Linked List Insertion and Deletion

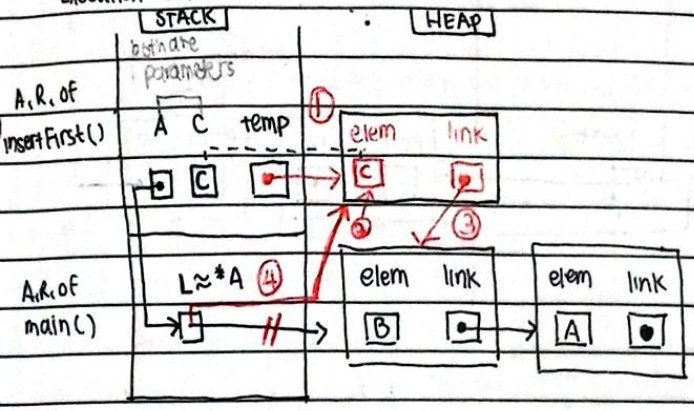
use this structure definition:

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

INSERT FIRST

```
void insertFirst (LIST * A, char c) {
    LIST temp = (LIST) malloc (sizeof (struct node));
① // temp is a pointer to a node, used for insertion (no need traversing)
    temp -> elem = c;   // filling in the node ②
    temp -> link = *A;  // connecting the node to its next ③
    *A = temp;          // placing the node to first ④
}
```

Execution Stack

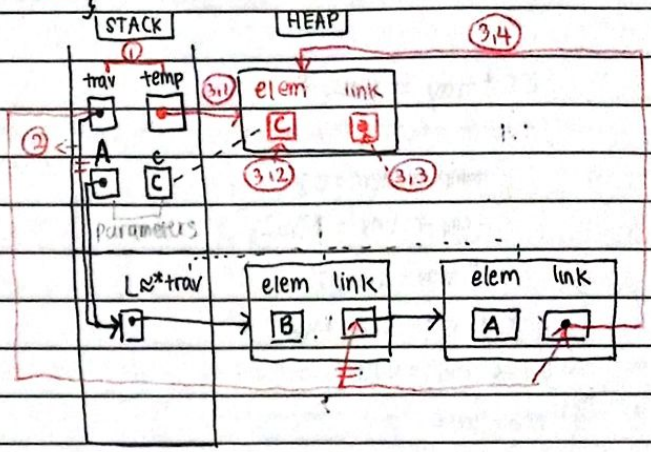


Oh btw, sab you forgot to ask if dynamic allocation is successful by asking if temp != NULL)

INSERT LAST

ideally it's better to do malloc first

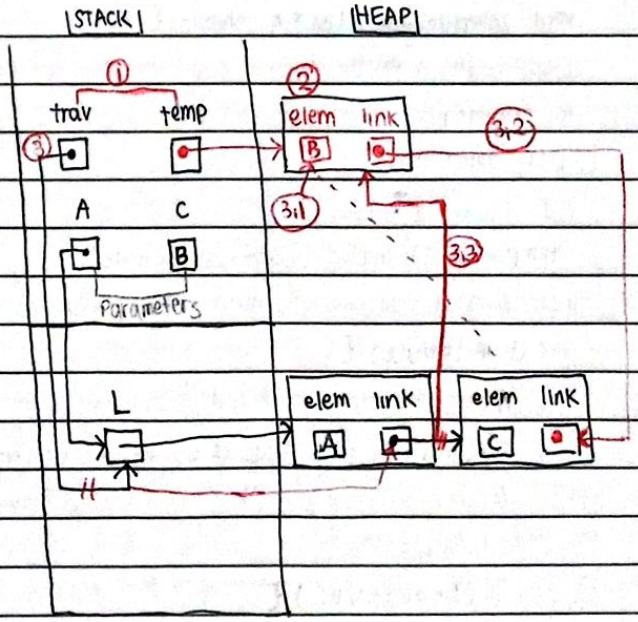
```
void insertLast (LIST * A, char c) {
① // temp is a PN, used for insertion, trav is a PPN used for traversing & accessing
    LIST temp, * trav;
② // loop for accessing
    for (trav = A ; * trav != NULL ; trav = & (* trav) -> link) {}
③ // insertion process
3.1 temp = (LIST) malloc (sizeof (struct node));
3.2 temp -> elem = c;   // filling in the node
3.3 temp -> link = NULL; // since last na man
3.4 * trav = temp;
}
```



INSERT SORTED

```
void insertSorted (LIST * A, char c) {
① // temp is a PN used for insertion, trav is a PPN used for traversing and accessing
    LIST temp, * trav ;
② // do dynamic allocation
    temp = (LIST) malloc (sizeof (struct node));
→ // checking if dynamic allocation is successful
    if (temp != NULL) {
③ //loop for accessing & additional condition for sorting
    for (trav = A ; trav != NULL && (*trav) -> elem <= c ; trav = & (*trav) -> next) {}
3.1 temp -> elem = c;
3.2 temp -> link = *trav;
3.3 * trav = temp;
    }
}
```

If unique it's just < only cuz if <=, you are allowing duplicates

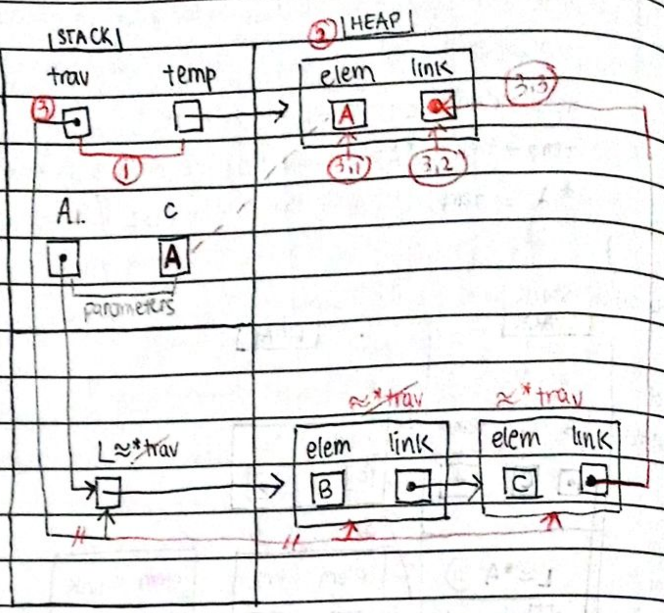


[SPECIAL SECTION]

Insert Unique with Positions

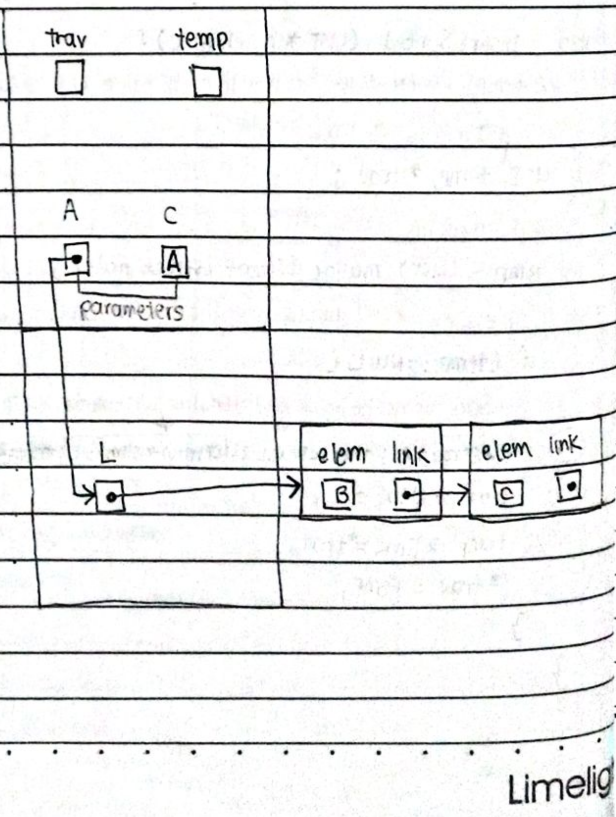
Unique & Last

```
void insertUnique (LIST * A, char c) {
① //temp is a PN used for insertion, trav is a PPN used for
  traversing and accessing
  LIST temp, * trav;
② // do dynamic allocation
  temp = (LIST) malloc (sizeof (struct node));
  // checking if dynamic allocation is successfull
  if (temp != NULL) {
③   // loop for accessing & additional condition for unique
    for (trav = L; *trav!=NULL && (*trav)->elem!=c; trav= &(*trav)->link){}
    // validation: You have reached end of the loop because there's
    no duplicate
    if ( *trav == NULL) {
    // do insertion ( SEE how this is similar to insert Last)
(3.1)     temp -> elem = c;        //fill in the node
(3.2)     temp -> link = NULL;     // since last na man
(3.3)     *trav = temp;
    } // for *trav == NULL
  } // for temp != NULL
} // Whole close
```



Unique & First

```
void insertUnique (LIST * A, char c) {
① //temp is still a PN for insertion, trav here tho is a PN used
  for holding that other node just for linking purposes
  LIST temp, trav;
  // do dynamic allocation
  temp = (LIST) malloc (sizeof (struct node));
  // checking if dynamic allocation is success full
  if (temp != NULL) {
    // loop for accessing & additional condition for unique
    for (trav = *L ; trav!=NULL && trav-> elem != c; trav=trav->link){}
    //validation: You have reached end of the loop because there's
      no duplicate
    if ( trav==NULL ) {
        temp-> elem=c;   // Fill in the node
        temp -> link= trav;   // linking the inserted to the list
        trav= temp; // making what is inserted the FIRST
    }
  }
}
```



[continuation]

Linked List Implementation (Delete)

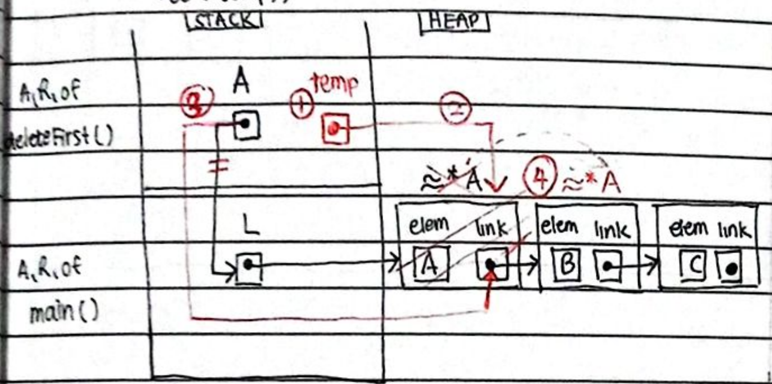
DATA STRUCTURE DEFINITION:

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

Delete First

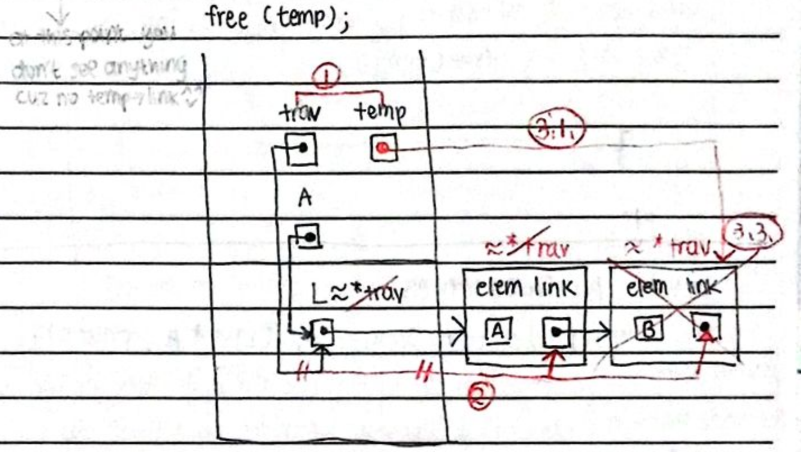
```
void deleteFirst (LIST * A){
①  // temp is a PN used to hold the first node (kay mao man
   ang i-delete)
   LIST temp;
②  // Separating this but you can do this one line ra, let
   temp point to the first node
   temp = *A;
③  // Let the original LIST (in the parameters) point to the
   next node
   * A = temp->link;
④  // Detaching the (originally) first node by free-ing
   free (temp);
```

(deletion process)



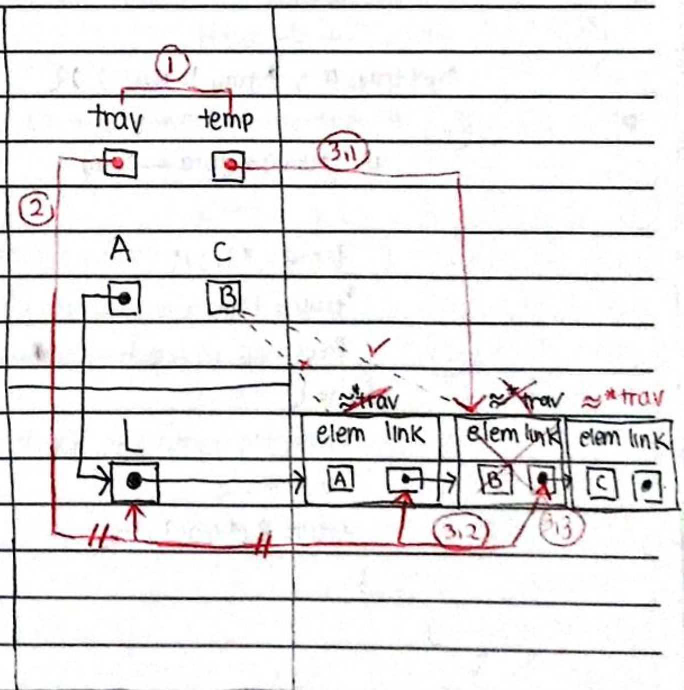
Delete Last

```
void deleteLast (LIST * A){
①  // temp is a PN used to hold the node to delete to,
   trav is a PPN used for traversing & accessing (obviously)
   LIST temp, *trav;
②  // loop for accessing
   for( trav = A; *trav!= NULL; trav= &(*trav)->link) {}
③  // deletion process
(3.1) // let temp point to target node
   temp =*trav;
(3.2) // let original LIST point to the next node
   *trav= temp->link; (this won't be pointing to trash, don't worry :))
(3.3) // Detaching the last node by free-ing
   free (temp);
```



Delete Element

```
void deleteElement (LIST *A, char c){
①  //temp is a PN used to hold the node to delete to,
   trav is a PPN used for traversing and accessing
   LIST temp, *trav;
②  // loop for accessing & additional condition for comparing
   for( trav=A; *trav!=NULL && (*trav)->elem != c;
        trav= &(*trav)->link) {}
   // validation: you have found the element & you didn't
   reached the last element
   if ( *trav != NULL) {
③      //do deletion
(3.1)  temp = *trav; // let temp point to target node
(3.2)  *trav= temp->link; //let ORIGINAL LIST point to the next node
(3.3)  free (temp); //Detaching the last node by free-ing
   }
}
```

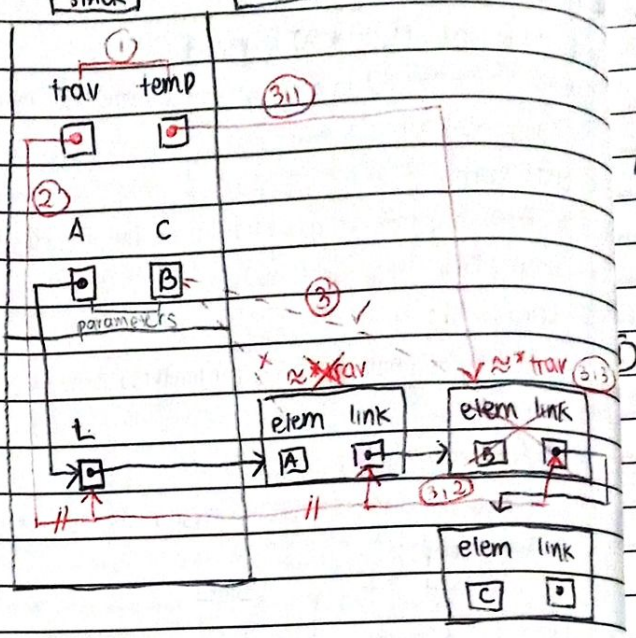


Limelight

More Derived Deletes

Delete Element Sorted

```
Void deleteElementSorted (LIST *A, char c){
(1)   // temp is a PN used to hold the node to delete to,
        trav is a PPN used for traversing & accessing
      LIST temp, *trav;
(2)   // loop for accessing & additional condition for sorted
      for(trav=A; *trav!=NULL && (*trav)->elem<c; trav=&(*trav)->link){}
(3)   // validation. you saw the element
      if(*trav!=NULL && (*trav)->elem==c){
(3.1)   // let temp point to target node
          temp = *trav;
(3.2)   // let original LIST point to the next node
          *trav = temp->link;
(3.3)   // Detaching the chosen node by free-ing
          free(temp);
      }
}
```



Delete All Occurences

```
void deleteAllOccurences (LIST *A, char c){
(1)   // temp is a PN used to hold the node to delete to,
        trav is a PPN used for traversing & accessing
      LIST temp, *trav
(2)   // loop for accessing (1 condition only since we need
        to check all elements)
      for(trav=A; *trav!=NULL; ){
(2.1)   // additional condition here, elem is found
        if((*trav)->elem==c){
(3)       // do deletion
(3.1)     temp = *trav; // let temp point to target node
(3.2)     *trav = temp->link; // let trav point to next node
(3.3)     free(temp); // detaching the last node by free-ing
        }else{
          // move trav pointer only if node will not
            be removed
          trav = &(*trav)->link;
        }
}
```