Structures and List Processing
Self-referential Structures

- Structures with pointer members that refer to the structure containing them

```c
struct list {
    int data;
    struct list *next; /* next is called a LINK */
} a;
```

- Dynamic data structures
  - Unspecified number of structures nested together
  - Each structure is linked to a succeeding structure by a link
struct list a, b, c;

a.data = 1;
b.data = 2;
c.data = 3;

a.next = b.next = c.next = NULL;
Example (continued)

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

printf("%d\n%d\n", a.next->data, a.next->next->data);
```
Linear Linked Lists

- A linear linked list is like a clothes line on which the data structures hang sequentially.
- Head pointer addresses the first element
- Each element points at a successor element
- Last element has a link value NULL
Example

/* In file list.h */

typedef char DATA;
struct linked_list {
    DATA d;
    struct linked_list *next;
};
typedef struct linked_list ELEMENT;
typedef ELEMENT *LINK;
Example: storing three characters n, e, w

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

![Diagram](image)
Example: storing three characters n, e, w

head->next = malloc(sizeof(ELEMENT));
head->next->d = 'e';
head->next->next = NULL;
Example: storing three characters n, e, w

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

```
head  n  e  w  NULL
```
(Linear) List Operations

- Creating a list
- Counting the elements
- Looking up an element
- Concatenating two lists
- Inserting an element
- Deleting an element
List Creation using Iteration

LINK s_to_l(char s[ ])
{
    LINK head = NULL, tail;
    int i;
    if (s[0] != '\0') {/* first element */
        head = malloc(sizeof(ELEMENT));
        head -> d = s[0];
        tail = head;
        for (i = 1; s[i] != '\0'; ++i) {/* add to tail */
            tail -> next = malloc(sizeof(ELEMENT));
            tail = tail -> next;
            tail -> d = s[i];
        }
        tail -> next = NULL;/* end of list */
    }
    return head;
}
s_to_I(“AB”)

```c
head = malloc(sizeof(ELEMENT));
head -> d = s[0];
tail = head;
```

```c
tail -> next = malloc(sizeof(ELEMENT));
tail = tail -> next;
tail -> d = s[i];
```

```c
tail -> next = NULL; /* end of list */
```
List Creation using Recursion

```c
LINK string_to_list(char s[]) {
    LINK head;
    if (s[0] == '\0') { /* base case */
        return NULL;
    }
    else {
        head = malloc(sizeof(ELEMENT));
        head -> d = s[0];
        head -> next = string_to_list(s + 1);
        return head;
    }
}
```
Counting the elements

/* Count a list iteratively. */
int count_it(LINK head)
{
    int cnt = 0;

    for (; head != NULL; head = head -> next) {
        ++cnt;
    }

    return cnt;
}
/* Count a list recursively. */

int count(LINK head)
{
    if (head == NULL) {
        return 0;
    }
    else {
        return (1 + count(head->next));
    }
}
void concatenate(LINK a, LINK b) {
    assert(a != NULL);
    if (a -> next == NULL) {
        a -> next = b;
    } else {
        concatenate(a -> next, b);
    }
}
Insertion

- insertion in lists takes a constant time (once the position in the list is found)

- insertion in an array takes time proportional to the length of the array (on average)
Recursive functions for list processing

```c
void generic_recursion(LINK head)
{
    if (head == NULL) {
        do the base case
    }
    else {
        do the general case and recur with
        generic_recursion(head -> next)
    }
}
```
void insert(LINK p1, LINK p2, LINK q)
{
    assert(p1 -> next == p2);
    p1 -> next = q; /* insert */
    q -> next = p2;
}

Insert an element
Delete an element

```c
void delete(LINK p)
{
    p->next=p->next->next;
    /*element containing B becomes a garbage*/
}
```
Delete a list

```c
void delete_list(LINK head)
{
    if (head != NULL) {
        delete_list(head -> next);
        free(head); /* release storage */
    }
}
```
Stacks
#include <stdio.h>
#include <stdlib.h>
#define EMPTY 0
#define FULL 10000
typedef char data;
typedef enum {false, true} boolean;
typedef struct elem { /* an element on the stack */
    data d;
    struct elem *next;
};
typedef struct stack elem;
struct stack {
    int cnt;
    /* a count of the elements */
    elem *top;
    /* ptr to the top element */
};
typedef struct stack stack;
void initialize(stack *stk);
void push(data d, stack *stk);
data pop(stack *stk);
data top(stack *stk);
boolean empty(const stack *stk);
boolean full(const stack *stk);
void initialize(stack *stk)  
{
    stk->cnt = 0;
    stk->top = NULL;
}

void push(data d, stack *stk)  
{
    elem*p;
    p = malloc(sizeof(elem));
    p -> d = d;
    p -> next = stk->top;
    stk->top = p;
    stk->cnt++;
}
data pop(stack *stk) 
{
    data d;
    elem*p;
    d = stk->top -> d;
    p = stk->top;
    stk->top = stk->top -> next;
    stk->cnt--; 
    free(p); 
    return d;
}
stack operations

data top(stack *stk)
{
    return (stk-> top -> d);
}

boolean empty(const stack *stk)
{
    return ((boolean) (stk-> cnt== EMPTY));
}

boolean full(const stack *stk)
{
    return ((boolean) (stk-> cnt== FULL));
}
Example

```c
#include "stack.h"
int main(void)
{
    char str[ ] = "My name is Joanna Kelly!"
    int i;
    stack s;
    initialize(&s); /* initialize the stack */
    printf(" In the string: %s\n", str);
    for (i = 0; str[i] != '\0'; ++i)
        if (!full(&s))
            push(str[i], &s); /* push a char on the stack */
    printf("From the stack: ");
    while (!empty(&s))
        putchar(pop(&s)); /* pop a char off the stack */
    putchar('\n');
    return 0;
}
```