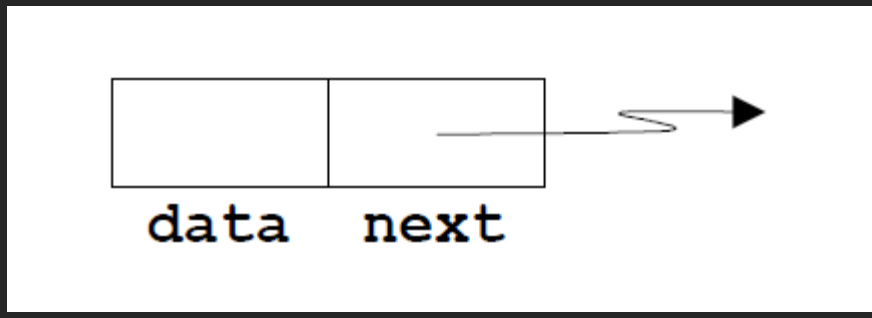
Structures and List Processing

Self-referential Structures

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

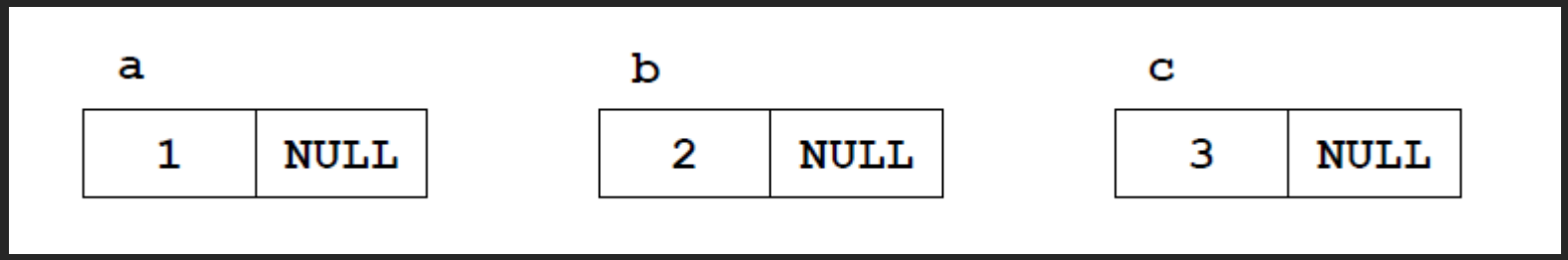



```
struct list {
        int data;
        struct list *next;  /* next is called a LINK */
} a;
```

- Dynamic data structures
  - Unspecified number of structures nested together
  - Each structure is linked to a succeeding structure by a link

Example

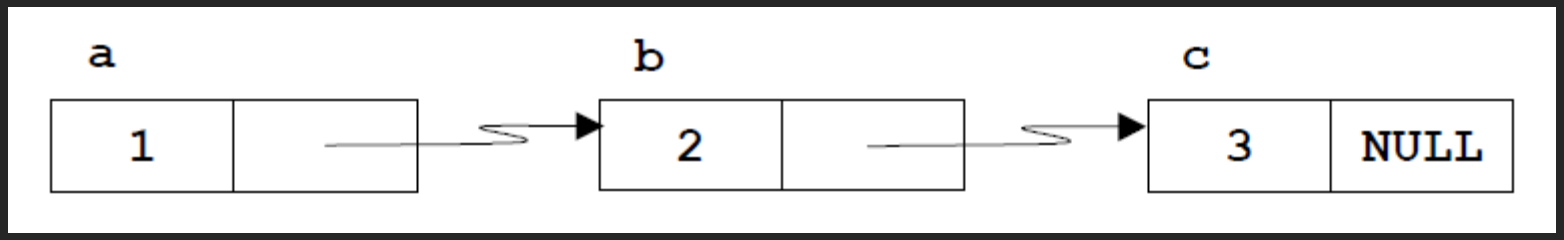
```
struct list a, b, c;
a.data = 1;
b.data = 2;
c.data = 3;
a.next = b.next = c.next = NULL;
```



Example (continued)

a.next = &b;

b.next = &c;

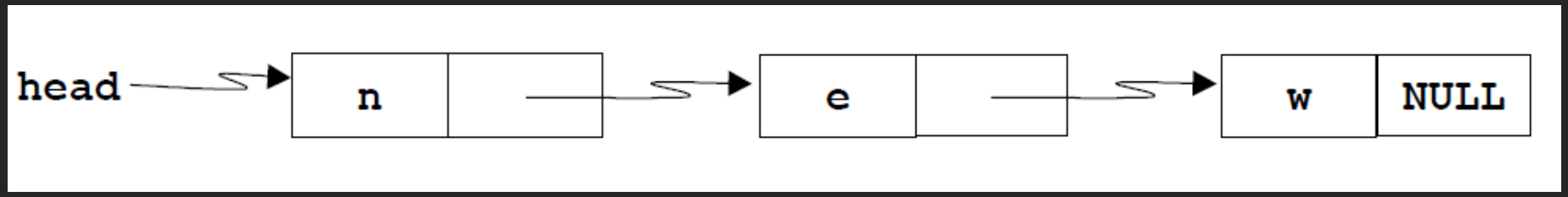


printf(“%d\n%d\n”, a.next->data, a.next->next->data);

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



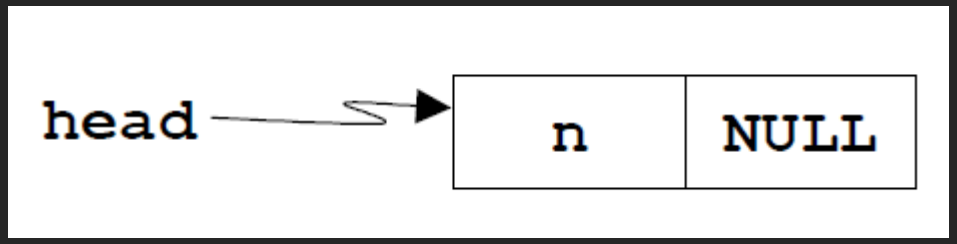
Example

```
/* In file list.h */

typedef char DATA;
struct linked_list {
        DATA d;
        struct linked_list *next;
};
typedef struct linked_list ELEMENT;
typedef ELEMENT *LINK;
```

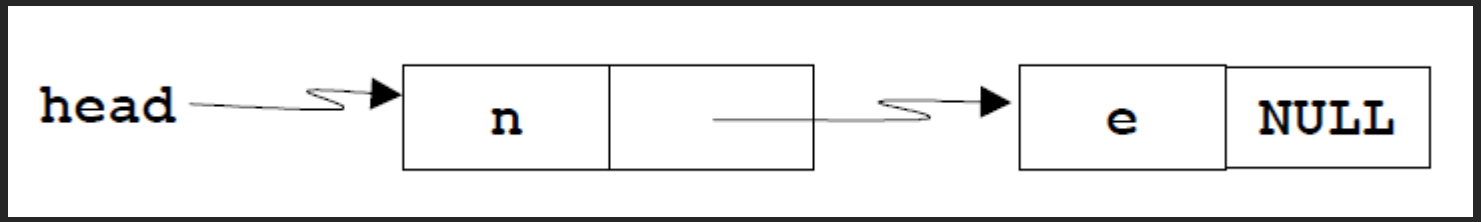
Example: storing three characters n, e, w

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



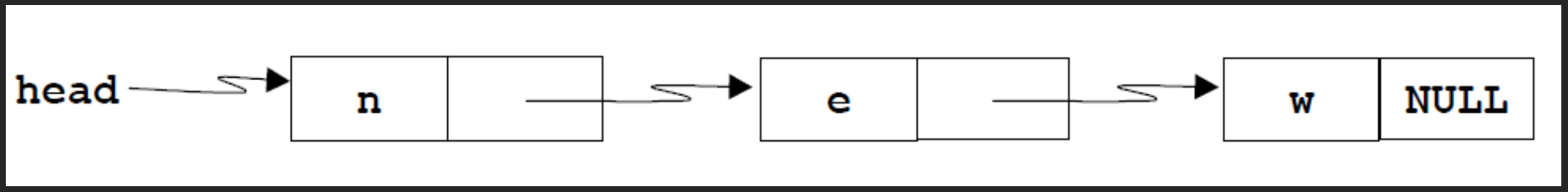
Example: storing three characters n, e, w

```
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;
```



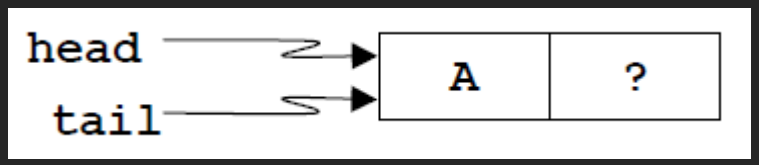
(Linear) List Operations

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

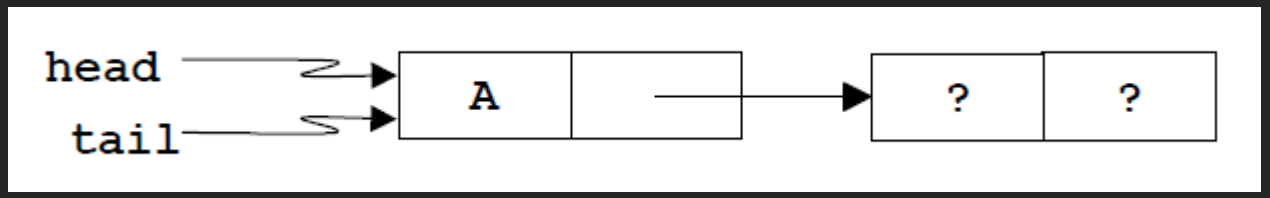
List Creation using Iteration

```
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;
}
```

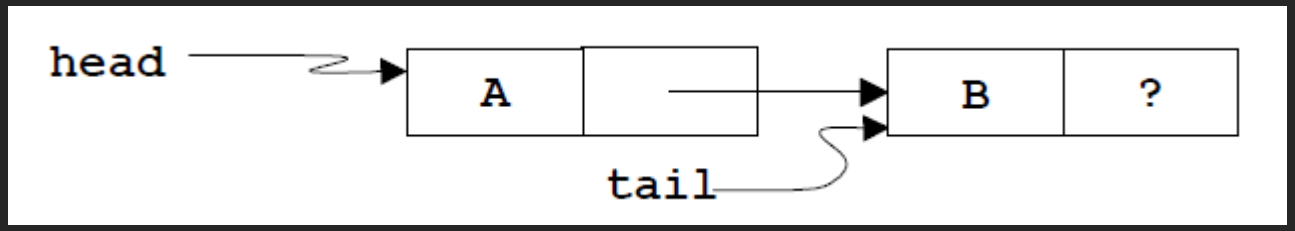
s_to_l("AB")

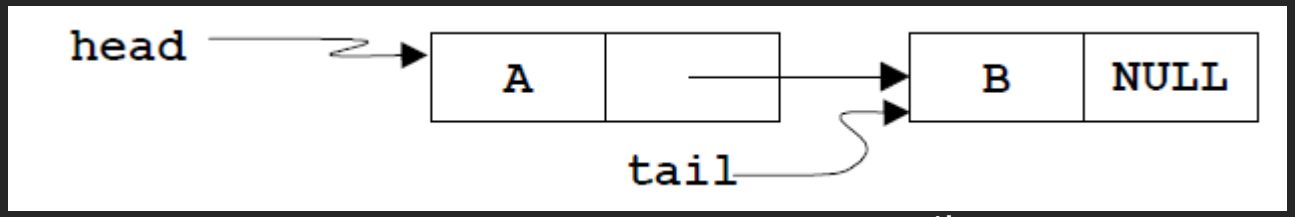



```
head = malloc(sizeof(ELEMENT));
head -> d = s[0];
tail = head;
```




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






```
tail -> next = NULL;/* end of list */
```

List Creation using Recursion

```
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;
}
```

Counting the elements

```
/* Count a list recursively. */
int count(LINK head)
{
        if (head == NULL) {
                return 0;
        }
        else {
                return (1 + count(head -> next));
        }
}
```

Concatenate list a and b with a as head

```
void concatenate(LINK a, LINK b)
{
        assert(a != NULL);
        if (a -> next == NULL) {
                a -> next = b;
        }
        else {
                concatenate(a -> next, b);
        }
}
```

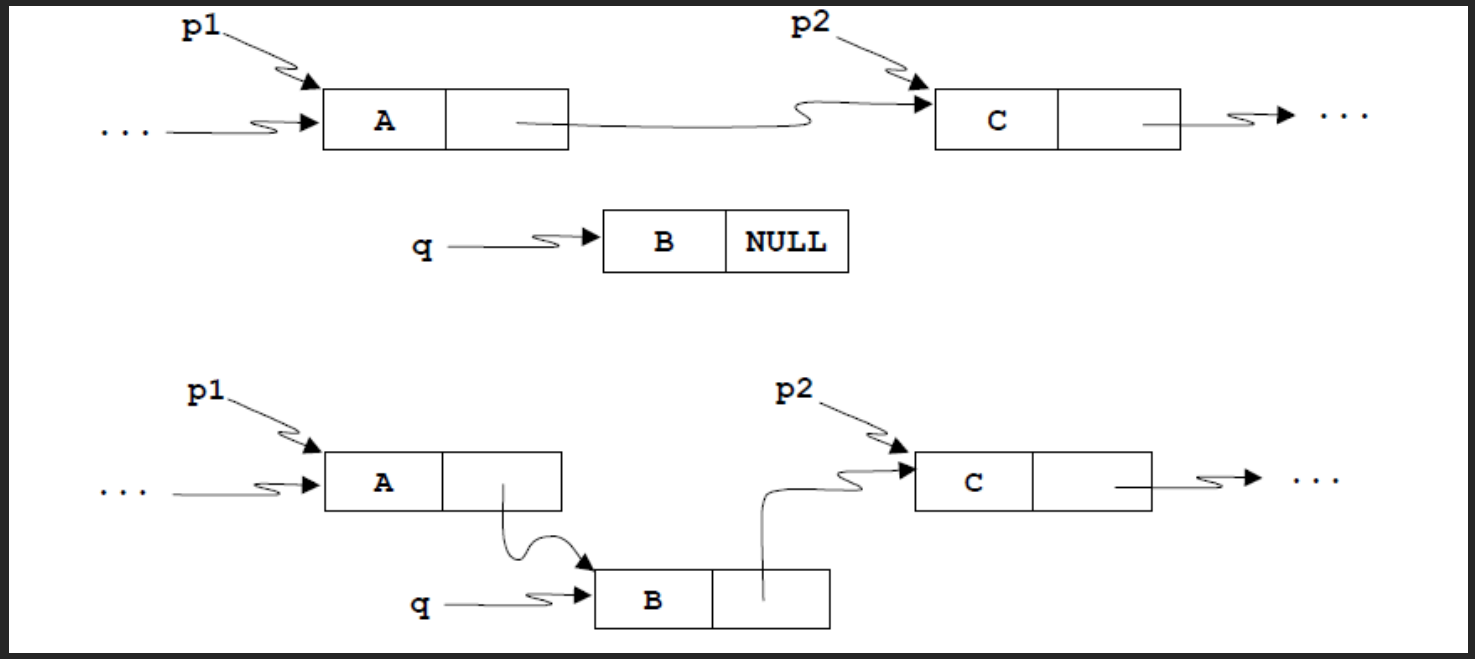
Insertion

- 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

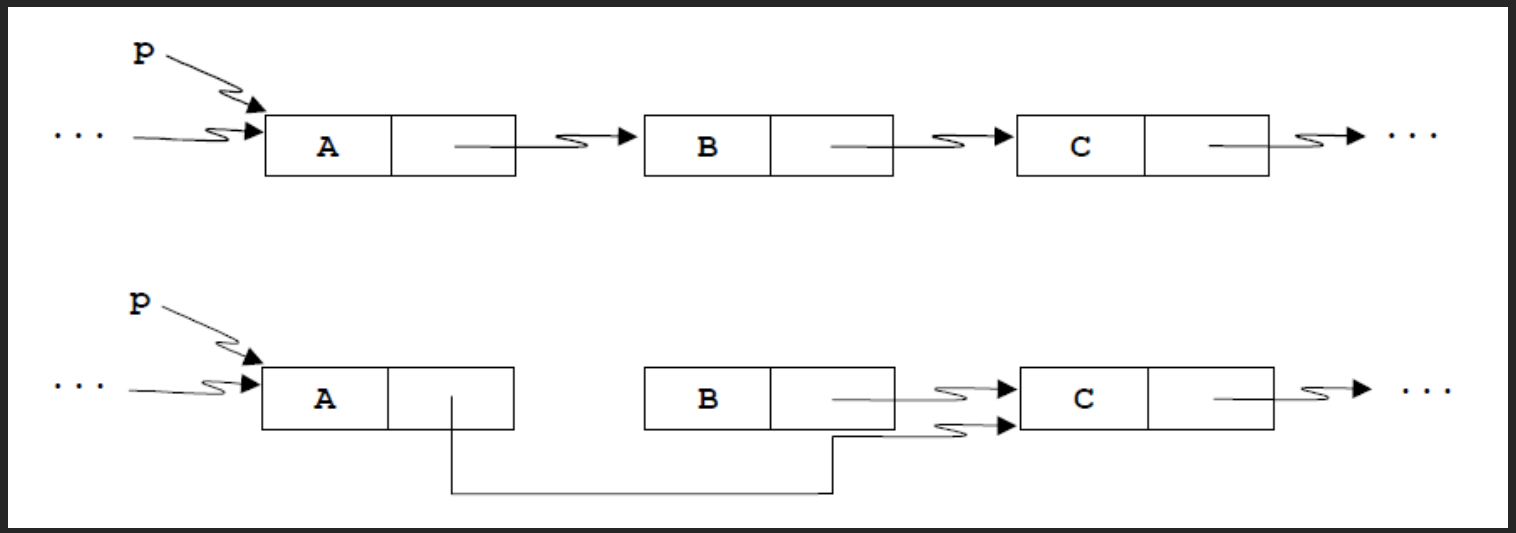
```
void generic_recursion(LINK head)
{
        if (head == NULL) {
                do the base case
        }
        else {
                do the general case and recur with
                generic_recursion(head -> next)
        }
}
```

Insert an element




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

Delete an element

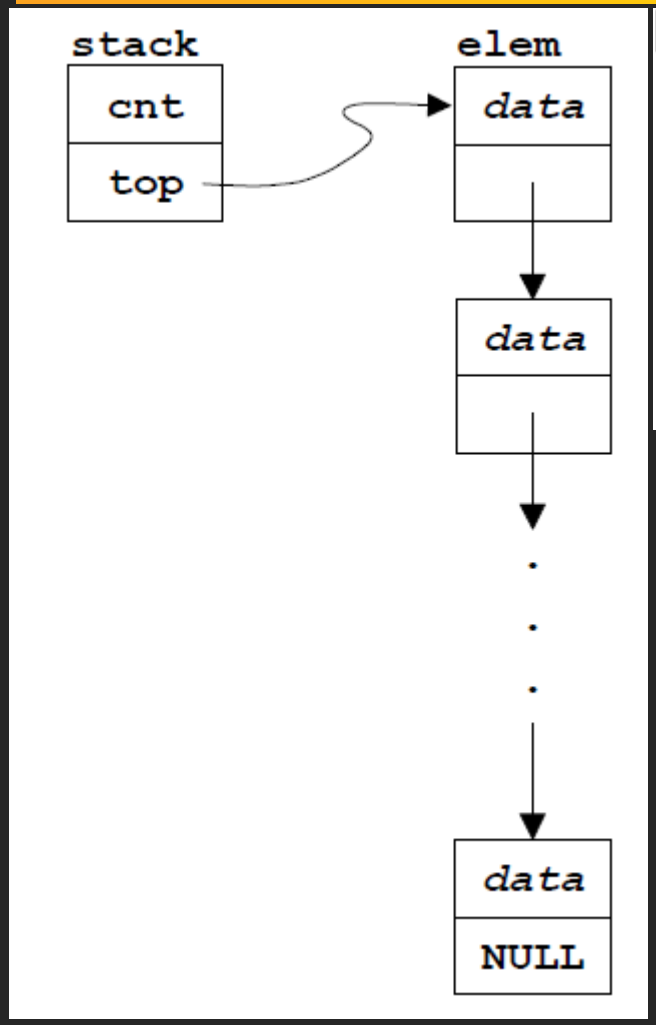


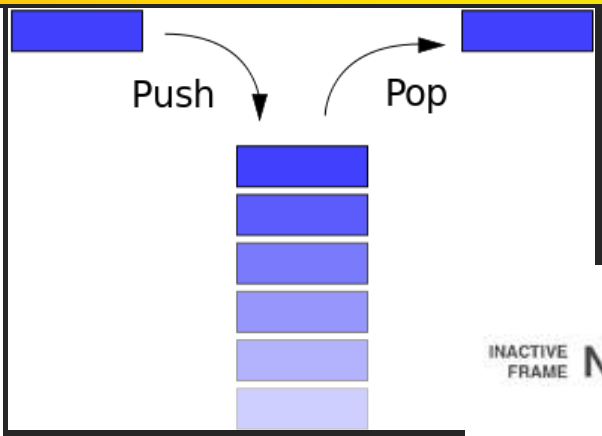

```
void delete(LINK p)
{
        p->next=p->next->next;
        /*element containing B becomes a garbage*/
}
```

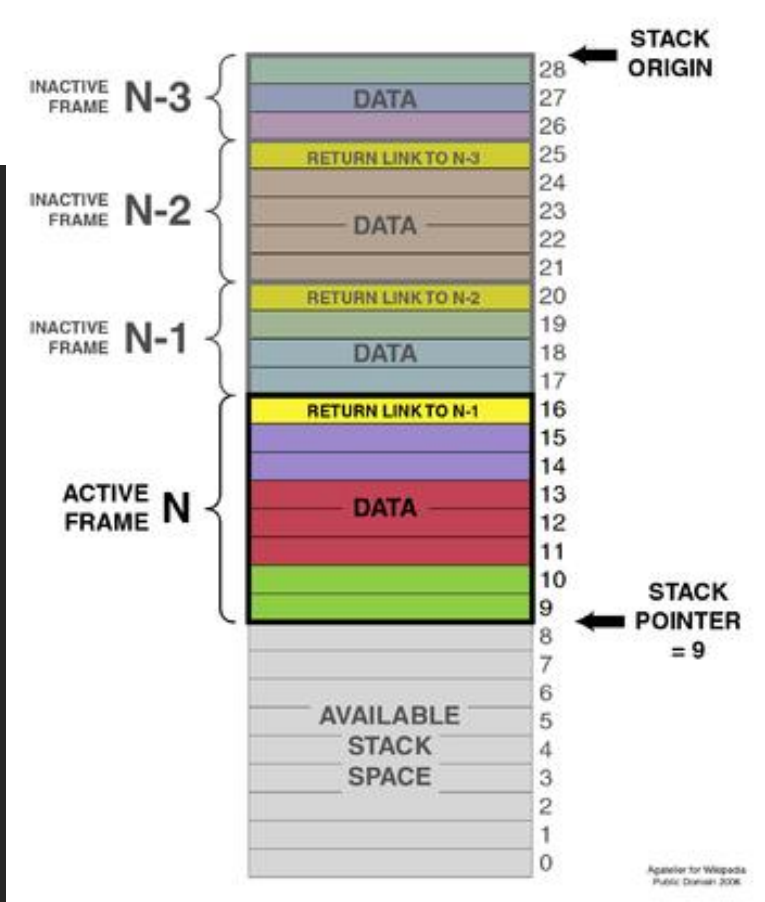
Delete a list

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

Stacks


stack
cnt
top
elem
data
data
data
NULL


Push
Pop


STACK ORIGIN
INACTIVE FRAME N-3
DATA
RETURN LINK TO N-3
INACTIVE FRAME N-2
DATA
RETURN LINK TO N-2
INACTIVE FRAME N-1
DATA
RETURN LINK TO N-1
ACTIVE FRAME N
DATA
STACK POINTER = 9
AVAILABLE STACK SPACE
28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

stack.h

```
#include <stdio.h>
#include <stdlib.h>
#define EMPTY 0
#define FULL 10000
typedef char data;
typedef enum {false, true} boolean;
struct elem { /* an element on the stack */
        data d;
        struct elem *next;
};

typedef struct elem elem;
```

```
struct stack {
        int cnt;
        /* a count of the elements */
        elem *top;
        /* ptrto 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);
```

stack operations

```
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++;
}
```

stack operations

```
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

```
#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;
}
```