C Programming

Linked Lists

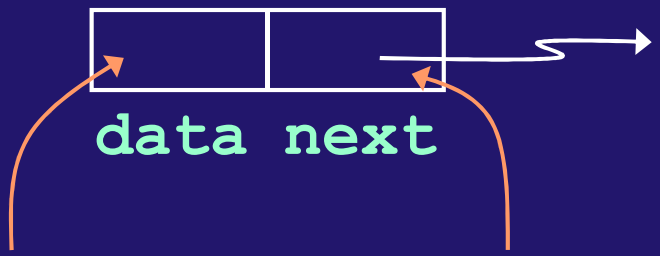
Self-Referential Structures

- Self-referential structures have pointer members that hold the address of the same structure type.
  - The pointer members allow the linking together of an unspecified number of such structures.
  - Example:

```
struct list {
   int          data;
   struct list *next;
};
```

Pictorial Representation

- To help us understand and think about self-referential structures, we use pictures:



The data members.

The pointer member.

An Example

```
struct list {
    int          data;
    struct list *next;
}
struct list a, b, c;
a.data = 1;
b.data = 2;
c.data = 3;
a.next = b.next = c.next = NULL;
```

The result of these declarations and initializations is pictured below:

Continuation of Example

```
a.next = &b;
b.next = &c;
```

Now we can use the links to retrieve data from successive elements.

```
a.next->data          has a value of 2
a.next->next->data    has a value of 3
```

Linear Linked Lists

- A linked list has a head pointer that addresses the first element of the list.
  - Then the pointer member in each structure in the list points to a successor structure.
  - The last structure has its pointer member set to NULL.
- Typically, a linked list is created dynamically.

Dynamic Storage Allocation

- "Dynamic" storage allocation refers to allocation of storage during program execution time, rather than during compile time.
  - Utility functions such as `malloc()` are provided in the standard library to allocate storage dynamically.
    - `malloc()` stands for "memory allocation".

Header File for Example of Dynamic Creation of a Linked List

```
#include <stdio.h>
typedef   char DATA;   /* use char in examples */
struct linked_list {
    DATA                    d;
    struct linked_list   *next;
};
typedef struct linked_list   ELEMENT;
typedef ELEMENT *            LINK;
```

Example of Dynamic Allocation of a Linked List

```
head = malloc(sizeof(ELEMENT));
head->d = 'n';
head->next = NULL;
```

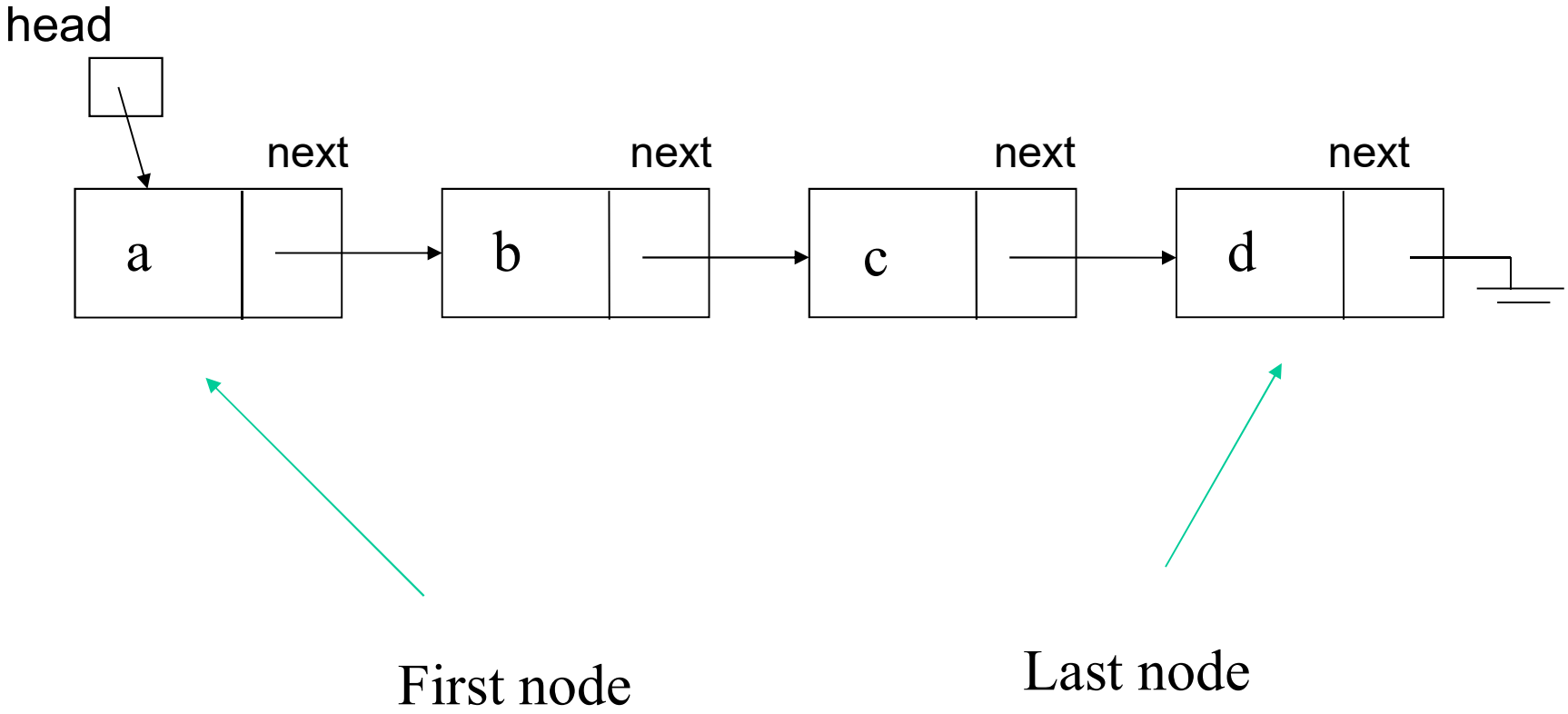

```
head->next = malloc(sizeof(ELEMENT));
head->next->d = 'e';
head->next->next = NULL;
```


```
head->next->next = malloc(sizeof(ELEMENT));
head->next->next->d = 'w';
head->next->next->next = NULL;
```

List Operations

- Basic List Operations
  - Creating a list
  - Counting the elements
  - Looking up an element
  - Inserting an element
  - Deleting an element

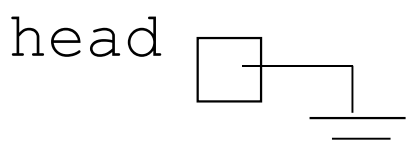
Singly Linked Lists



Empty List

- Empty Linked list is a single pointer having the value of NULL.

```
head = NULL;
```



Counting the Elements in a List

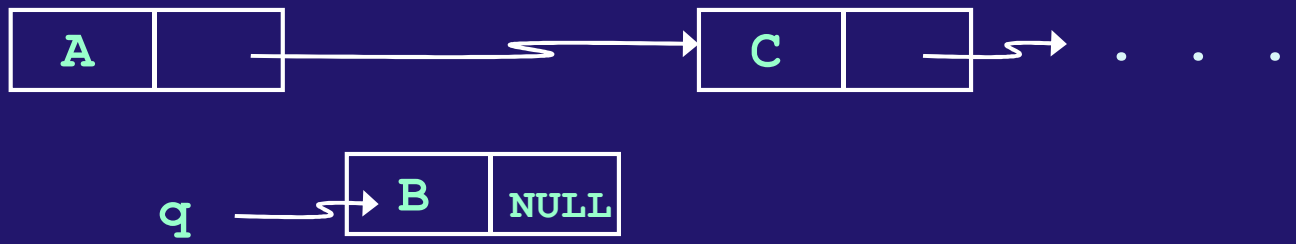
```
/* Count the elements recursively */

#include "list.h"

int count(LINK head)
{
    if (head == NULL)
        return 0;
    else
        return(1 + count(head->next));
}
```

Lookup c in the List Pointed to by head

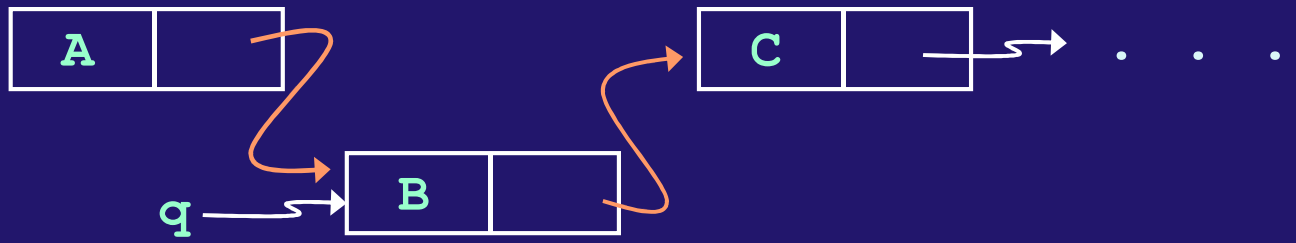
```
#include "list.h"

LINK lookup(DATA c, LINK head)
{
    if (head == NULL)
        return NULL;
    else if (c == head->d)
        return head;
    else
        return(lookup(c, head->next));
}
```

Illustration of Insertion of a New List Element

Before insertion:

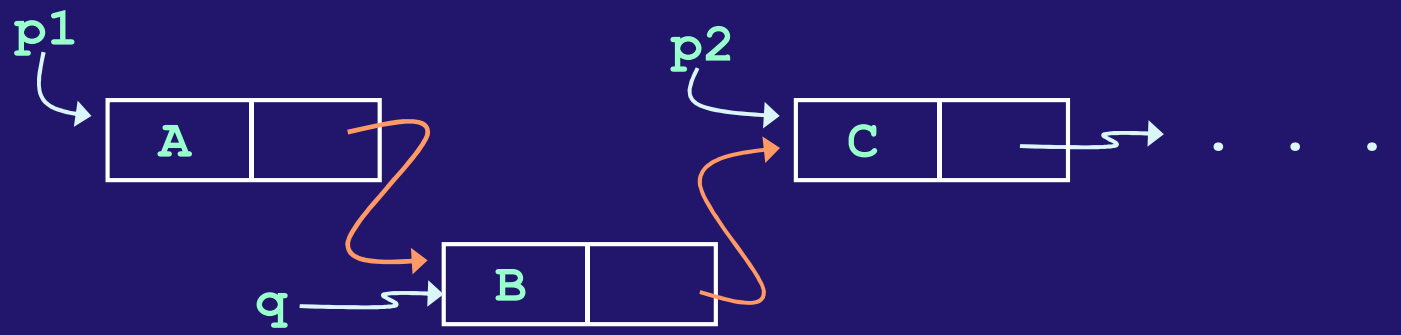


After insertion:



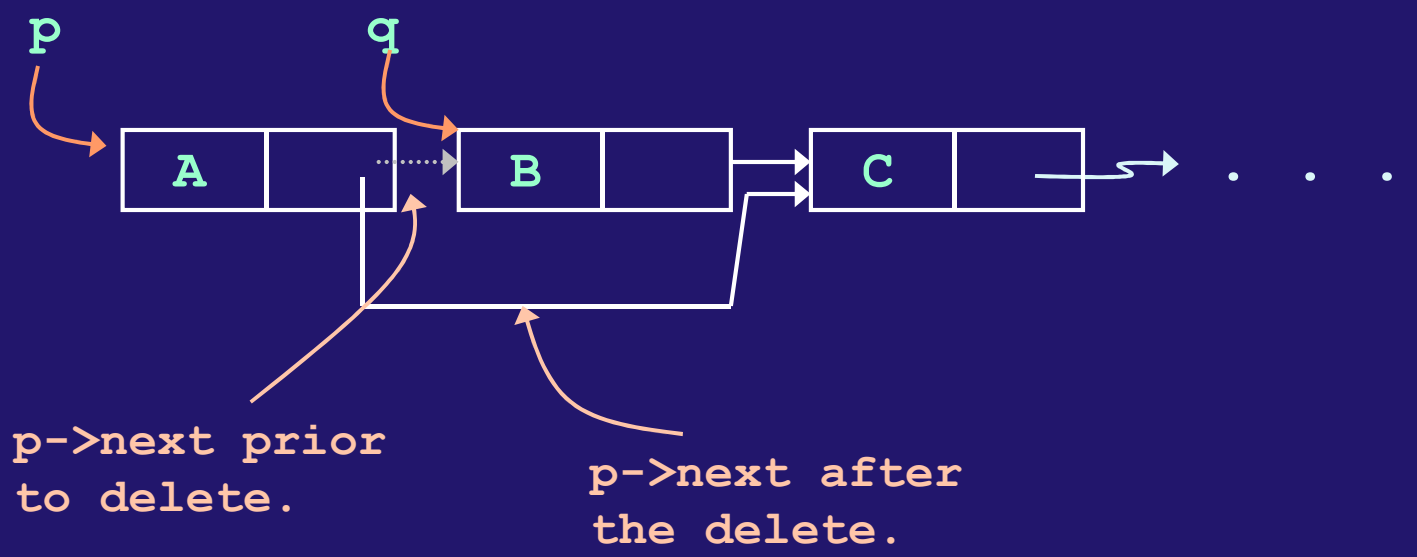
Recursive Function to Insert an Element in a List

```
#include "list.h"

void insert(LINK p1, LINK p2, LINK q)
{
    p1->next = q; /* insertion */
    q->next = p2;
}
```



Deleting an Element from a List

```
p->next = q->next;
```



Note that the deleted node is now garbage (of no use). Its storage can be returned to the system by using the standard library function free().

Recursive Deletion of a List

```
/* Recursive deletion of a list. */

#include "list.h"

void delete_list(LINK head)
{
    if (head != NULL) {
        delete_list(head->next);
        free(head);  /* release storage */
    }
}
```

Demonstration Structures

```
struct date_rec {
   int month;
   int day;
   int year;
};    /* year is stored as yyyy */

struct friend_rec {
   char              fname[15]; /* assume first name is stored in caps */
   char              phone[9];   /* local number only, such as 555-1234 */
   struct date_rec birthday;
};

struct my_friends {
   struct friend_rec   a_friend;
   struct my_friends *next;
};
. . .
struct date_rec      dob;
struct friend_rec   friend;
struct my_friends *head, *current, *new;
```