
- Easy to achieve by adding two placeholder nodes or dummy nodes at either end
- Header node
  - Placeholder node at the beginning of a list
  - Used to simplify list processing, or to contain information about the list, or both
- Trailer node
  - Placeholder node at the end of a list
  - Used to simplify list processing
- If a list of students is ordered by last name, for example, we may assume that there is no student named `"      "` (all blanks) or `"zzzzzz"`

- A version of `create_list` function that initializes the header and trailer nodes of such a list is given as

```
/* Add to declarations of list_element_type                                         */

#define  MINVALUE "      "
#define  MAXVALUE "zzzzzz"

/* Initialize a header and trailer node for the list.  The list will be            */
/* ordered alphabetically with respect to the key field.  It will be assumed       */
/* to be empty.                                                                     */

list_type * create_list ( list_type * list )
{
    /* Set up the header                                                            */

    list = ( list_element_type *) malloc ( sizeof ( list_element_type ) );
    strcpy ( list -> key, MINVALUE );

    /* Set up the trailer                                                           */

    list->next = ( list_element_type *) malloc ( sizeof ( list_element_type ) );
    strcpy ( ( list -> next ) -> key, MAXVALUE );
    ( list -> next ) -> next = NULL;

    return ( list );
}
```

- Checking for the list being empty
  - Our old routine will not work any more as we always have something in the list
  - Revised function to check for list being empty

```
int empty_list ( list_type * list )
{
    /* Return true if list is empty, false otherwise        */

    return ( ! strcmp ( ( list->next )->key == MAXVALUE ) );
}
```

- Finding an element in the list

```
#define TRUE  1
#define FALSE 0

list_element_type * find_element ( list_type * list, key_type key_val, \
        list_element_type * pred_loc )
{
    /* list is a pointer to the last node in the circular list        */
    /* If key_val is found in the list, return a pointer to the node  */
    /*    with key key_val and set pred_loc to point to the preceding */
    /*    list node.                                                  */
    /* If key_val is not found in any list node, return NULL and set  */
    /*    pred_loc to point to the logical predecessor of a node with */
    /*    the key key_val                                             */
```

```
        list_element_type *ptr;    /* Keep track of the element being looked at */

        /* Set up the search                                                     */

        ptr = list;               /* Beginning of list                           */

        /* Search for node containing key_val until:                             */
        /*    1. we reach key_val's place in the list, or                        */
        /*    2. we reach the trailer node                                       */

        while ( ptr -> key < key_val )
        {
            pred_loc = ptr;
            ptr = ptr->next;
        }

        if ( ptr -> key != key_val ) /* Key value not found */
            ptr = NULL;

        return ( ptr );
    }
```

- Inserting an element into the list
  - We have to worry only about the case when the element is inserted in the middle of the list
  - No value for the key field will be smaller than that in the header node or larger than the trailer node

```
/* Add new element to list, leaving key value-ordered structure of list       */
/* intact.  The list has a header and a trailer node.                         */
list_type * insert_element ( list_type * list, list_element_type * new_element,\
                             int * error )
{
    list_element_type *ptr;          /* Location of element in the array     */
    list_element_type *pred_loc;     /* Previous location                    */
    list_element_type *tmp;

    *error = 0;       /* Error is initially false                            */

    /* Check if the lement already exists, if so, cannot insert new element */

    if ( location = find_element ( list, new_element -> key, pred_loc ) )
    {
        *error = TRUE;
        return ( list );    /* Error, list is returned unchanged             */
    }

    /* Make room for the new element by allocating memory                    */

    tmp = ( list_element_type * ) malloc ( sizeof ( list_element_type ) );
    tmp -> info = *new_element;

    /* Reassign the pointers                                                 */

    tmp -> next = pred_loc -> next;
    pred_loc -> next = tmp;
```

```
        return ( list );
    }
```

- Deleting an element

```
/* Remove the node containing the key delete_val from the linked list pointed */
/* to by list.  Assumes that this key is present in the list.                */

list_type * delete_element ( list_type * list, key_type delete_val, int * error)
{
    list_element_type * ptr;      /* Location of element in the array */
    list_element_type * pred_loc; /* Previous location                */

    *error = 0;       /* Error is initially false                         */

    /* Error if the element does not exist in the list                    */

    if ( ( ptr = find_element ( list, delete_val, pred_loc ) ) == NULL )
    {
        *error = TRUE;
        return ( list );    /* Error, list is returned unchanged            */
    }

    pred_loc -> next = ptr -> next;
    free ( ptr );

    return ( list );
}
```

- Printing a list
  - Take care of the nodes that always exist
  - Header and trailer nodes are internal to the implementation and should be ideally hidden from the user

```
void print_list ( list_type * list )
{
    list_element_type *ptr;

    ptr = list->next; /* First node with actual information; could be trailer */
    while ( ptr->key != MAXVALUE )
    {
        print_info ( ptr -> info );
        ptr = ptr->next;
    }
}
```

  - A recursive version of the same

```
void print_list ( list_type * list )
{
    list_element_type *ptr;

    ptr = list->next; /* Next node with actual information; could be trailer */

    if ( ptr->key != MAXVALUE )
```

```
        {
            print_info ( ptr -> info );
            print_list ( ptr );
        }
    }
```

- Count the number of executable lines of code for all the list operations (with different lists) and check whether there is a change in the efficiency of algorithms in different implementations in terms of Big-O
- Other uses of header nodes
  - May be used to keep data about the list (such as number of elements in the list)

Doubly linked lists

- What if you want to delete a given node, given only a pointer to the node
- How do you traverse a linked list in reverse
- Doubly linked lists allow us to traverse a list in both directions
- Each node of a doubly linked list contains three parts

  info The data to be stored in the list
  next Pointer to the following node
  prev Pointer to the previous node

- In a doubly linked list, each node has a successor and a predecessor
- The structure can be coded as

```
    typedef char   key_type;    /* Type of the key field                   */
    typedef struct node_type
    {
        key_type   key;         /* Key field                               */
        ...                     /* Other fields in the structure as needed */
        struct node_type *next; /* Pointer to the next element in the list */
        struct node_type *prev; /* Pointer to previous element in the list */
    } list_element_type;        /* Type of the structure of each element   */

    typedef list_element_type list_type;

    list_type * list;
```

- Creating the doubly-linked list with a header and a trailer node

```
list_type * create_list ( list_type * list )
{
    /* Set up the header node                              */

    list = ( list_type * ) malloc ( sizeof ( list_type ) );
    list -> key = MINVALUE;
    list -> prev = NULL;

    /* Set up the trailer node                             */

    list -> next = ( list_type * ) malloc ( sizeof ( list_type ) );
    ( list -> next ) -> key = MAXVALUE;
```

```
        ( list -> next ) -> prev = list;
        ( list -> next ) -> next = NULL;

        return ( list );
    }
```

- Finding an element in the doubly linked list
  - Now, we do not need to return the predecessor node as we can return to the same from the found node
  - The interface to the routine will be slightly modified

```
#define TRUE  1
#define FALSE 0

list_element_type * find_element ( list_type * list, key_type key_val, \
    int *found )
{
    /* list is a pointer to the header node in the doubly linked list        */
    /* If key_val is found in the list, return a pointer to the node with    */
    /*    key key_val                                                         */
    /* If key_val is not found in any list node, return 0 in found and the   */
    /*    logical successor of the node with the key key_val                 */

    list_element_type *ptr;    /* Keep track of the element being looked at  */

    /* Set up the search                                                     */

    ptr = list;               /* Beginning of list                           */

    /* Search for node containing key_val until:                             */
    /*    1. we reach key_val's place in the list, or                        */
    /*    2. we reach the trailer node                                       */

    while ( ptr -> key < key_val )
        ptr = ptr->next;

    *found = ( ptr -> key == key_val );

    return ( ptr );
}
```

- Inserting an element into the list
  - Whenever we insert an element, we have to modify pointers in both the predecessor and well as the successor nodes
  - No value for the key field will be smaller than that in the header node or larger than the trailer node
  - On return from `find_element`, we will have a pointer to the logical successor node

```
/* Add new element to list, leaving key value-ordered structure of list       */
/* intact.  The list has a header and a trailer node.                         */
list_type * insert_element ( list_type * list, list_element_type * new_element,\
                             int * error )
{
    list_element_type *ptr;          /* Location of element in the array     */
    list_element_type *tmp;
    int found;                       /* Whether the element already exists   */

    *error = 0;       /* Error is initially false                            */

    /* Check if the element already exists, if so, cannot insert new element */

    ptr = find_element ( list, new_element -> key, &found );
    if ( found )
    {
        *error = TRUE;
        return ( list );    /* Error, list is returned unchanged             */
    }

    /* Make room for the new element by allocating memory                    */

    tmp = ( list_element_type * ) malloc ( sizeof ( list_element_type ) );
    tmp -> info = *new_element;

    /* Reassign the pointers                                                 */

    tmp -> prev = ptr -> prev;
    tmp -> next = ptr;
    ptr -> prev -> next = tmp;
    ptr -> prev = tmp;

    return ( list );
}
```