 Date: / /

//insertLast (PPN) - Linked List (one condition)

- Definition

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

```
typedef struct node {
  name stud;
  struct node *link;
} *LIST;
```

- Code

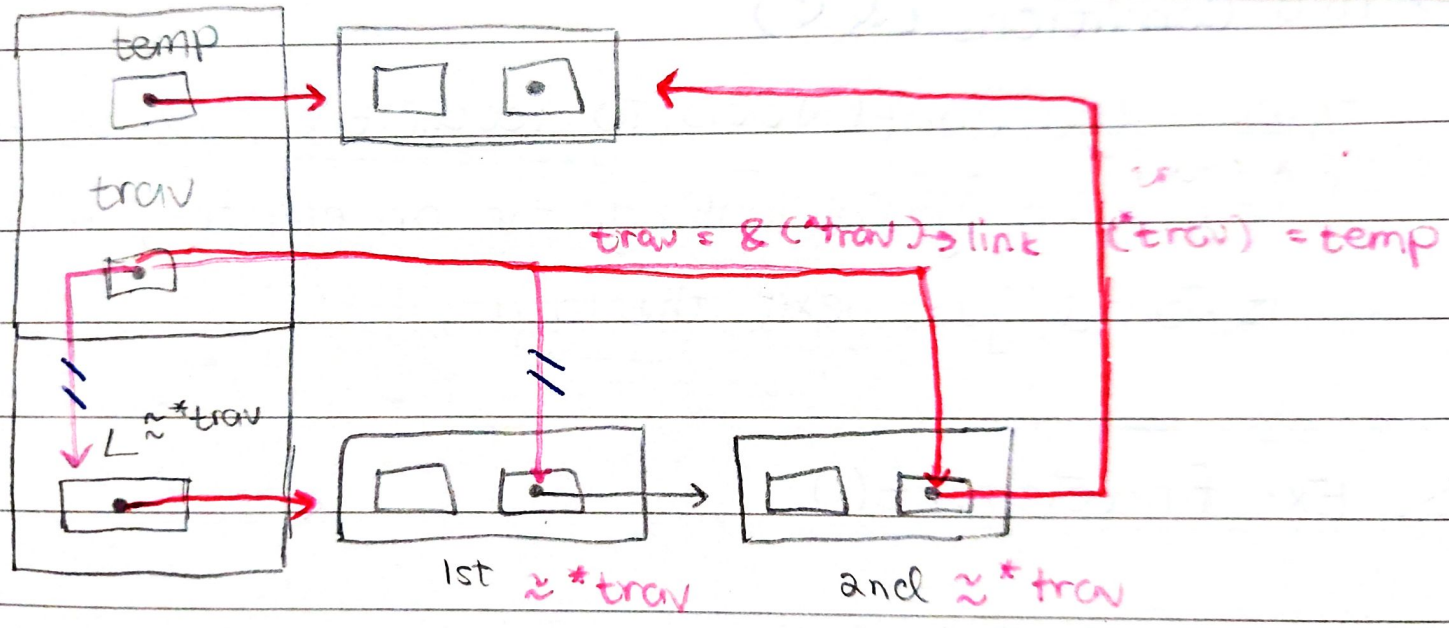
```
void insertLast (LIST *L, name newStud) {
① LIST *trav;
    trav = L;
```

bcs trav is a PPN!

```
② LIST temp = (LIST) malloc (size of (struct node));
  if (temp != NULL) {
    strcpy (temp → stud. LName = newStud. LName);
    strcpy (temp → stud. FName = newStud. FName);
    temp → stud. MI = newStud. MI;
    temp → link = NULL;
```

* only one condition bcs it has to go thru the WHOLE list.

```
③ while (trav != NULL) {
      trav = &(*trav) → link;
④ } (*trav) = temp;
}
```


temp
trav
trav = &(*trav)->link
(*trav) = temp
*trav
1st *trav
2nd *trav

① Initialize trav variable.

- will get address pointed to by L.

L ≈ *trav

points to 1st node.

- will traverse thru whole list to get to the end.

② Initialize temp node to act as new node.

- initialize values to hold newStud

③ Traverse thru whole list until the end.

4. Once trav gets to the end, (*trav) = temp to make it point to the new node.

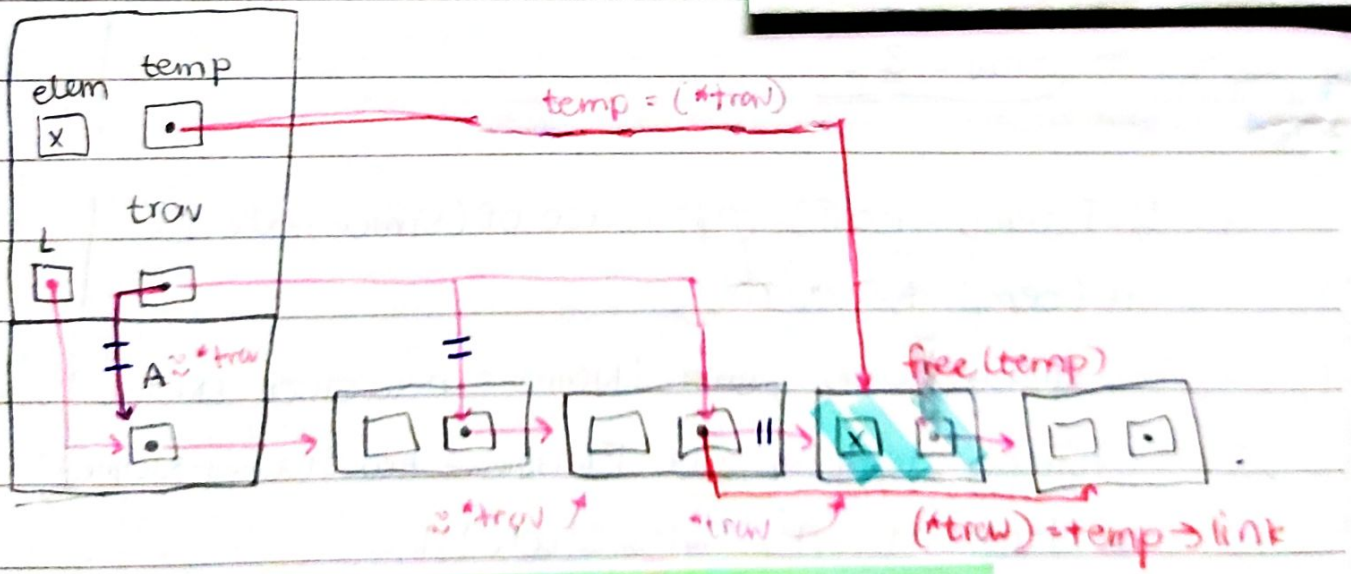
Date: / /

// delete() (PPN) - Linked List (2 conditions)

- Code

```
void delete (LIST *L, char elem) {
① LIST *trav, temp;

② for (trav = L; (*trav) -> ~~link~~ != NULL && (*trav) -> stud.MI ==
                                    elem, (trav) = &(*trav) -> link)
③   temp = (*trav);
④   (*trav) = temp -> link;
⑤   free (temp);
}
```

Iteration



① Initialize trav variable.
- will traverse thru list using the -> link field

STEPS

(1.) Initialize trav variable.

- will traverse thru list using the
  → link field

Initialize the temp variable.

- will point to the node to be deleted

(2.) Traverse thru list until a.) until reaches the end
b.) finds the elem to delete

(3.) Make temp variable point to node to be deleted.

4. Make *trav (or the →link field of the node before the deleted node) POINT to the next node using temp's → link field.

5. Delete temp.

Date: / /

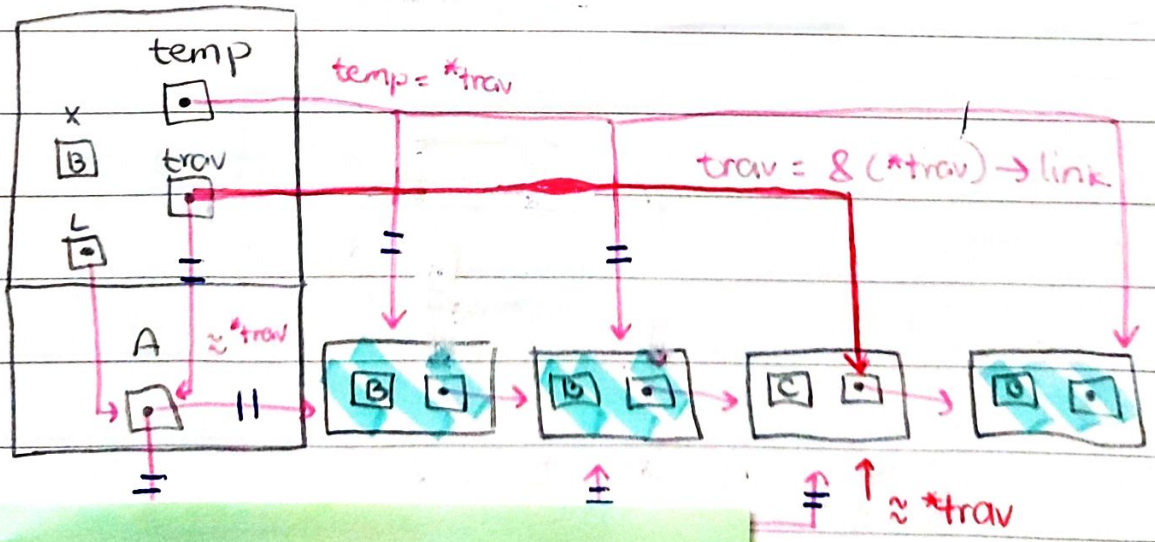
//delete All Occurences () (PPN) - Linked List (2 conditions)

```
void deleteAllOccurences (LIST *L, char x) {
    LIST *trav, temp;

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

* only ONE condition bcs we need to check each element & delete accdgly. prior to moving to next node.

*checks each element. if elem found, delete, if not, skip to next.

//delete 'B'

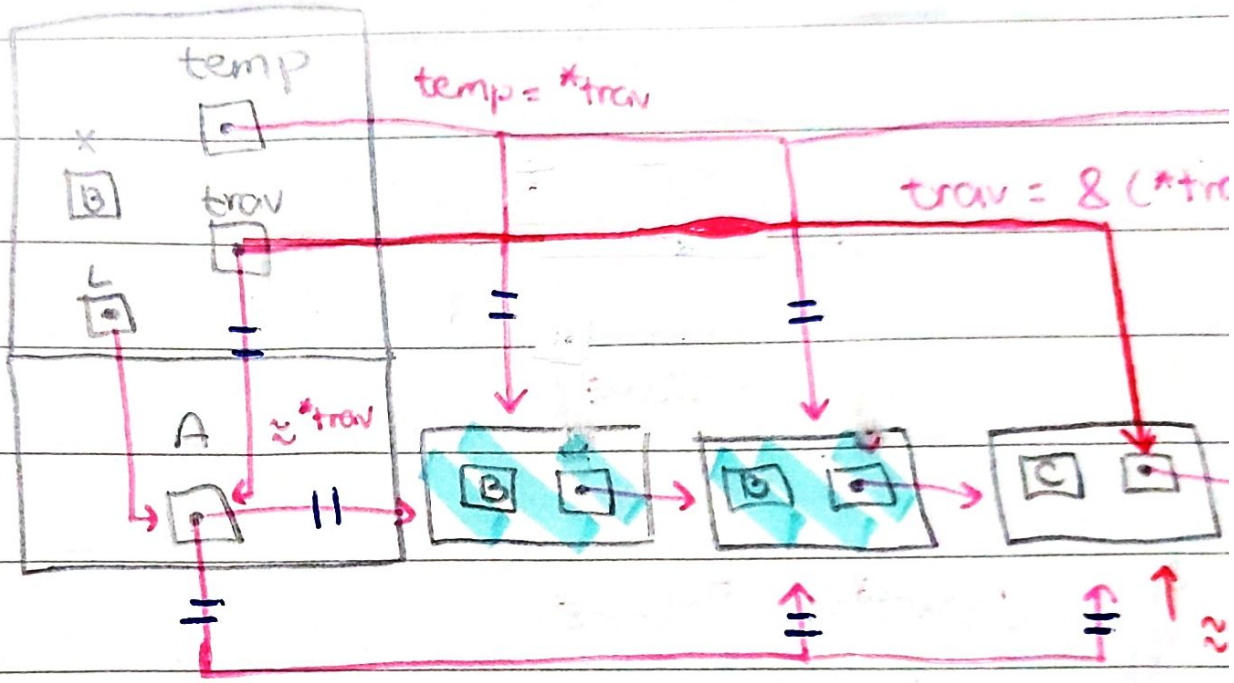


STEPS

*trav is now NULL. or 3rd's -> link field is NULL.

found, dele
next.

```
        free (temp);
    } else
        trav = &(*trav)->link;
    }
}
                    //delete 'B'
}
```



1. Initialize *trav & temp.
2. Traverse thru loop, checking each element w/ only one condition. And condition is INSIDE for loop.
   a.) if elem is found, shift.
   b.) if not, skip to next node.
3. After shifting, delete.

Date: / /

//deleteFirstTwo () (PPN) - (2 conditions)

```
void deleteFirstTwo (LIST *L, char x) {
    LIST *trav, temp;          // stops deleting after the 2nd
    int ctr;                   node.

for (trav=L, ctr=0; (*trav)!=NULL && ctr <= 2) {
    if ((*trav)!=data == x) {
        temp = *trav;
        *trav = temp -> link;
        free (temp);
        ctr++;
    } else
        trav = &(*trav) -> link;
    }
}
```

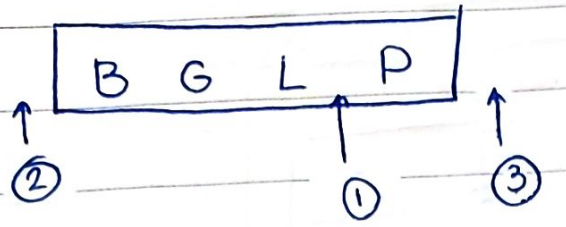
* TWO conditions bcs we dont need to reach the end of the list every time.

// Test Cases

1. A B A C (delete B where will pointer be?)
2. A C B B

Date: / /

//insert Sorted () (PPN) - 2 conditions



1.) Insert N
2.) Insert A
3.) Insert S

- Definition

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

- Code

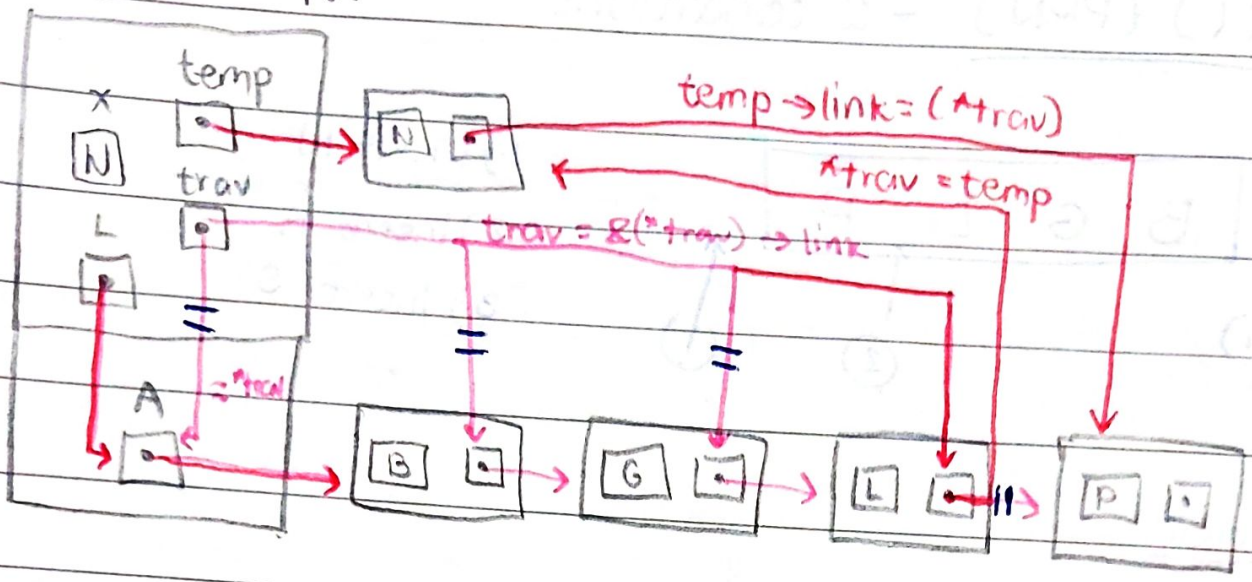
```
void insertSorted (LIST *L, char elem) {
    LIST *trav, temp;

for (trav = L; (*trav) != NULL && (*trav) -> data < elem;
                                    trav = &(*trav) -> link)
    ;

    temp = (LIST) malloc (sizeof (struct node));
    if (temp != NULL) {
        temp -> data = elem;
        temp -> link = *trav;
        *trav = temp;
    }
}
```

* Two conditions : either go thru whole list OR stop when data is bigger than elem.

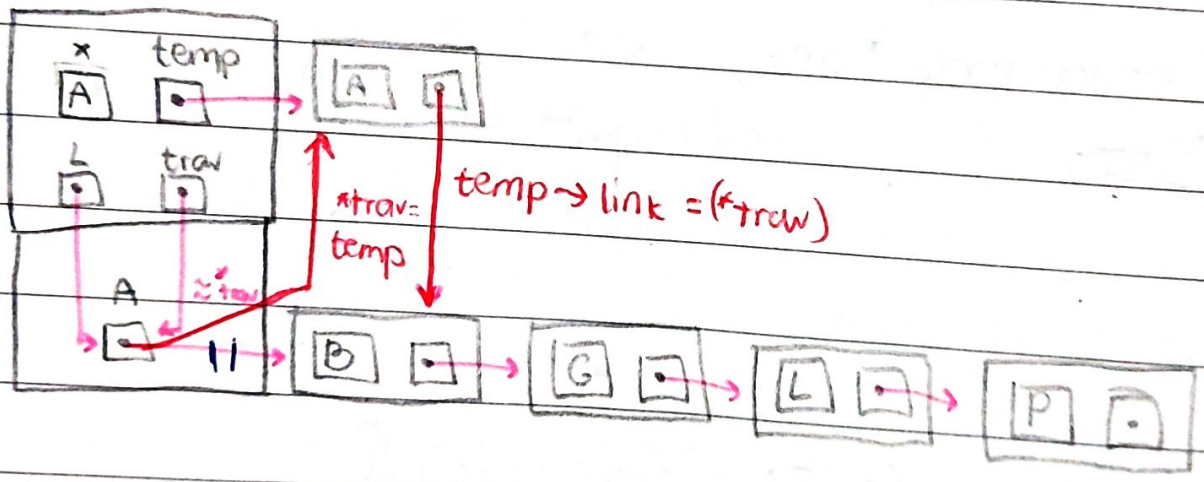
* butterfly notation

Date: / /

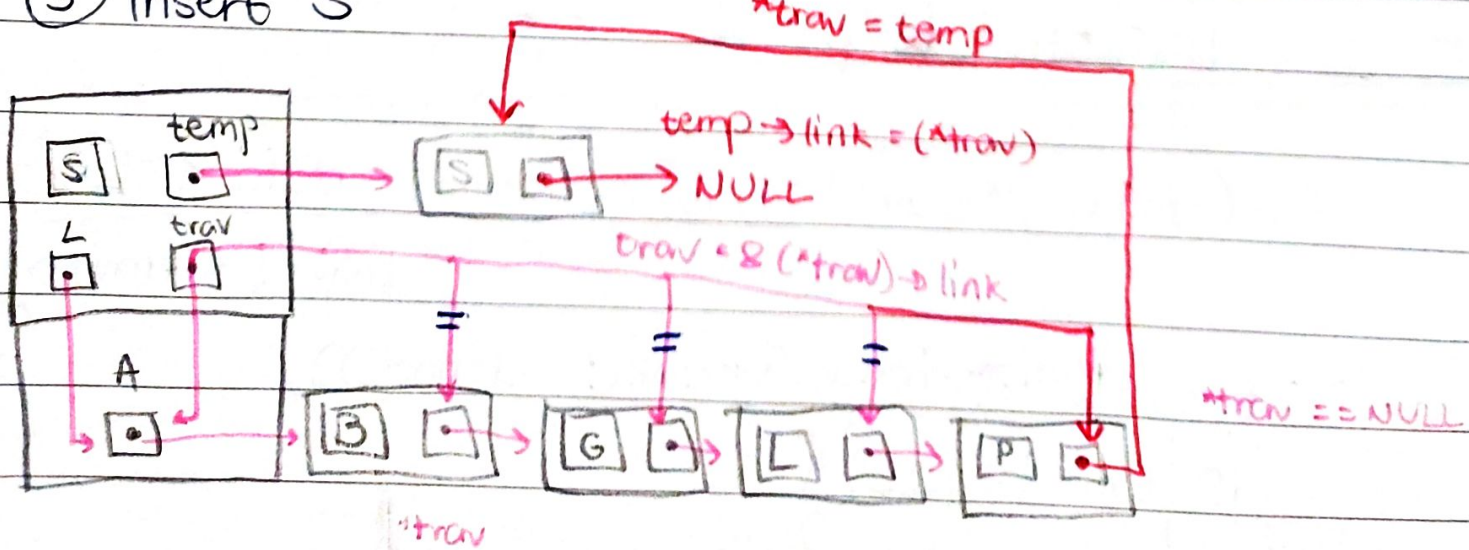
1. Insert N.



2. Insert A



3. Insert S



* PRACTICE PROBLEM

Date: / /

Remove all instances of chocolate that weigh less than 10.0 g & place in new array list. Return list to main. Given is a list.

// linked list + array list problem

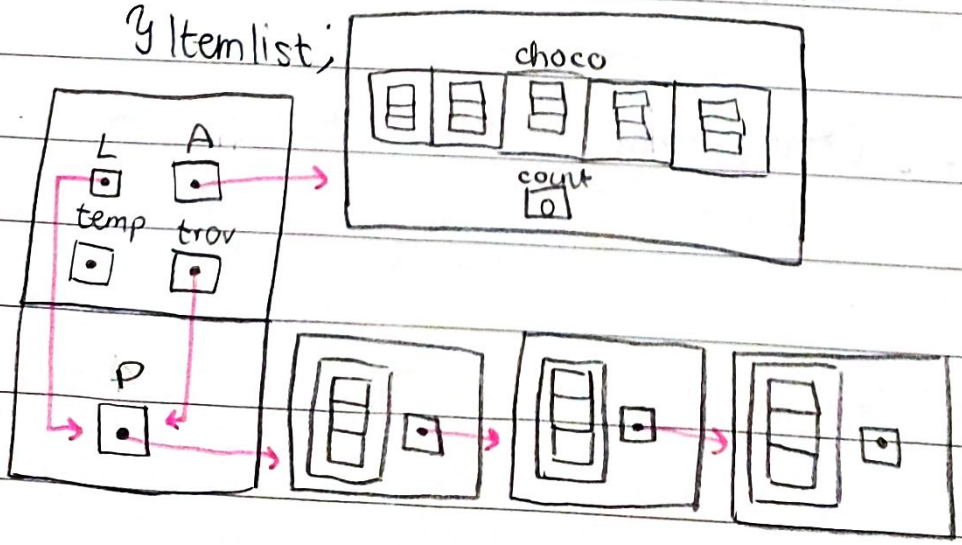
* Definition

```
#define MAX 10
typedef struct {
    char Name[16];
    float weight;
    char Brand[16];
} Item;
```

```
typedef struct node {
    Item choco;
    struct node * next;
} * List;
```

ARRAY DATA STRUCTURE

```
typedef struct node2 {
    Item choco[MAX];
    int count;
} Itemlist;
```



Date: / /

• Code

```
Itemlist chocoInstance (List *L) {
    List *trav, temp;                          * dynamically allocate A so we can return to main.
    Itemlist A = (Itemlist) malloc (sizeof (struct node2));
    A -> count = 0;        * initialize count

    for (trav = L; (*trav) != NULL ; ) {       * only one condition in for loop, other condition is inside.
        if ((*trav) -> Choco. weight < 10.0 && A -> count != MAX) {
            Strcpy( A -> choco [A -> count ]. Name, (*trav) -> Choco. Name );
            Strcpy (A -> choco [A -> count] Brand, (*trav) -> Choco. Brand);
            A -> Choco [A -> count]. weight = (*trav) -> choco. weight;
            A -> count ++;
        }
        temp = (*trav);
        (*trav) = temp -> link;
        free (temp);
    } else
        trav = & (*trav) -> link;
}
}
return A;
}
```

* check if elem's weight is < 10.0g

* if yes, store into inside array cell & update count

* CAN DO WHOLE STRUCTURE ASSIGNMENT!

* delete node code must be after so we can store values first

* move to next node

* return the new array list

Linked List	Curs
Pointer to Node	int
Pointer to	

- CURSOR-BASED IMPLEMENTATION -

* Cursor-based Implementation

- used by all languages that do not support pointers.
- uses INDICES instead
- combination of array & linked list
  - ↳ uses the STRUCTURE
  - ↳ uses method of ELEMENT MANIPULATION
- simulates linked lists BUT w/o pointers & it puts elements in contigouous space of memory → in a VIRTUAL HEAP

* Using Array Cells

Assumptions:

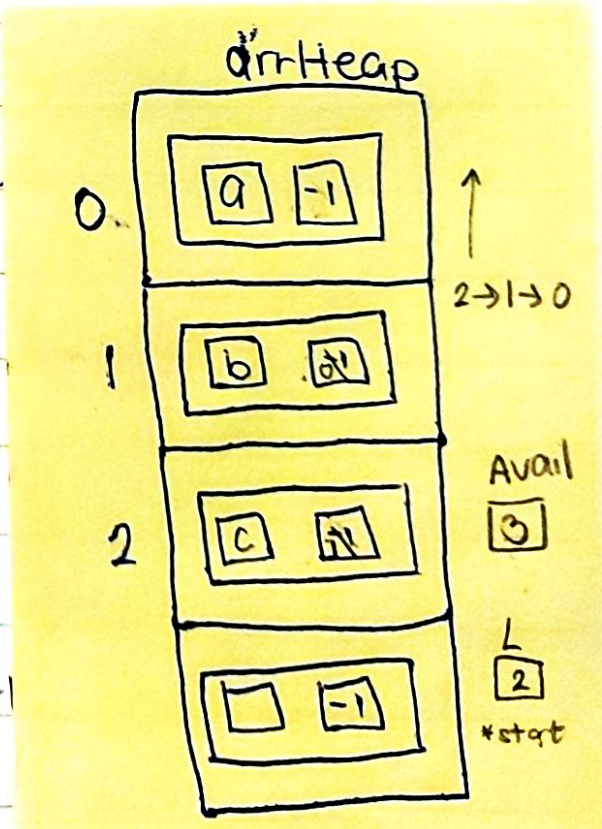
1.) Each array (cell) is a structu containing data & link field.

2.) Next field is -1, if its not pointing to any other cell.

-1 ≈ NULL

3.) Order of array cell is the availability from 0 to MAX-1

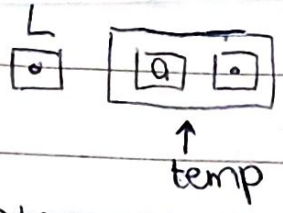
4.) Elements a, b, c will be inserted in List L using inser



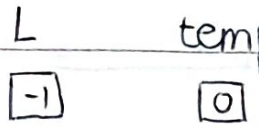
Linked List	Cursor-based
Pointer to Node	int (index of array)
Pointer to Pointer to Node	int *
NULL	-1

* Example of inserting `a' at 1st position

Linked List



List using Array Cells



• Statement:

arrHeap	Array	[ ]	arrHeap [0]
arrHeap [0]	structure	.	arrHeap [0]. data
arrHeap[0]. data	char		

= `a';

* arrHeap is NOT a pointer. so use : .

So,

* temp is next availability

* L is the start of the list!

* Example of inserting `b'

• Statement :

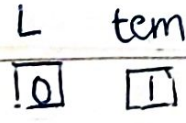
arrHeap [temp]. data = `b';

arrHeap [temp]. link = L;

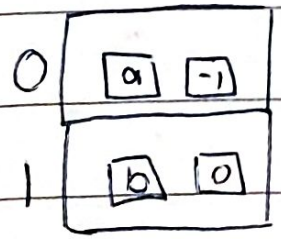
* gets the value of L

then, increment L & temp.

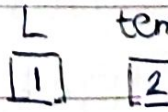
Now,

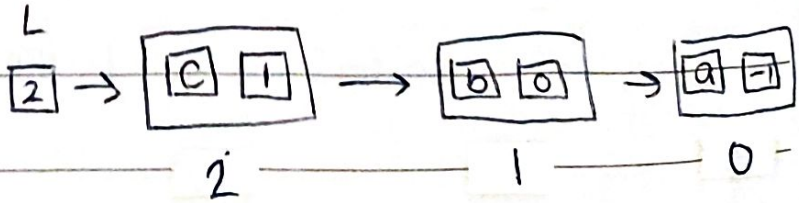


* L is start of list.
Temp is next available cell



* Insert `c'





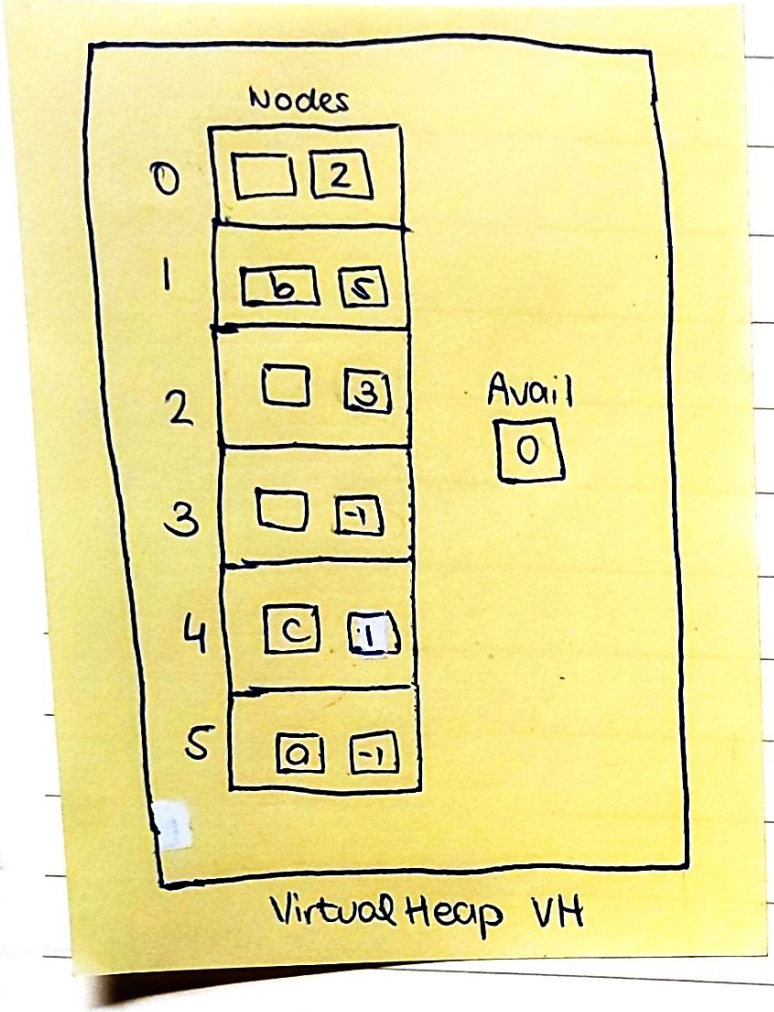
Linked List View

Date: / /

* Elements in random cells

How to know which is NEXT AVAILABLE CELL?

> Avail > Linking available spaces



* List starts at [4]

L
[4]

* next available cells are:

[0] → [2] → [3]

Avail
[0]

the whole list looks like this:

[4] → [1] → [5]

* Creating IMAGINARY HEAP (VH)

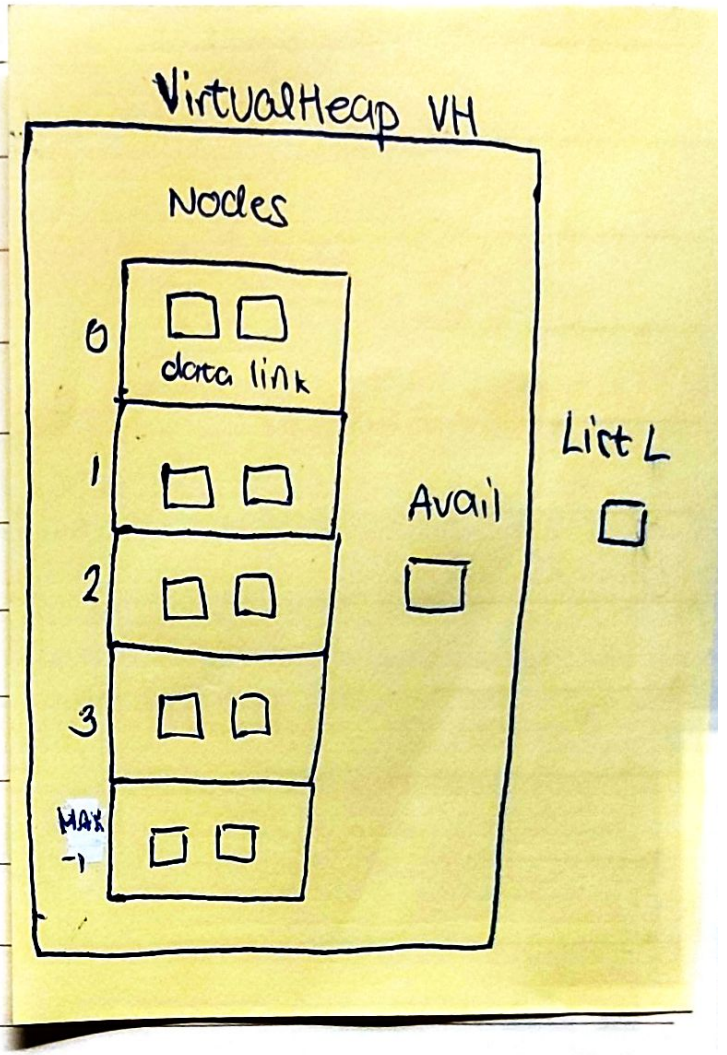
- can have multiple lists at once BUT can only store some elements; the REAL HEAP can store a mix of elements.

* PURPOSE: to limit memory space
to avoid shifting

• Definition:

① VirtualHeap

```
typedef struct node {
    char data;
    int link;
} nodetype

#define MAX 10

typedef struct heap {
    nodetype Nodes[MAX];
    int avail;
} VirtualHeap;
```



② List

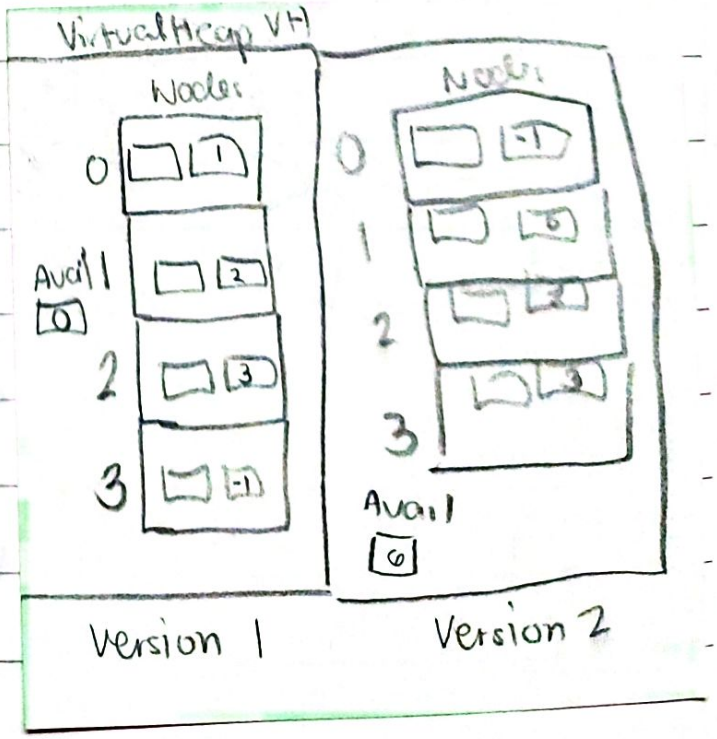
```
typedef int List;
```

clarify if NOT virtualHeap*

• Initialise // 2 versions

```
// Version 2        #define MAX 7
VirtualHeap initVH() {
    VirtualHeap VH;
    VH→Avail = 0;
    int i;
for(i=0; i<MAX; i++) {
    VH→Nodes[i].link = i-1;
}
}
```

*can also use (*VH)



*Heap

1. initialize VH - link all nodes in the array
   - "-1" indicates the end of the list
2. allocspace() - remove first available node need in VH & return index of node
   - if "-1" returned = NO SPACE()
3. deallocspace() - free()
   - given index, put back node to availables

// deleteFirst() - Linked List *to show same logic for VH

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

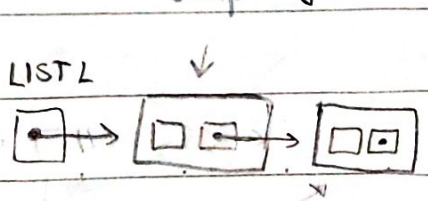
* follows the SAME LOGIC for delete() for VH!

- Code

```
void deleteFirst (LIST *L) {
    LIST *trav, temp;

    if ((*L) != NULL) {
        trav = L;
        temp = *trav;
        *trav = temp -> link;
        free (temp);
    }
}
```

VERSION 2

```
LIST temp;

if((*L) != NULL) {
    temp = L;
    *L = temp -> link;
    free (temp);
}
```



* insertFirst() - fn. will insert at the 1st position, the node in the list of available nodes

Date:

- HEAP MANAGEMENT FUNCTIONS CODE -

```
typedef struct node {
    char data;
    int link;
} nodetype;

typedef int List;

typedef struct heap {
    nodetype Nodes [MAX];
    int avail;
} VirtualHeap;
```

//initVH() - Initialize Virtual Heap

```
VirtualHeap initVH () {
    VirtualHeap VH;
    int index;
    VH . avail = MAX -1;

    for (index = 0; index ≤ MAX-1; index++) {
        VH . Nodes [index] . link = index-1;
    }
    return VH;
}
```

* actualy has 4 VERSIONS:
- locally declared
- void / ptr
- iteration: -1 → MAX-1
: MAX-1 → -1

*use . bcs VH is LOCALLY DECLARED!

WHY LOCALLY DECLARE VH?

Date: / /

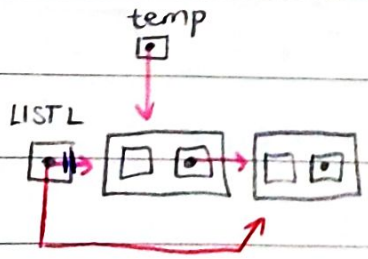
// allocSpace() / deleteFirst() - CREATE AND GIVE SPACE

```
int allocSpace (VirtualHeap *VH) {
    int temp = -1;
① if (VH→avail != -1) {
    ② temp = VH→avail;
    ③ VH→avail = VH→Nodes[VH→avail].link;
  }
④ return temp;
}
```

* SIMILAR LOGIC TO LINKED LIST deleteFirst()!

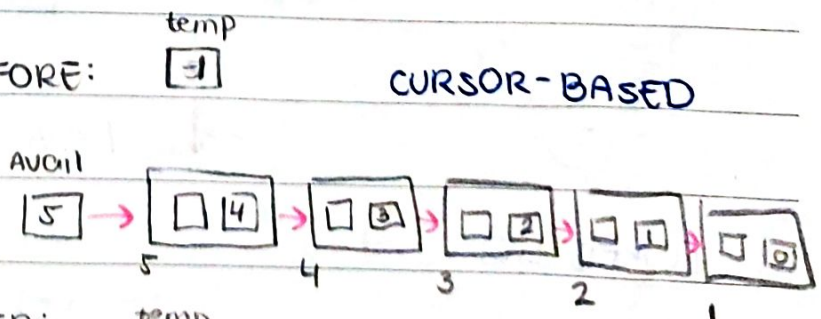
```
① if ((*L) != NULL) {
② temp = *L;
③ *L = temp→link;
④ free (temp);
}
```

- Execution Stack



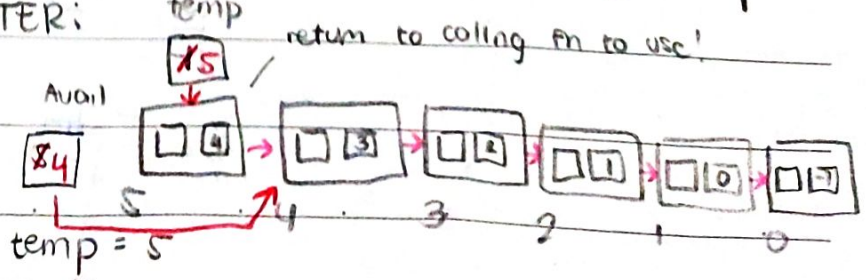
LINKED LIST

- BEFORE:



CURSOR-BASED

- AFTER:



Date: / /

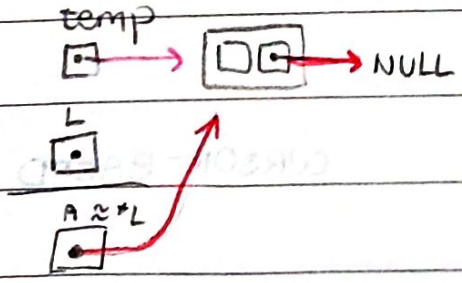
// deallocSpace / insertSpace - PUT BACK NODE TO AVAILABLE NODE

```
void deallocSpace (VirtualHeap *VH, int index) {

    (1) VH → Nodes[index].link = VH → avail;
    (2) VH→ avail = index;

}
```

* SIMILAR LOGIC TO LINKED LIST insertFirst()!

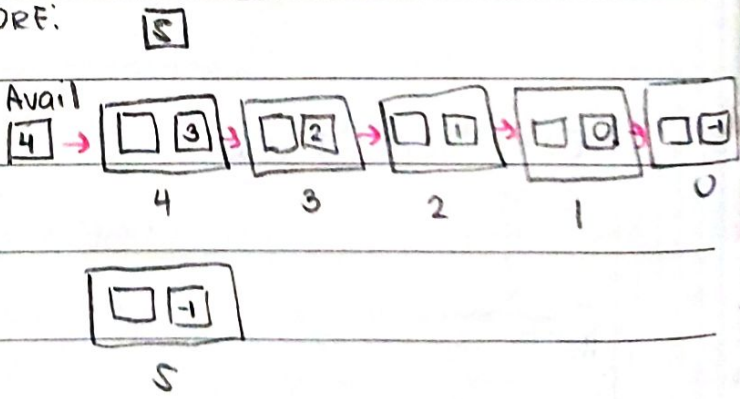
```
temp = (LIST) malloc(sizeof
        (struct node));
temp → data = x;
(1) temp→ link = *L;
(2) *L = temp;
```

• Execution Stack

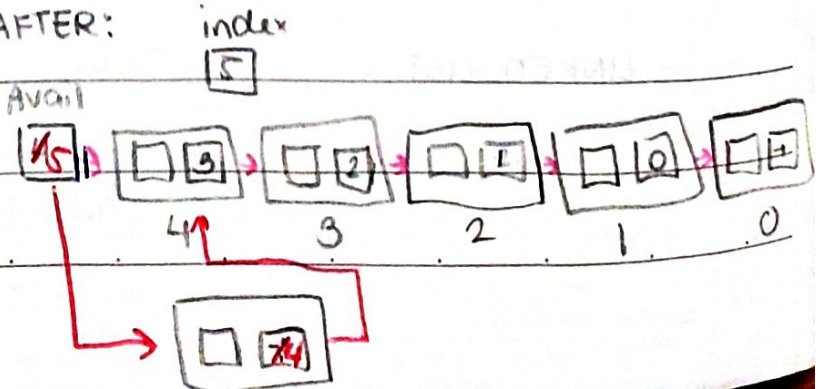


LINKED LIST

• BEFORE:



• AFTER:



- CURSOR BASED LIST OPERATIONS - Date: / /

a.) initList()

b.) insertLast() / insertFirst()

c.) deleteElem()

d.) displayList()

//insert()

① Allocate space for a new node

② Assign data to node

③ Link node to remainder of list

④ Link area of insertion to new node

(LL)	(CB)
• temp = (LIST) malloc(sizeof(struct node));	• temp = allocSpace()
• temp → data = elem	VH → Nodes [temp]. data = elem
• temp → link = *trav	VH → Nodes [temp]. link = *trav
• *trav = temp	*trav = temp

//delete()

① Let temp hold node to be changed

② Link node before temp to the node after temp

③ Free the memory space consumed by temp

(LL)	(CB)
• temp = *trav	
• *trav = temp → link	
• free(temp)	

Date: / /

//initList ()

* has 2 versions: void (ptr) & locally declared

```
void initList (List *L){
    (*L) = -1;
}
```

```
List initList (){
    List L;
    return L = -1;
}
```

* can even forego declaring a variable & just return -1 bec its just an integer

//insertFirst()

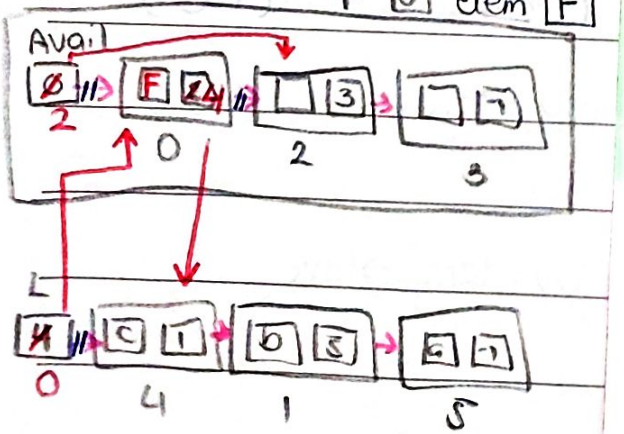
Given: VH, List L & elem to insert

```
void insertFirst (List *L, VirtualHeap *VH, char elem) {
    List temp = allocSpace (VH);
    if (temp != -1){
        VH -> Nodes [temp]. data = elem;
        VH -> Nodes [temp]. link = *L;
        *L = temp;
    }
}
```

* same logic as linked list insertFirst()

allocSpace() = 0

temp 0 elem F



1. Alloc space from VH using allocSpace()
   - check if allocation was successful!
2. Assign elem to that node using temp as address
3. Link temp node to node pointed to by L. (head).
4. Link L (head) to temp node to reconnect list.

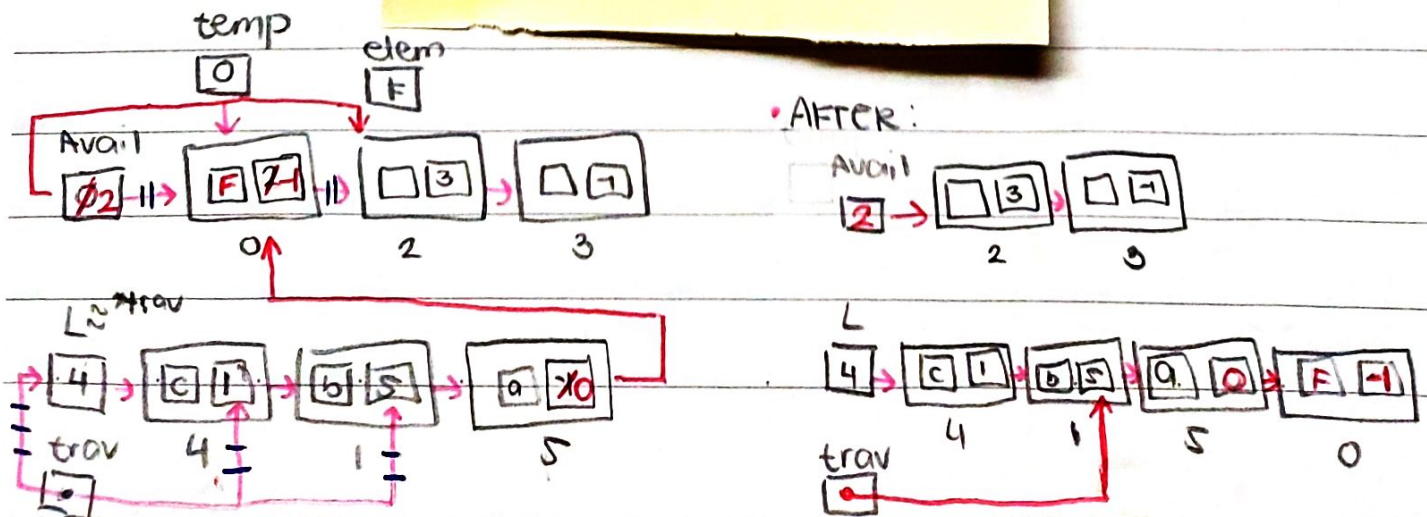
Date: / /

//insertLast () * uses PPN!

```
void insertLast (VirtualHeap *VH, List *L, char elem) {
    int temp = allocSpace (VH);
    ① int *trav;
    for (trav = L; (*trav) != -1; ② ③ trav = &VH->Nodes[*trav].link) ④ {
        if (temp != -1) {
            VH->Nodes[temp].data = elem;
            VH->Nodes[temp].link = -1;
            VH->Nodes[*trav].link = temp;
        }
    }
```

* SIMILAR LOGIC TO LINKED LIST insertLast ()!

```
① List *trav;
for (trav = L; (*trav) != NULL; ② trav = ③ &(*trav)->link) ④
```



Date: /

// displayList() * just PASS BY COPY!

```
void displayList (List L, VirtualHeap VH) {
    List trav;

for (trav =L; trav != -1; trav = VH.Nodes [trav].link ) {
    printf ("%c", VH.Nodes [trav]. data);
}
```