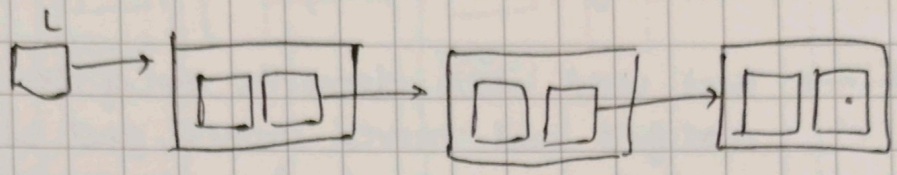
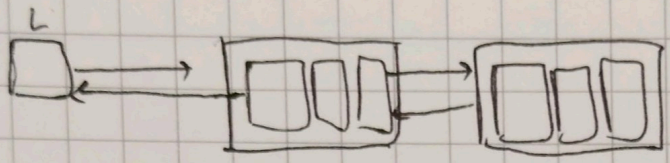
☆ LINKED LIST IMPLEMENTATION

SINGLY - LINKED



DOUBLY - LINKED

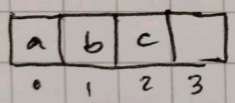


& CIRCULAR (Which I wont draw because screw dat.)

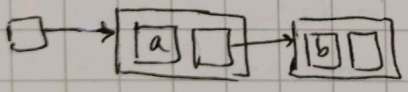
Characteristics :-

① Elements are not stored physically the same as logical representation

② There is overhead storage (extra storage needed to contain the link)

Eg. 

← one char elem = 1 byte



← one char elem = 9 bytes.

↳ TRAVERSAL

Has to begin from 1st node to the last node (In the case of singly-linked list)

WHEN TO USE ARRAY VS. LINKED LIST?

Array → Best used when number of elements are known and changes are rarely anticipated.

Linked list → Best used when elements are often modified and moved around, and when the maximum number of elements is not known or defined.

PRACTICE

```
typedef struct {
    char LName [16];
    char FName [24];
    char MI;
} Nametype;
```

```
typedef struct node {
    Nametype name;
    struct node * link;
} *LIST, nodetype;
```

tag name → struct node

*LIST → pointer to structure

nodetype → structure

3 data types generated :-
struct node , LIST , nodetype

* Note :
struct node ≈ nodetype
nodetype * ≈ LIST
struct node * ≈ LIST

(sideline, for reference)

Internet version

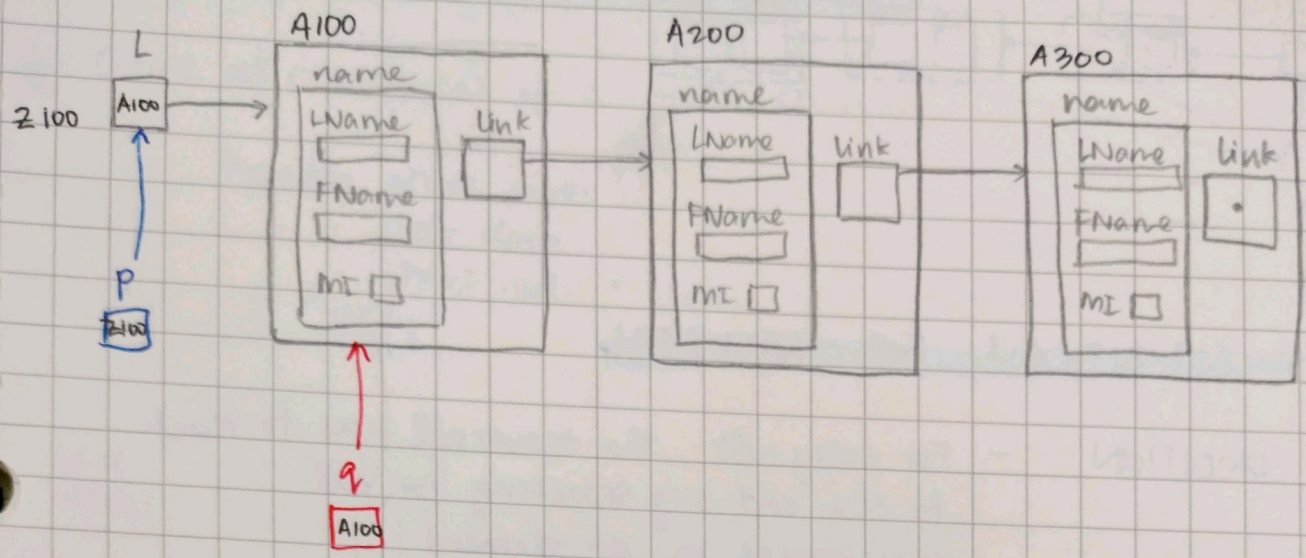
```
struct node {
    struct Name name;
    struct node * link;
};
```

PASS BY COPY :

```
    void displayList (LIST A)
→ void displayList (struct node *A)
```

PASS BY ADDRESS :

```
    void insertFirst (LIST *A, Nametype N)
→ void insertFirst (struct node **A,
                    struct Name N)
```




```
LIST *p;  // declaring *p. *p exists but not pointing
p = &L;   // making *p point to L

LIST q;   // q not pointing to anything yet
q = L;    // q pointing to first node
```

Q. ACCESS THE VARIABLE "MI" VIA L, q & p.

ANS.
a) via L — L → name.MI
b) via q — q → name.MI
c) via p — (*p) → name.MI

Q Statements to move q and p point to the next node.

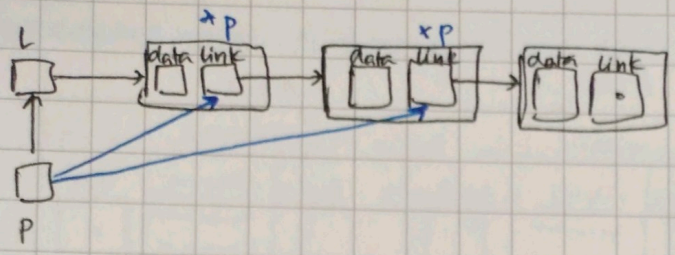
ANS.

```
q = q → link
p = &(*p) → link
```

lets break this down

Pointer of a pointer i.e the pointer holds the address of another pointer. The "link" portion of the node is a pointer. *p is an alias for the pointer pointing to the node with link. p holds (*p) → link's address. i.e p is pointing to (*p) → link.

Illustrated next page. →



LOOP CONDITIONS FOR TRAVERSAL

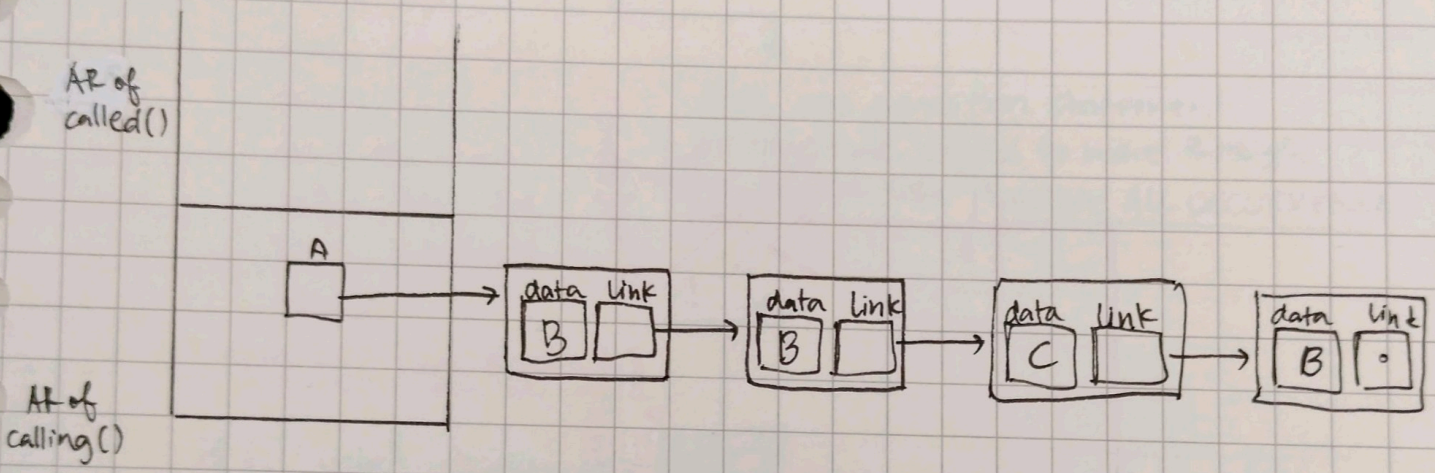
1 CONDITION - For every call, the traversal has to reach to the end e.g searching for all occurrences of an element.

2 CONDITIONS - When the traversal does not require to traverse the whole list to reach the end.
Eg. delete the first instance of an element.

PRACTICE EXERCISES & SIMULATIONS

Q: Delete all occurences of a given element.

```
typedef struct node {
        char data;
        struct node * link;
} * LIST;
```



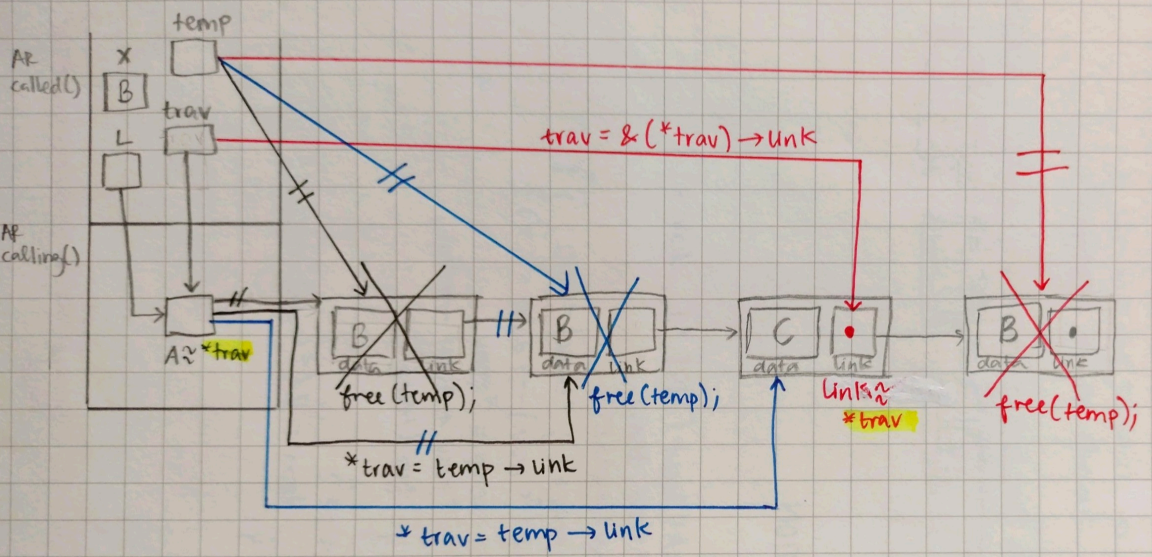
ANS. & SIMULATION

```
void deleteAllOccur (LIST *L, char x) {
    LIST *trav, temp;
    if (*trav != NULL) {
        for (trav = L; *trav != NULL; ) {
            if ((*trav) -> data == x) {
                temp = *trav;
                *trav = temp -> link;
                free (temp);
            }
            else {
                trav = &(*trav) -> link;
            }
    }
}
```

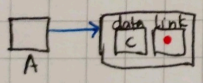
SIMULATION & DRAWING ON NEXT PAGE

deleting all occurrences

SIMULATION



Resulting LIST



CODE

```
void deleteAllOccur(LIST * L, char x) {
```

↳ deletion involved, pass by address

```
LIST * trav, temp;
```

↳ pointer pointer node

↳ pointer-node. will hold the address of the node to be deleted.

```
for (trav = L; *trav != NULL; ) {
```

↳ Note: In the execution stack, trav & L hold the same value; they are both pointers to pointer.

trav = L, not trav = &L.

↳ Only one condition statement because we expect to move through the entire list. "delete ALL OCCURRENCES"

```
if ((*trav) -> data == x) { // checking if data == x, value to be deleted.
    temp = *trav; // let temp hold the address of the node to be deleted;
    *trav = temp -> link;
```

↳ the link pointing to the current node

↳ Let it point to the node after the current node (the current node is the one that needs to be deleted

```
    free(temp); // deallocate the node that needs to be deleted.
}
else {
    trav = &(*trav) -> link;
```

↳ moving to the next node if current node doesnt need to be deleted.

```
}
}
```

Q. Same structure, write the code of the function that will delete the first two occurences

```
void deleteFirstTwo (LIST *L, char x){
    LIST *trav, temp;
    int count = 0;
    for (trav = L; *trav != NULL && count != 2; ){
        if ((*trav) -> data == x){
            temp = *trav;
            *trav = temp -> link;
            free (temp);
            count++;
        }
        else{
            trav = &(*trav) -> link;
        }
    }
}
```

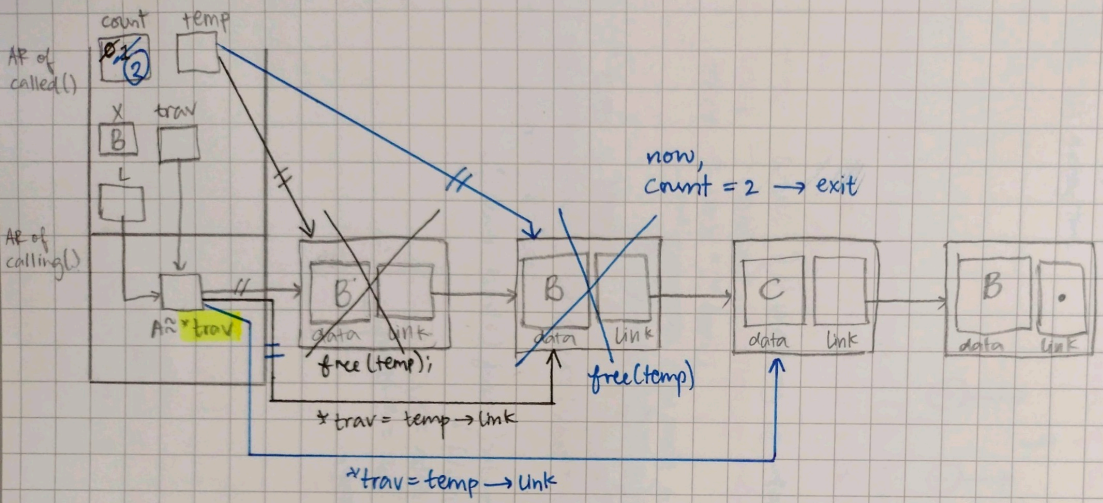
CODE

```
void deleteFirstTwo (LIST *L, char x){
    LIST *trav, temp;
    int count = 0; // counter to count no. of occurrences
    for (trav = L; *trav != NULL && count != 2; ){
        // two conditions because we dont expect to traverse the whole list
        if ((*trav) -> data == x){
            temp = *trav;
            *trav = temp -> link;
            free (temp);
            count++; // increment count if occurence is spotted
        }
        else{
            trav = &(*trav) -> link; // traverse if data ≠ x
        }
    }
}
```

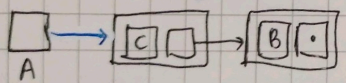
→ deletion code

deleting 2 occurrences

SIMULATION



RESULTING LIST



☆ PRACTICE

WITH TEST CASES, delete B

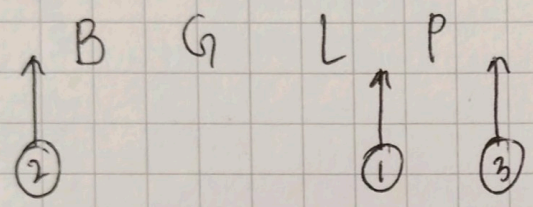
A B A C

A C B B

Q. WITH THE SAME DATASTRUCTURE, INSERT SORTED FUNCTION

LIST CONTAINS
B G L P

TEST CASES
① INSERT N
② INSERT A
③ INSERT S




```
typedef struct node {
    char data;
    struct node * link;
} LIST;
```

ANS.

```
void insertSorted (LIST *L, char x) {
    LIST temp, *trav;
    for (trav = L; *trav != NULL && (*trav) -> data < x;
                                    trav = &(*trav) -> link) { }
    temp = (LIST) malloc(sizeof (struct node));
    if (temp != NULL) {
        temp -> data = x;
        temp -> link = *trav;
        *trav = temp;
    }
}
```

CODE

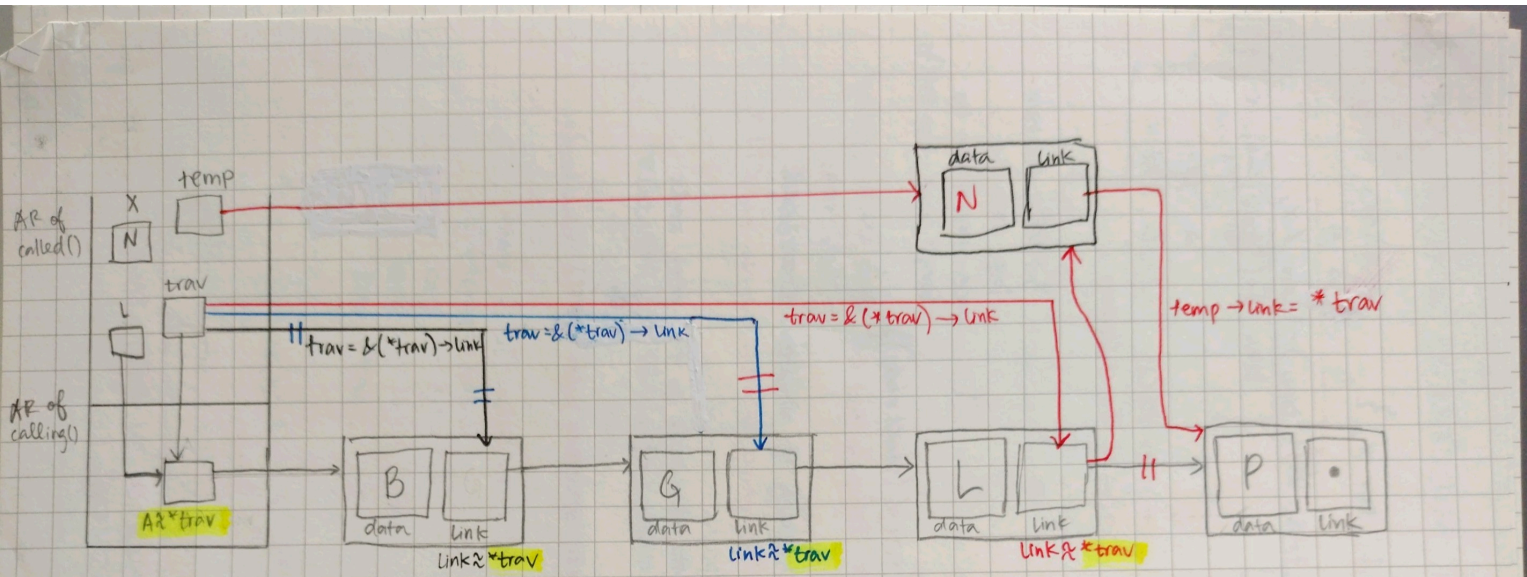
```
void insertSorted (LIST *L, char x) {
    LIST temp, *trav;
    for(trav=L; *trav != NULL && (*trav)->data < x;
        trav = &(*trav)->link) {}
    temp = (LIST)malloc(sizeof(struct node));
    // dynamically allocate temp so that it
       points to a new node
    if (temp != NULL) { // checking if allocation is successful
        temp -> data = x; // assigning values to the new node
        temp -> link = *trav;
                ↳ connects new node to the next node
                  in the list (OR NULL if last node)
        *trav = temp;
                ↳ connects previous node to the new node
    }
}
```

two conditions, we're not expected to traverse through the whole list everytime

compares ASCII value of the characters

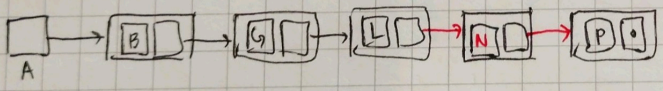
Butterfly notation loop, once the conditions are no longer met, loop exits.

SIMULATION ON NEXT PAGE
TEST CASE 1



N < P ✓

RESULTING LIST



★ PRACTICE WITH OTHER REMAINING 6 TEST CASES