- EXECUTION STACK -

Date: / /

* Definitions

- Execution Stack - LIFO (Last In, First Out)

- attributes & variables
  1.) name
  2.) value
  3.) data type
  4.) address
  5.) scope & lifetime
      - scope → program text
      - lifetime → program execution

* Problem to Code General

1.) Function Header
2.) Declare & Initialize V
    Appropriate Function
3.) Draw Execution Sta
4.) Code of the Funct

I - input
input to the function through parameters

O - output
return the data to the calling fn.

Date: / *Drawing Execution Stack*

* Problem Specifications : Pass by Value

The function will exchange the values of 2 given integers

```
void exchange (int x, int y) {
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

* Function Call:

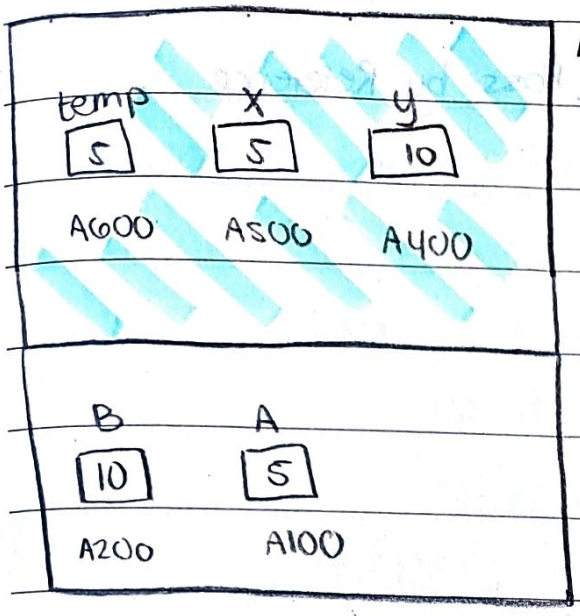
```
int A = 5;
int B = 10;
exchange (A, B);
```

REMEMBER!

- Lifetime & Scope
  - ↳ area / region of code where a variable is available / accessible
  - ↓
  - duration of time the computer allocates memory for it.

Using an execution stack, show that the function will work or not based on the fn. specification.

Explain why / why not.



A.R. of ex

A.R. of mc

exchange

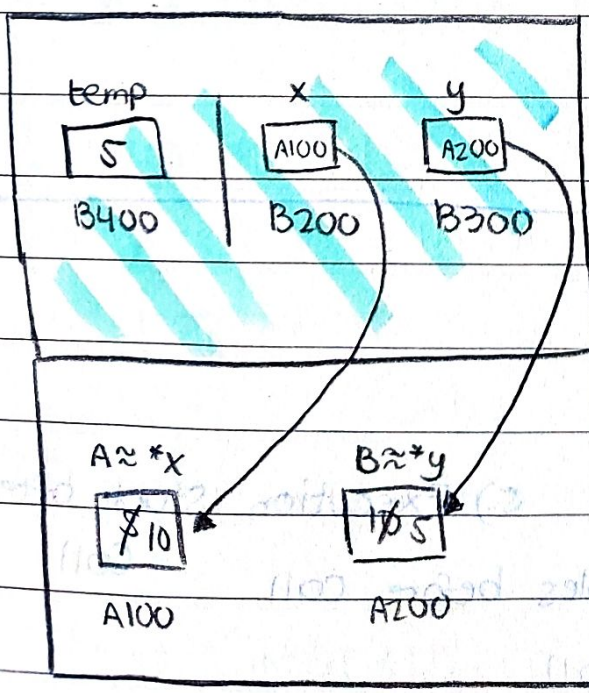
* No need to use a "trav" variable when traversing a list that was passed by copy.

BUT, it is good practice to use a trav variable in case data needs to be changed in the future.

No exchange happens bcs (A,B)'s address is not being passed. Only the value is.

exchange () has NO ACCESS to the variables in main ().

SOLUTION: give access (address)



A.R. of exchange ()

exits/ terminates

A.R. of main ()

A100 A200

exchange (&A, &B);

Date: / *SOLUTION

① void exchange (int *x, int *y) : Pass by Reference

② int A=5;
int B=10;
exchange (&A, &B);

③ execution stack ←

④ void exchange (int *x, int *y) {

int temp;
temp = *x;
*x = *y;
*y = temp;
}

(*) = dereference the variable
get the value of the
address its pointing to

---

*Problem Steps

1.) Function Header
2.) Declare & Initialite Variables before call
Appropriate function call
3.) Execution Stack
4.) Code
5.) Execution Stack after
Call

- LINKED LISTS -

Date: / /

* Data Structure Definition:

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

SIZE OF STRUCT NODE

* Problem Specifications

The function will insert an element at the 1st position in the given list.

(1.) void insert (LIST L , int elem)

(2.)
```
LIST A;          // assume list has 3 elements
int b = 5;
insert (A,B);
```

(3.) execution stack →

(4.)
```
void insert (LIST L, int elem
    LIST temp = malloc (size
    temp → data = elem;
    temp → link = L;
```

* Remember to check if dynamically allocated list is EMPTY or FAILED TO ALLOCATE before use.
* Also, typecast when you malloc.

- LINKED LISTS -

Date: / /

* Data Structure Definition:

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

* Problem Specifications

The function will insert an ... in the given list.

① void insert (LIST L, ... in)

② LIST A;
... ;
insert (A,B); // assume list has 3 elements

③ execution stack →

④ void insert (LIST L, int elem

```
LIST temp = malloc (size
temp → data = elem;
temp → link = L;
```

* calculating SIZE of struct node

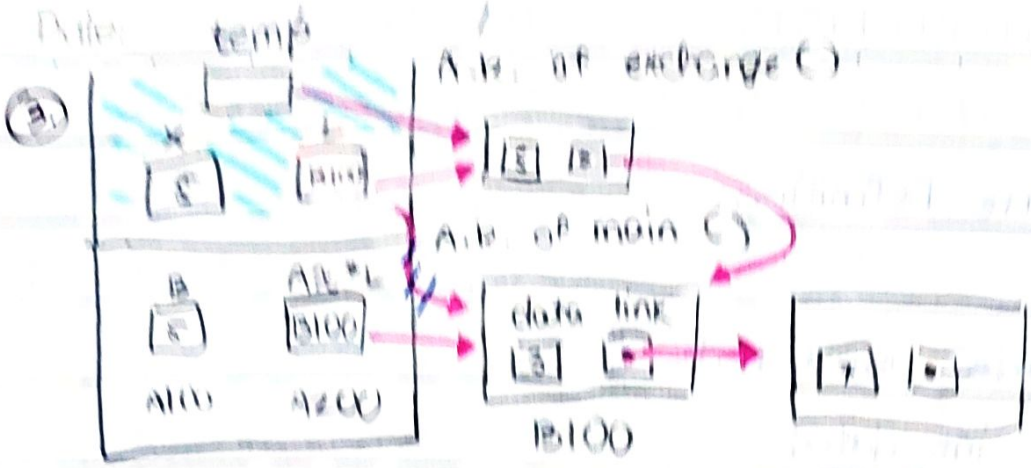
int = 4 bytes
pointer = 8 bytes

4+8 = 12 bytes (w/o padding)
= 16 bytes (w/ padding)

64-bit computers extracts data in multiples of 8.

* Remember to check if dynamically allocated list is EMPTY or FAILED TO ALLOCATE before use.

* Also, typecast when you malloc.



* List is OUTSIDE of stack, its in the HEAP because the list is DYNAMICALLY ALLOCATED.

The exchange WONT WORK because list was passed by copy. Can't change the address its pointing to

LIST L should point to new node temp.

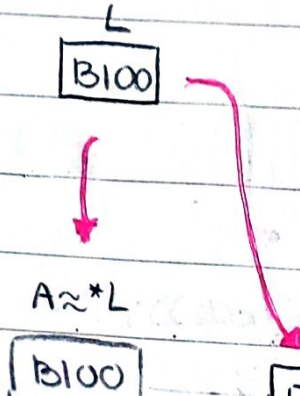
* SOLUTION - Insert First

* Test Case 1: 3 elements

a. void insertFirst (LIST *L, int x)

b. insertFirst (&A, B);

d.




```
void insertFirst (LIST *L, int x)
LIST temp = (LIST) malloc (sizeof (struct node
temp → data = x;
temp → link = *L;
*L = temp;
```

check if temp is empty!

Date: / /

(e.) Simulate the code.

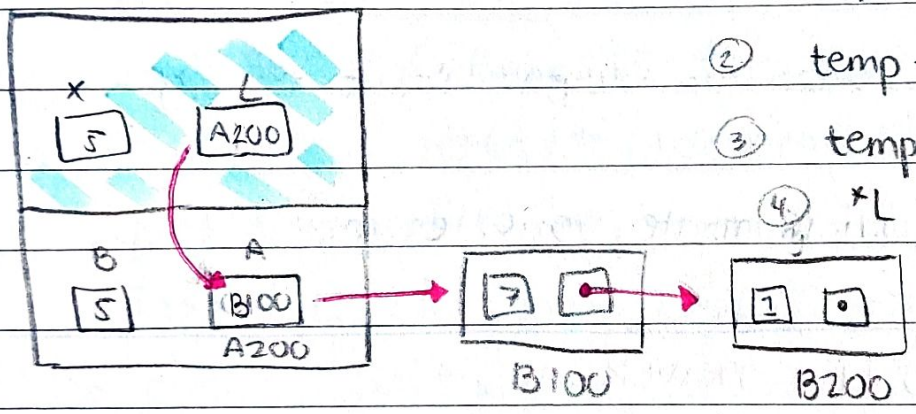
* Test case 1: 3 elements

2: List is empty

① Dynamically allocate new node

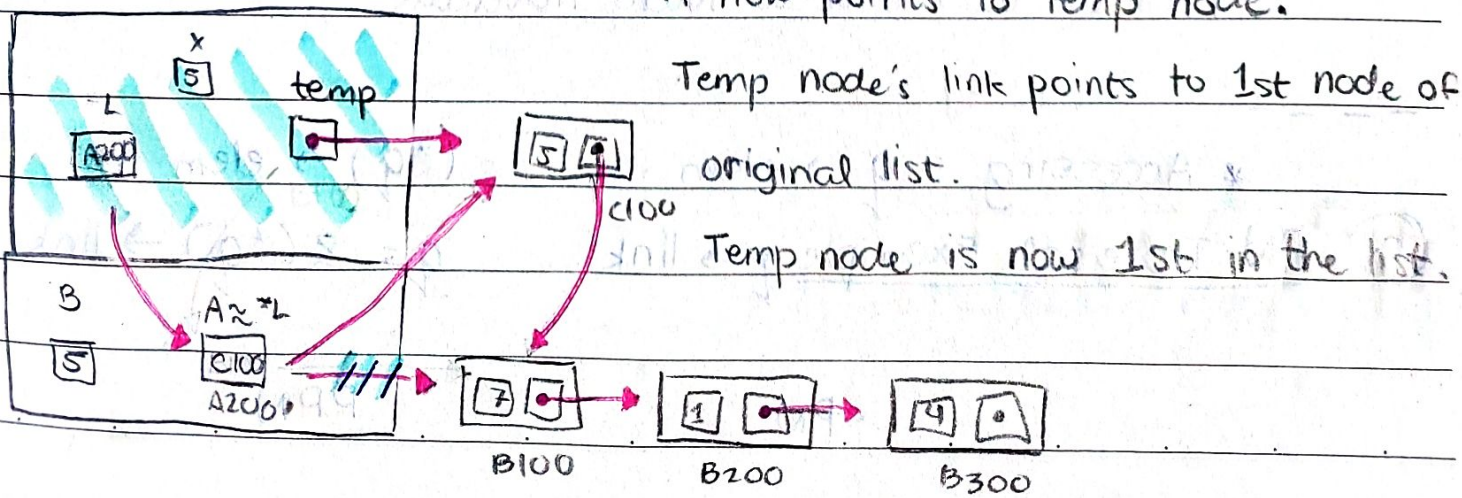
STEPS

BEFORE




```
void insertFirst(LIST *L, int x) {
① LIST temp = (LIST) malloc (sizeof(struct node
   if (temp != NULL) {
② temp → data = x;
③ temp → link = *L;
④ *L = temp;
}
```

AFTER



* A now points to temp node.

Temp node's link points to 1st node of original list.

Temp node is now 1st in the list.

Date: / /

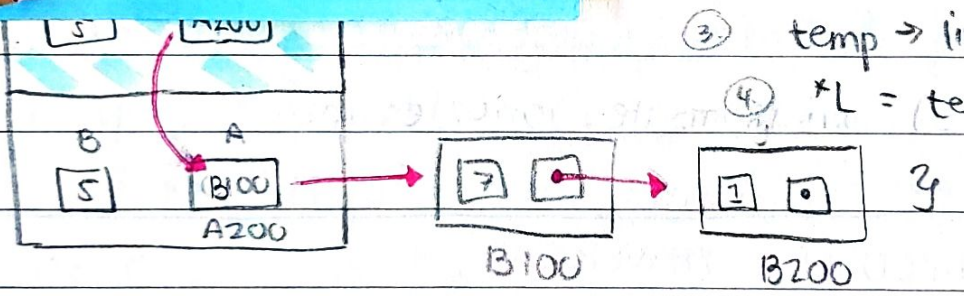
(e.) Simulate the code.

* Test case 1: 3 elements

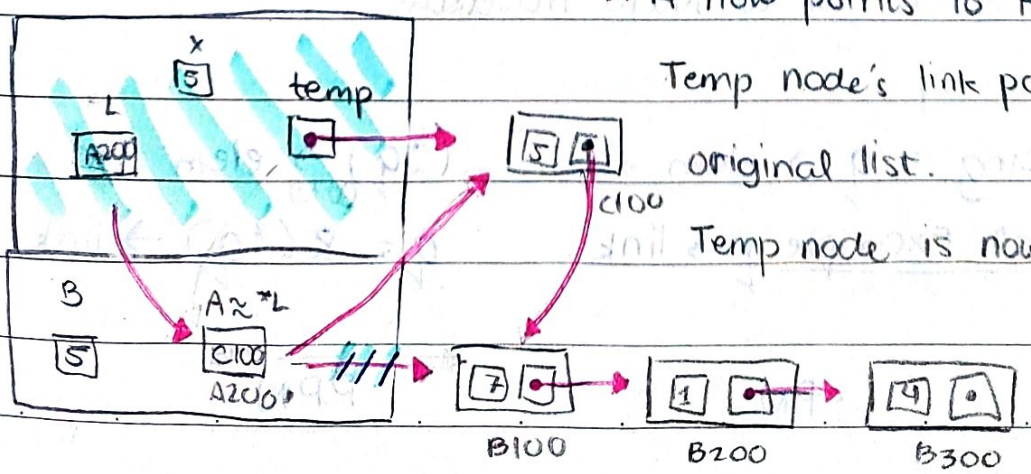
2: List is empty

1. Dynamically allocate new node (temp)
2. Assign data to temp.
3. Set temp's link to point to address held by *L.
4. Set *L to point to temp node.

```
void insertFirst(LIST *L, int x) {
① LIST temp = (LIST) malloc (sizeof(struct node));
  if (temp != NULL) {
② temp → data = x;
③ temp → link = *L;
④ *L = temp;
}
```



AFTER



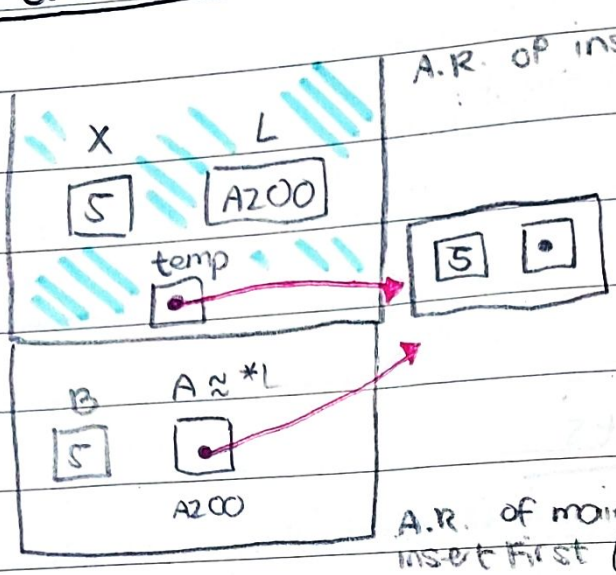
* A now points to temp node.

Temp node's link points to 1st node of original list.

Temp node is now 1st in the list.

Date: / /

* Test Case 2: List is empty.



* A will point to new node temp which is set by dereferencing *L in insertFirst().
insertFirst () will terminate, variables inside scope will no longer exist. Changes made are still kept.

- LINKED LIST TRAVERSALS -

Two types:

1.) Pointer to Node Traversal

2.) Pointer to Pointer to Nod

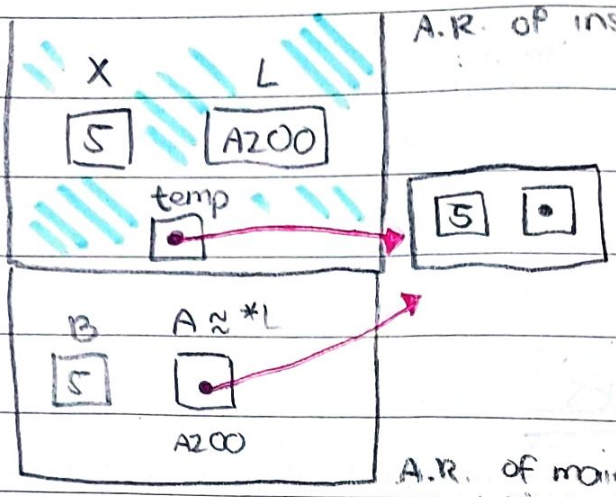
COMMON OPERATIONS

* Accessing : q → elem (*q) → elem
* Traversal : q = q → link q = &(*q) → link

PN PPN

Date: / /

*Test Case 2: List is empty.



* A will point to new node temp which is set by dereferencing *L in insertFirst().
insertFirst() will terminate, variables inside scope will no longer exist. Changes made are still kept.

- LINKED LIST TRAVERSALS -

Two types:

1.) Pointer to Node Traversal

2.) Pointer to Pointer to Nod

* Accessing : q → elem

* Traversal : q = q → li

PN Traversal :
display()
sort()

PPN Traversal :
insert()
delete()

Date: / /

① PN Traversal Example

```
typedef struct {
    int ID;
    char name[30];
    char course[8];
    int yearlevel;
} studtype;
```

```
typedef struct node {
    studtype stud;
    struct node *link;
} *List;
```

```
typedef enum {TRUE, FALSE} boolean;
```

(A.) Given the data structure dfn. Write the code of the fn. that will return the total # of 1st year students in the given list.

① int getList (List L)

- return type: int
  bcs its asking for total #
- list is given; only PASS BY COPY bcs list isnt being ALTERED/CHANGED.

② Variables & Function Call

```
List A;
getList (A);
```

CODING FUNCTIONS STEPS

Date: / /

① PN Traversal Example

```
typedef struct {
    int ID;
    char name[30];
    char course[8];
    int yearlevel;
} studtype;
```

```
typedef struct node {
    studtype stud;
    struct node *link;
} *List;
```

```
typedef enum {TRUE, FALSE} boolean;
```

(A.) Given the data structure dfn. Write the code of the fn. that will return the total # of 1st year students in the given list.

① int getList (List L)

- return type: int
  bcs its asking for total #
- list is given; only PASS BY COPY bcs list isnt being ALTERED/CHANGED.

② Variables & Function Call

```
List A;
getList (A);
```

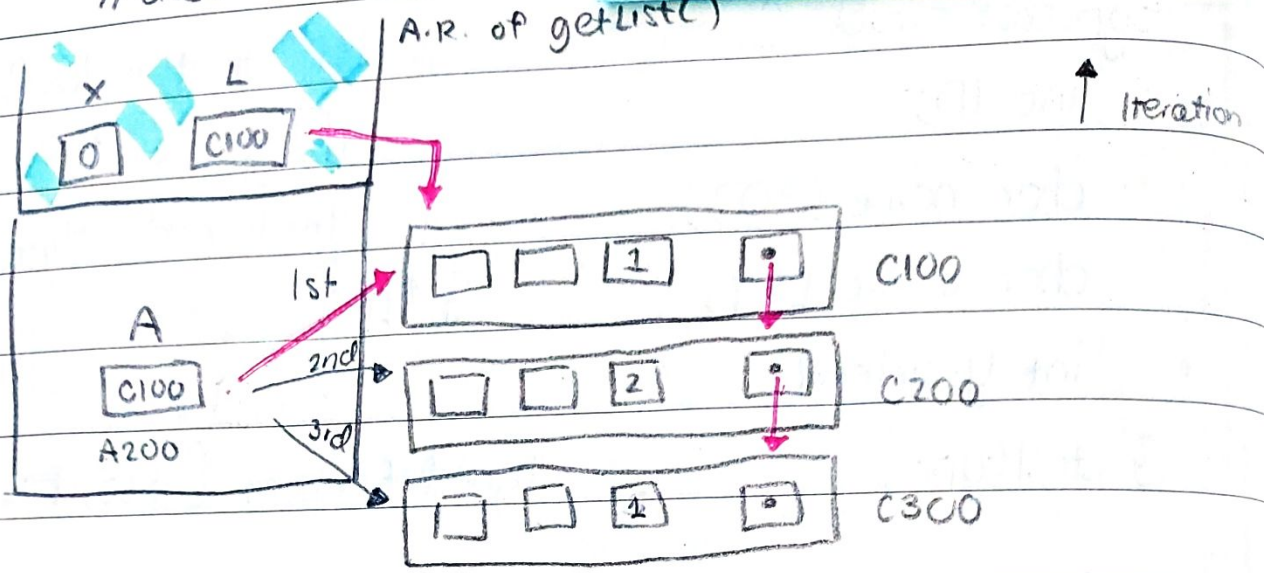
REMEMBER
1.) Function Header
2.) Variables & Fn. Call
3.) Execution Stack
4.) Code of the Function
5.) Test Cases!

AFTER 3 ITERATIONS

Date: / /

(3) Execution Stack

// assume that there ar



* L will be used to traverse thru the list as it is a COPY of A / points to the same node A does.

(4) Code of the Function

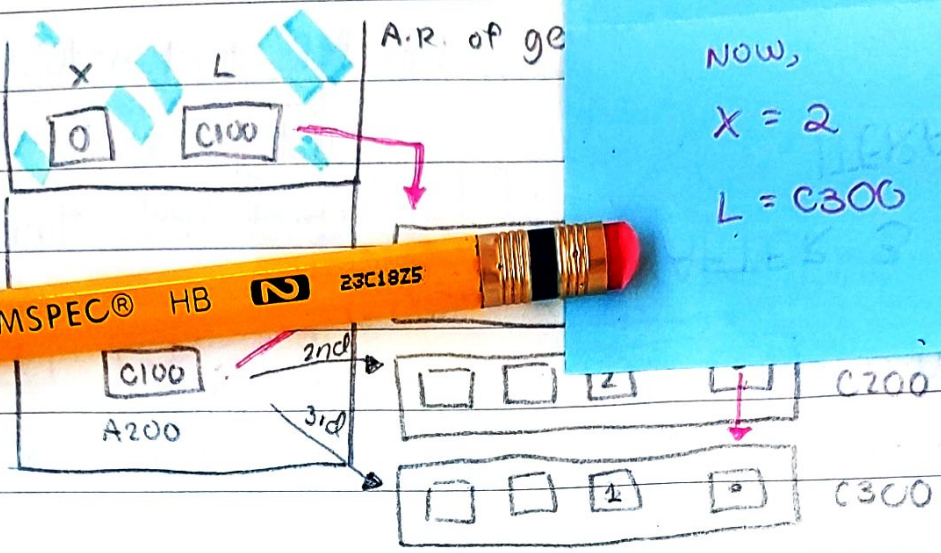
```
int getList (List L) {
    int x;
    for (x=0; L!=NULL; L=L->link) {
        if (L-> stud. yearlevel == 1) {
            x++;
        }
    }
    return x;
}
```

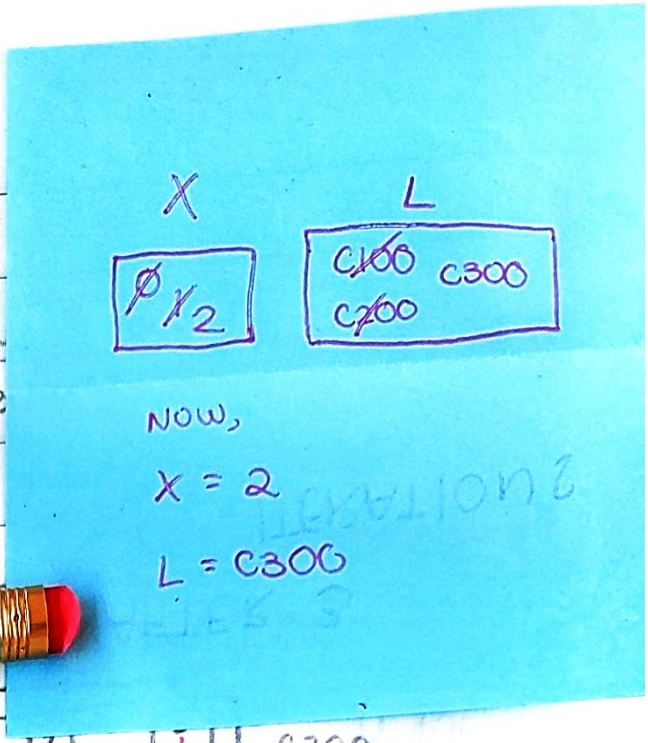
moves to next node after every iteration

Date: / /

③ Execution Stack

// assume that there ar





TPencil® by AMSPEC® HB 23C18Z5

* L will be used to traverse thru the list as it is a COPY of A / points to the same node A does.

④ Code of the Function

```
int getList (List L) {
    int x;
    for (x=0; L != NULL; L = L->link) {
        if (L-> stud. yearlevel == 1) {
            x++;
        }
    }
    return x;
}
```

moves to next node after every iteration

Date: / /

B. Write the code of the fn. that will return TRUE if the record bearing the given ID is in the given list; otherwise, return FALSE.

1. Function Header

*NOT CHECKED!

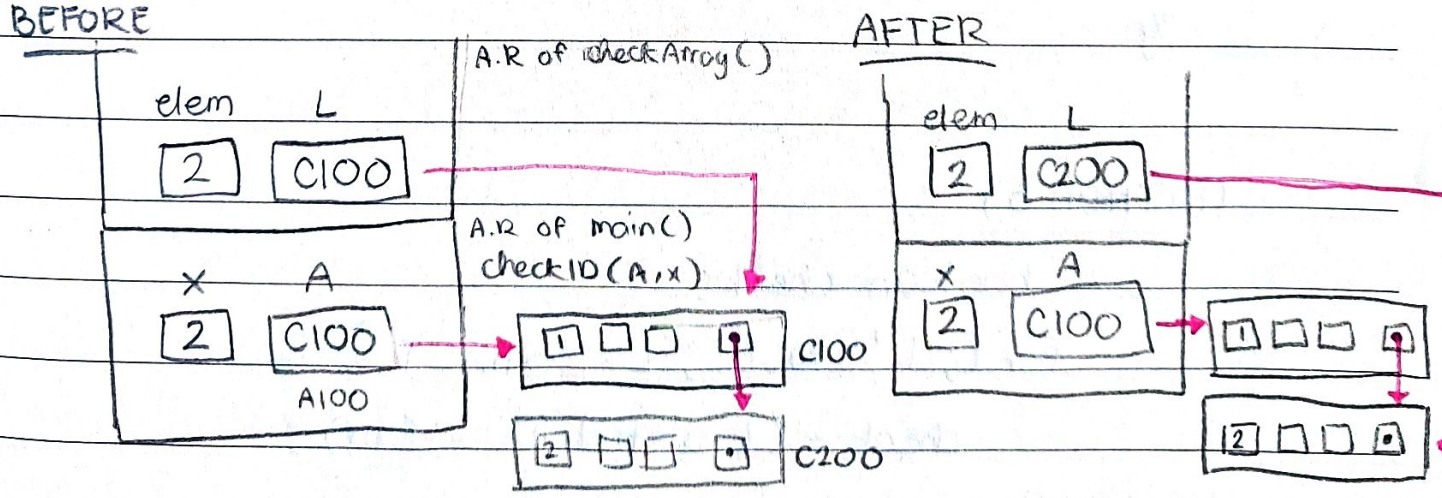
```
booleon checkID(List L, int elem)
```

2. Variables & Fn. Call

```
List A;
int x=2;
checkID=(A,x);
```

// only pass by copy since its just checking

3. Execution Stack



Date: / /

④ Code of the Function

(METHOD 1)

```
boolean checkID (List L, int elem) {
        while (L != NULL) {
            (L→stud.ID == elem)? return TRUE: L=L→
        }
    return FALSE;
}
```

(METHOD 2)

```
for (; L != NULL; L=L→link) {
    if (L→stud.ID == elem) {
        return TRUE;
    }
return FALSE;
}
```

(METHOD 4)

(METHOD 3)

```
booleon check;
for (; L != NULL; L=L→link) {
    check = (L→stud.ID == elem);
}
return check;
```

```
boolean checkID (LIST L, int elem) {
    while (L != NULL) {
        (L→stud.ID == elem) ? return TRUE : L=L→link;
    }
    return FALSE;
}
```

(METHOD 2)

```
for (; L != NULL; L=L→link) {
    if (L→stud.ID == elem) {
        return TRUE;
    }
}
return FALSE;
}
```

(METHOD 3)

```
boolean check;
for (; L != NULL;
    check = (L→
}
return check;
```

```
int i;
for(i; L != NULL && L→stud.ID != elem; L=L→link);
return (L != NULL) ? TRUE : FALSE;
```

* for loop (2 conditions)
  - stop when list is at end
  - stop when element is fo
* return
  - if list not null, TRUE
  - if empty, FALSE

Date: / /

- ARRAY TRAVERSALS -

(A.)

* Problem Specifications

Given the array of integers, the size of the array & an element x, the function will return TRUE if x is in the array, otherwise FALSE.

```
typedef enum {TRUE, FALSE} boolean;
```

(1.) Function Header

```
boolean checkArray(int A[], int size, int x)
```

(2.) Variables & Fn. Call

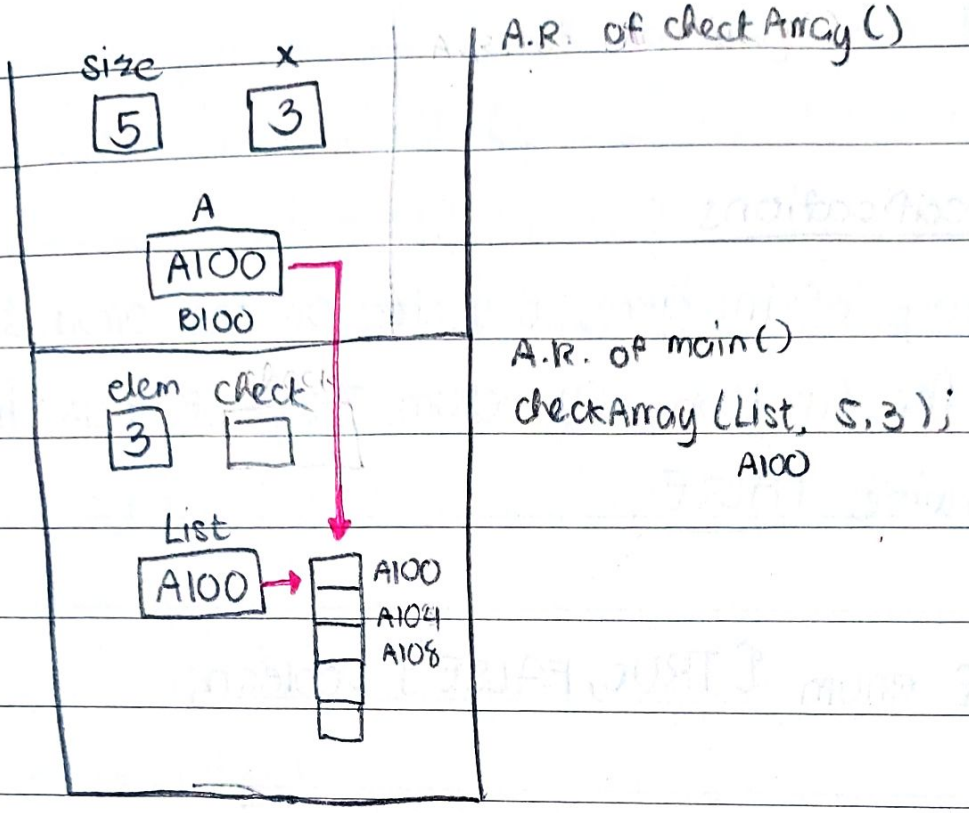
```
int List[5];    List[5] = {1,2,3};
int elem = 3;
boolean check;
check = checkArray(List, 5, 3);
```

* The name of the array is the ADDRESS of the 1st component. its also a POINTER.

* Make a catcher variable that the fn. will return to.

Date: / /

(3.) Execution stack



(4.) Code

(METHOD 1)

```
boolean checkArray(int A[], int size, int x) {
    boolean check = TRUE;
    int n;
for(n=0; n<size && x!= A[n]; n++) {}
    if (n == size) {
        check = FALSE;
    }
    return check;
}
```

Date: / /

(METHOD 2) CONCISE

```
boolean checkArray (int A[], int size, int x) {
    int n;
    for (n=0; n<size && A[n] != x; n++) {}
[   return (n<size)? TRUE : FALSE; ]
}
```

↑ combine the if statement & return statement TO BE CONCISE

or...

```
return check = (n==size)? FALSE :TRUE;
return (n==size)? FALSE: TRUE;
```

(B.)

* Problem Specifications

Function inputArray() allows user to input from the keyboard the total # of integers N, to be stored in the newly created array. Put N at index 0 of the new array. The values from index 1 to N will also be inputted from the keyboard. In addition, the newly created array will be returned to the calling function.

① Fn. Header

```
int* inputArray ()
```

* Use * when returning an ARRAY from a function.
* Because its a POINTER that was DYNAMICALLY ALLOCATED so it doesnt expire when the function terminates.

② Code

```
int * inputArray () {                    N = size
    int N;
    int ctr;
① N = scanf ("%d", &N);

② int* arr = (int*) malloc(sizeof(int)*(N+1));

    if (arr != NULL) {        → check if allocation was successful
③   arr[0] = N;
④   for (ctr = 1 ; ctr < N; ctr ++) {
        scanf ("%d", &arr[ctr]);
    }
⑤   return arr;
}
}
```

SUBTASKS

① Input valve of N
② allocate dynamically (malloc / calloc)
③ Initialize index 0
④ Input N integers
⑤ Return the array

Date: / /

- DISPLAYING ARRAYS -

* Problem Specifications

Function displayArray() will accept as parameters, the array of integers & size of the array. It will display the elements of the given array.

* Program Structure

A. Include files (#include <stdlib.h>)

B. Function Prototypes

inputArray()

displayArray()

isMember()

C. Main() Function

- Declare variables
- Printf ("Task 1")

// call inputArray() & displayArray()

- Printf ("Task 2")

// call isMember() 2 times

① Element is found

② Element is not found

D. Definition of the Functions listed in the Function Prototypes

Date: / /

-MALLOC vs CALLOC-

- **Malloc**

(void *) malloc (size_t size)

↑ data type ↑ var

- **Calloc**

(void *) calloc (size_t noOFItems, size_t elemsize)

* Lifetime of dynamically allocated space

starts at : malloc / calloc

ends at : restart / free ()

Library : #include <stdlib.h>

- Array List Traversal -

C code

* displayArray ()
* inputArray ()
* isMember ()

Library : #include <stdlib.h>

```
 1: // displaying array lists
 2:
 3: #include <stdio.h>
 4: #include <stdlib.h>
 5:
 6: // data type definitions
 7: typedef enum {TRUE, FALSE} boolean;
 8:
 9:
10: // function prototypes
11: void displayArray(int A[], int size);
12: int* inputArray();
13: boolean isMember(int A[], int size, int x);
14:
15: void main () {
16:
17:     int* List;
18:     int elem = 3;
19:     boolean check;
20:
21:     printf("\n**** Task 1 **** \n");
22:     List = inputArray();
23:     displayArray(List+1, List[0]);
24:
25:     printf("\n**** Task 2 **** \n");
26:     check = isMember(List+1,List[0],3);
27:     (check == TRUE) ? printf("Element is found! \n") : printf("Element is not found
28:
29:     check = isMember(List+1,List[0],7);
30:     (check == TRUE) ? printf("Element is found! \n") : printf("Element is not found
31:
32: }
33:
34: void displayArray(int A[], int size) {
35:     int i;
36:
37:     printf("\nElements in array list: \n");
38:
39:     for (i = 0; i < size; i++) {
40:         printf("%d\n", A[i]);
41:     }
42: }
43: int* inputArray() {
44:     int N, ctr;
45:
46:     printf("\nInput size of array: ");
47:     scanf("%d", &N);
48:
49:     int* arr = (int*) malloc (sizeof(int)*(N+1));
50:
```

pass array as pointer pointing to starting addres. A[1]

```
51:     if (arr != NULL) {
52:         arr[0] = N;
53:
54:         printf("\nInput numbers:\n");
55:         for (ctr = 1; ctr <= N; ctr++) {
56:             scanf("%d", &arr[ctr]);
57:         }
58:     }
59:     return arr;
60: }
61:
62: boolean isMember(int A[], int size, int x) {
63:     int n;
64:
65:     for (n = 0; n < size && A[n]!= x; n++) {}
66:     return (n < size) ? TRUE : FALSE;
67: }
68:
69:
70:
```

Date: / /

-ABSTRACT DATA TYPES (ADT) -

- a mathematical model together with a set of operations defined on the model

- object-oriented

Example:

Model = ADT Stack

Operations = pop, push

* ADT List

- sequence of 0 or more elements (can be: scalar aggregate)

- cont be on ADT List WITHOUT operations

- Operations
  1.) Initialize
  2.) Insert
  3.) Delete
  4.) isMember

* ADT List Representation (in memory)

a.) Array Implementation
b.) Linked List
c.) Cursor-based

Date: / /

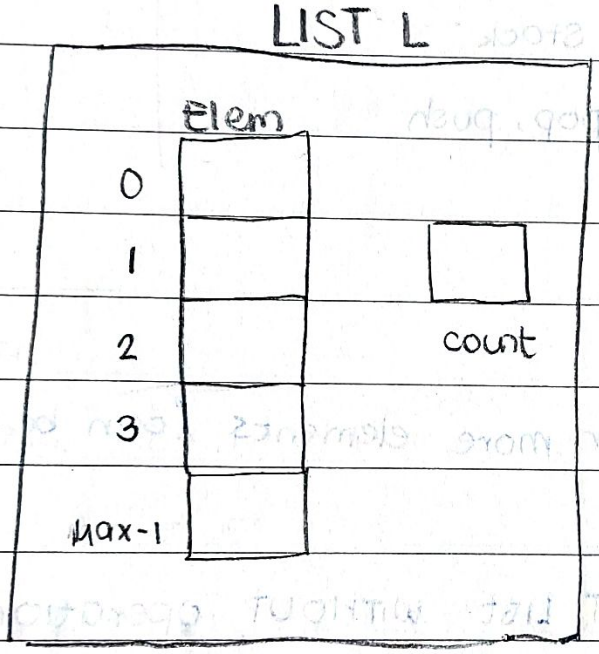
-ARRAY IMPLEMENTATION-

4 VERSIONS

* VERSION 1

List is a structure containing an array & count.

a.) Version A: Scalor Element



① Definition

```
#define MAX 10

typedef struct node {
    int Elem [MAX];
    int count;
} LIST;
```

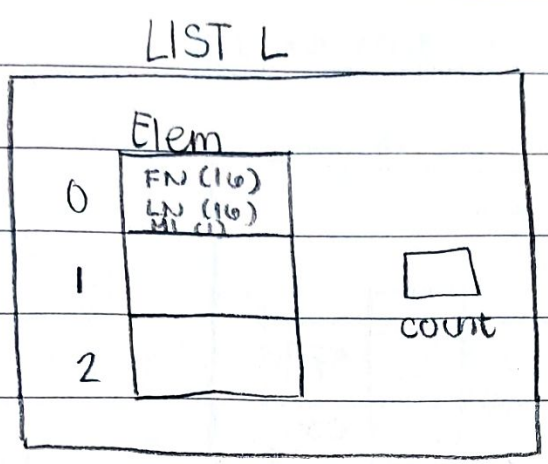
② Initialize = initList ()

```
void initList (LIST *A) {
    A → count = 0;
}
```

* we pass by address but no need to return anything : void.
* we access the list using count so just initialize this.

Date: / /

b.) Version B : Aggregate Element



* Size of L = 44

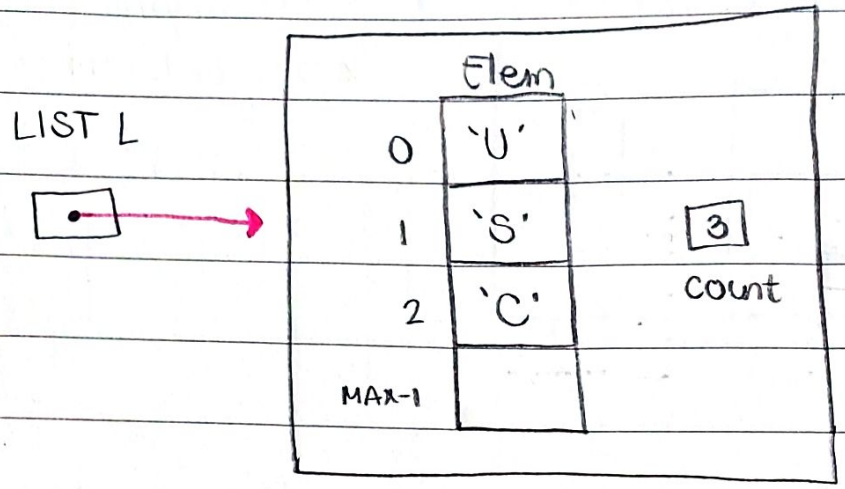
① Definition

```
#define MAX 10

typedef struct node {
    char FN[16];
    char LN[16];
    char MI;
} studNOme;

typedef struct node2 {
    studNOme Elem[MAX];
    int count;
} LIST;
```

Date: / /

*VERSION 2

List is a pointer to a structure containing an array and count.



initList ()
Execution Stack

① Definition

```
#define MAX 10
typedef struct node {
    char Elem [MAX];
    int count;
} *LIST;

LIST A; // 8 bytes allocated
```

BEFORE the name

* No need for structure name
* ~~BUT BETTER! for malloc/calloc~~ bcs you'll have to manually calculate the data type.

② Initialize = initList ()

- Prepare list to be used.
- Update count.

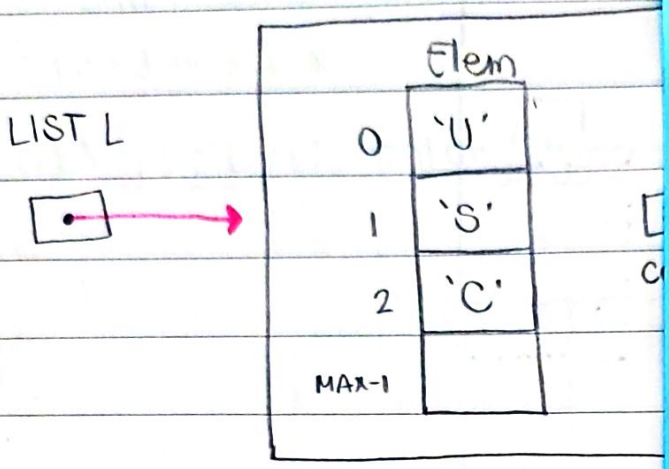
```
void initList ( LIST *L) {
    *L = (LIST) malloc (sizeof( struct node));
    if (L != NULL) {
        (*L) -> count = 0;
    }
}
```

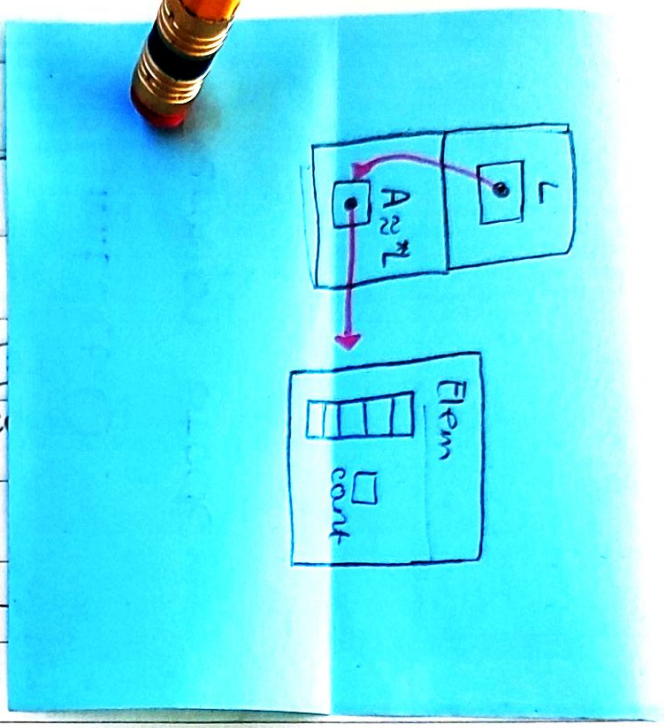
* Enclose in () to ensure proper precedence.

Date: / /

*VERSION 2

List is a pointer to a structure containing an array and count.





① Definition

```
#define MAX 10
typedef struct node {
    char Elem[MAX];
    int count;
} *LIST;

LIST A;  // 8 bytes allocated
```

BEFORE the name

* No need for structure name
* ~~BUT BETTER! for malloc/calloc~~ bcs you'll have to manually calculate the data type.

② Initialize = initList()

- Prepare list to be used.
- Update count.

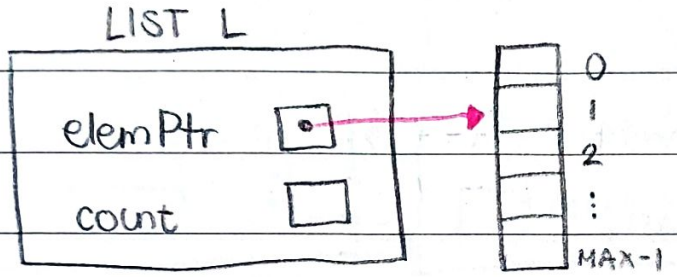
```
void initList(LIST *L) {
    *L = (LIST) malloc(sizeof(struct node));
    if (L != NULL) {
        (*L) -> count = 0;
    }
}
```

* Enclose in () to ensure proper precedence.

Date: / /

* VERSION 3

List is a structue containing a variable count & a pointer to the 1st element of a dynamically allocated array.



(1) Definition

```
#define MAX 10
typedef struct node {
    int* elemPtr;
    int count;
} LIST;

LIST L;
```

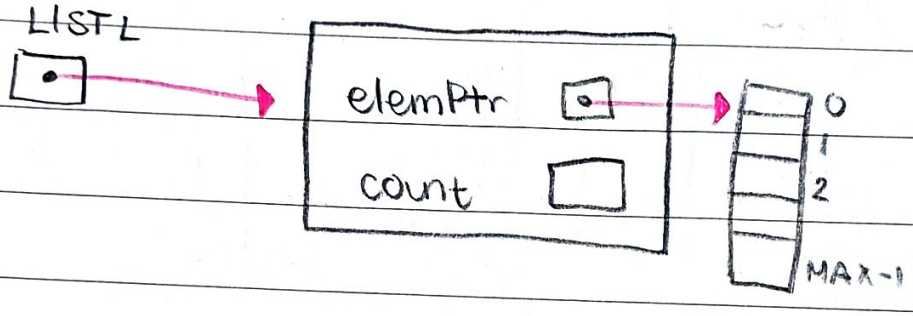
(2) Initialize

```
void initList (LIST *L) {
    (*L)->elemPtr = (int*) malloc
                    (sizeof(int) * (MAX));
    if (L != NULL) {
        L->count = 0;
    }
}
```

Date: / /

* VERSION 4

List is a pointer to a structure containing variable count and a pointer to the 1st element of a dynamically allocated array.



// double malloc!

① Definition

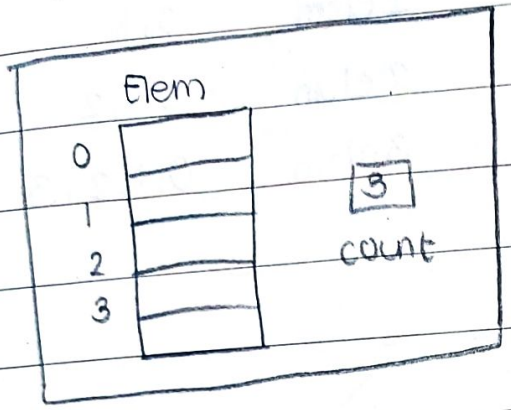
```
#define MAX 10
typedef struct node {
    int* elemPtr;
    int count;
} *LIST;

LIST L;
```

② Initialize

```
void initList(LIST *L) {
    *L = (LIST) malloc(sizeof(struct node));
    if (L != NULL) {
        (*L)->elemPtr = (int*) sizeof(int) MAX);
        (*L)->count = 0;
    }
}
```

* Array Implementation

- elements are stored in contiguous cells of the array
- NO GAP!



Example:

insert 'C' at index 0 CUS

insert 'C' at index 1 UCS

insert 'C' at index 2 USC

* insertPosition() Definition

```
#define MAX 10
typedef struct {
  char Elem [MAX];
  int count;
} LIST;

typedef int Position;
```

* Function Specifications

Given the list, element & position, write the code of the fn. insertPosition(), IF there is space on the list & position is valid.

// assume 1st position is 0!

Date: / /

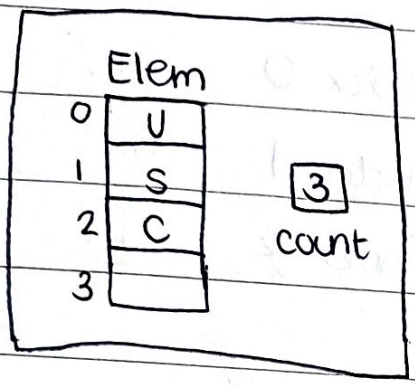
⊙ To help visualize:

Elem [3] = Elem [2]

Elem [2] = Elem [1]

Elem [1] = Elem [0]

count	Valid Position
0	0
1 elem	0,1
2 elem	0,1,2
3 elem	0,1,2,3



CODE SIMULATIONS

* Code // element shift starts at the BOTTOM!

```
void insertPosition (LIST *L, char x, Position pos) {
  ① if ((*L)→ count != MAX && pos ≤ (*L)→ cant) {
        int y;
  ② for( y=(*L)→ count; y > pos ; y-- ){
        (*L)→ Elem [y] = (*L)→ Elem [y-1];
    }
  ③ (*L)→ Elem [pos-1] = x
  ④ (*L)→ count = (*L)→ count
    }
}
```

STEPS

1.) check if there is space & position is valid.

2.) if yes, make space (SHIFT)

3.) insert new element

4.) update count

index [0] = pos 1

Date: / /

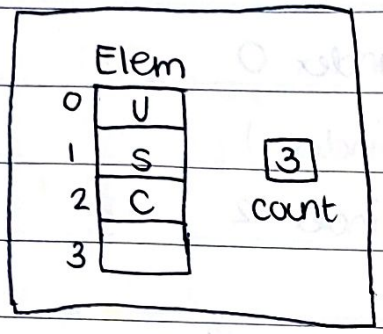
⊙ To help visualize:

Elem [3] = Elem [2]

Elem [2] = Elem [1]

Elem [1] = Elem [0]

count	Valid Position
0	0
1 elem	0,1
2 elem	0,1,2
3 elem	0,1,2,3



[pos = 1] index = 0 elem = "H"

(y = 3; y > 1 ~~&& y >= 0~~; y--)

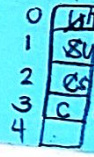
y=3 Elem [3] = Elem [2]

y=2 Elem [2] = Elem [1]

y=1 Elem [1] = Elem [0]

y=0 X Elem [0] = Elem [0]

y Elem [0] = "H"



* Code // element shift stc

```
void insertPosition (LIST *
① if ((*L)→ count != MAX
        int y;
② for( y=(*L)→ count; y > pos ; ){
        (*L)→ Elem [y] = (*L)→ E    [y-1];
    }
③ (*L)→ Elem [pos-1] = x
④ (*L)→ count = (*L)→ count
    }
}
```

S e

1.) eck if there is space & position is valid.

) if yes, make space (SHIFT)

3.) insert new element

4.) update count

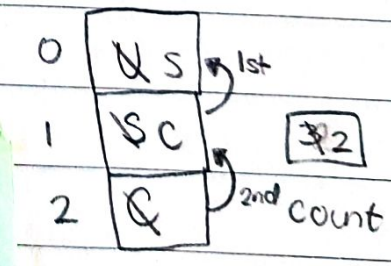
index [0] = pos 1

Date: / /

* delete() Problem Specifications

Given the list & element, write the code of the fn. that will delete the element from the list.

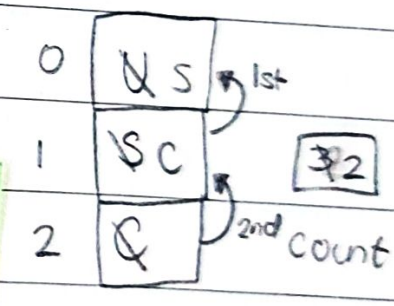
```
void delete (LIST *L, char x) {
    int i,y;
    *element shift starts at the TOP! ↑ )
     opposite of insert()
① if ((*L)->count != 0) {
② if ((*L)->Elem[*L->count-1] != x) {
③ for (y=0; y<(*L)->count && (*L)->Elem[y] != x; y++);
④ if (y != (*L)->count) {
⑤ for (i=y; i<(*L)->count-1; i++) {
        (*L)->Elem[i] = (*L)->Elem[i+1];
    }
  } else {
      printf("Element not found!");
  }
⑥ (*L)->count--;
 }
}
```



SIMULATION

insert ()

```
1  if (*L) -> count != 0) {
2    if ((*L) -> Elem [*L -> count-1] != x) {
3      for (y=0; y < (*L) -> count && (*L) -> Elem [y]
4        if (y != (*L) -> count) {
5          for (i=y; i < (*L) -> count -1 ; i++ ) {
               (*L) -> Elem [i] = (*L) -> Elem [i+1];
           }
         } else {
           printf (" Element not found!");
         }
6      (*L) -> count --;
       }
     }
```



x = 'S' = index = 1 count = 3

if(3 != 0)? YES

if (Elem[2] != 'S') YES

for(y=0; y < 3 && Elem[y] != S) xtt

1st: y=0, 0<3 U != S

2nd: y=1, 1<3 S != S FOUND!

if(y=1 != 3) YES

for (i=1; i<2; i++)

Elem[1] = Elem [2]

i=2, 2<2 ? NO

count --;

Date: / / PROCESS

1. Check if list is empty. > If yes, exit.
   > If no, proceed.
2. Check if the given element is NOT in the LAST INDEX of the list. > If not, proceed.
   > If yes, exit and decrement count.
3. Traverse thru the list & look for the element to delete.
   > If element not found (in each iteration), keep going.
   > If found OR counter has reached *L→count, exit.
4. Check if y != count. > If yes, proceed.
   > If no, element DNE (traversed thru the whole list and did not find elem).
5. From that position (when elem is found), shift the elements UPWARDS.
6. Update count.

Date: / /

*ADT List

1.) Array Implementation

↳ 4 versions

↳ characteristics:

1.) physical order of the elements IS THE SAME as the logical order

2.) space is finite (limited)

3.) elements sorted in contigouous cells of the array

↳ operations:

> insertion: shift DOWN

> deletion: shift UP

↳ runtime: O^N

O^1 (at last position)

* Why the need to pass the SIZE of an array to a fn. ?

what is passed is ONLY THE POINTER to the 1st address of the array.

The SIZE indicates how far we access from the 1st address.

* What about for strings?

No need to pass the size bcs a string has a '\0' that terminates / indicates the end of an array.

Date: / /

2.) Linked List Implementation

a) Singly-Linked List - one-directional

b) Doubly-Linked List - bidirectional

> Circular Doubly Linked

↳ does NOT require shifting

↳ physical order of elements IS NOT THE SAME as the logical order

* When to use ADT List & Linked List?

① When # of elements is already known beforehand

↳ ADT List

② If you only insertLast()

↳ ADT List bcs O^1

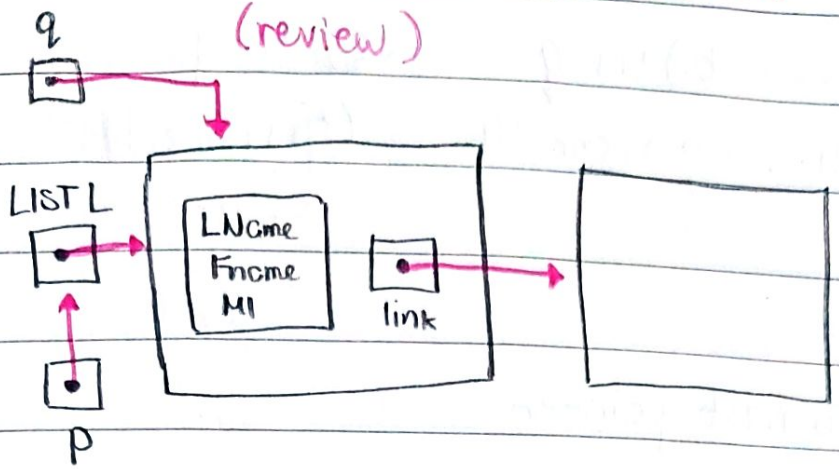
③ A lot of insertion & deletion

↳ Linked List bcs needs a lot of shifting.
ADT = O^N
LL = O^1

Date: / /

-TRAVERSAL & ACCESSING-

(review)



- Definition : LIST

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

```
typedef struct node {
    studName stud;
    struct node *link;
} *LIST, nodetype;
```

△ Aliases

struct node ≈ nodetype

nodetype* ≈ LIST

struct node * ≈ LIST

① Declare p & q.

* *p will exist when p has something to point to.

```
LIST *p;
p = &L;
```
PPN

```
LIST *q;
q = L;
```
PN

Date: / /

(2.) Access MI.

a.) via L

L→name.MI

b.) via q

q→name.MI

c.) via p

(*p) name.MI

↑ dereference

remember: *p ≈ L

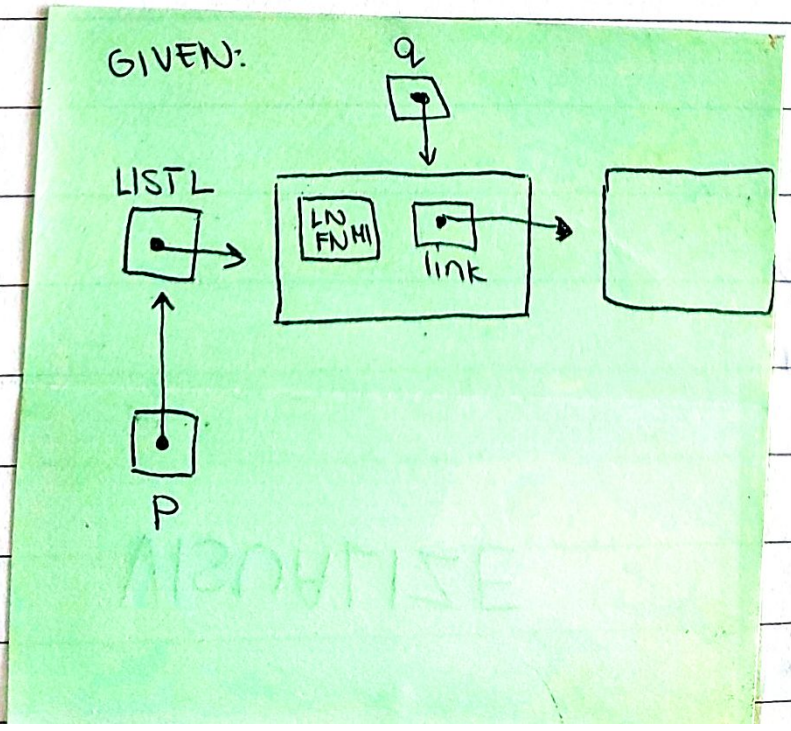
(3.) Let p move to next pointer.

p = &(*p)→link

↑ get address of next node by accessing link in the current node

(4.) Let q point to next node.

q = q→link;



Date: / /

* Traversal

- Array: Low-Index (0) to High-Index
  High-Index to Low-Index

- Linked List: 1st Node to the last node

* Loop Conditions

> One Condition

If the call HAS to end up at the last node.
Basically, needs to TRAVERSE THRU ALL nodes.

Ex: displayList()

> Two conditions (&&)

When you DON'T NEED TO reach the end of the list
i.e. when you're looking for an element, when element is found, just exit the loop.

Ex: findElement()

Date: / /

* Problem Specifications

Write the code for the fn. that will return the # of students bearing the given last name in the given list.

```
int getStudent (LIST A, char name[]) {
    int i;
for (i=0; A!=NULL; A= A→link) {
    if (strcmp (A→name.LName, name) == 0) {
        itt;
    }
}
return i;
}
```

// PN Traversal