# Abstract Data Type List

ABSTRACT DATA TYPE (ADT) - a mathematical model together with a set of operations defined on the model

A BREAKDOWN

ADT LIST

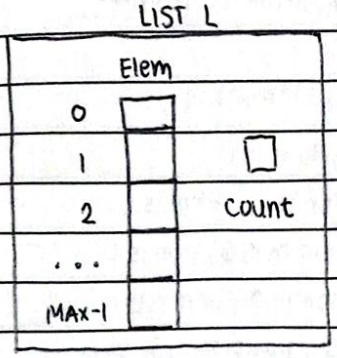
List → a sequence of 0 or more elements

ADT List → a sequence of 0 or more elements together with a set of operations such as insert, delete, member

Representations or Implementations of ADT List

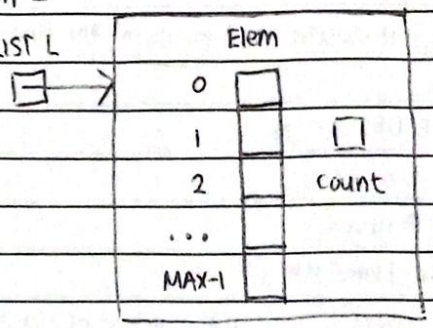
① Array Implementation (4 versions)

Version 1



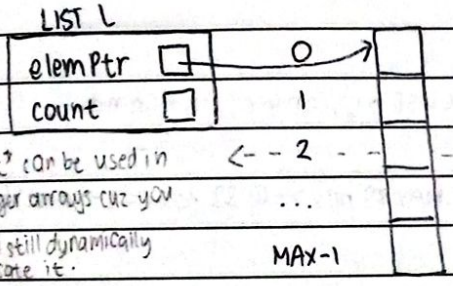

```
# define MAX 5
typedef struct {
    char Elem[MAX];
    int count;
} LIST;
```

Version 2




```
# define MAX 5
typedef struct {
    char Elem[MAX];
    int count;
} *LIST;
```

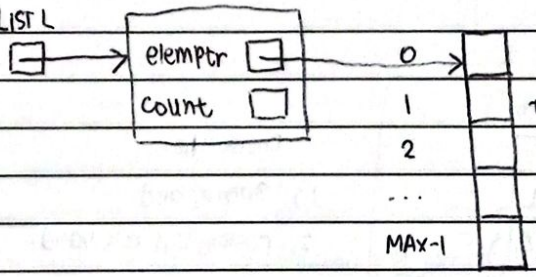
Version 3




int* can be used in integer arrays cuz you will still dynamically allocate it.

```
# define MAX 5
typedef struct {
    int* elemPtr;
    int count;
} LIST;
```

Version 4

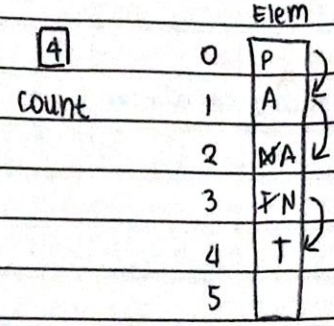



```
define MAX 5
typedef struct {
    int* ElemPtr;
    int count;
} *LIST;
```

② Linked List Implementation → Singly Linked or Doubly Linked

③ Cursor Based Implementation

Array Implementation → elements are stored in contiguous cells of the array




x = count to pos exclusive | y = count to pos inclusive

→ Elem[4] = Elem[3]
Elem[3] = Elem[2]
Elem[2] = Elem[1]

= OKAY, HERE'S A PRACTICE! =

Given the LIST, the element, and position, the function will insert the element at the given position if it exists.

DATA STRUCTURE DEFINITION

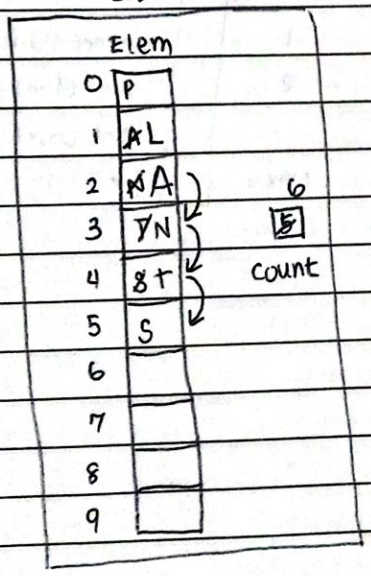
```
# define MAX 10
typedef Struct{
    char Elem[MAX];
    int count ; // actual number of elements in
                   the array
} LIST;
typedef int Position; // first position is in index 0
```

Simulation : Position is 1, Element is L

TEST CASES (we can casually do this)

1. List is empty, position is 1, Elem is L.
2. List has 5 elements, position is 6, Elem is L.
3. List is empty, position is 0, Elem is L
4. List has 6 elements, position is 3, Elem is L

LIST L




Check the:

1.) Space, and
2. position, if it's valid

```
void insertPosition (LIST *L, char c, Position ndx){
    int ctr;
    if(L->count < MAX && ndx >= 0 && ndx <= L->count){
        for(ctr=L->count ; ctr != ndx ; ctr--){
            L->Elem[ctr] = L->Elem[ctr-1];
        }
        L->Elem[ndx] = c; // placing character in pos
        L->count++;
    }
}
```

array restriction (size)

valid index of array (non-negative)

check if index exists in array

shifting

= Deletion Practice Let's Gaurrr =

Given the list and position, the function will delete the element at the given position and return TRUE if an element is deleted, otherwise FALSE.

```
typedef enum Boolean = {TRUE, FALSE};
        Boolean deletePosition (LIST *L, Position pos){
            int y;
            if(pos < L->count && pos >= 0){
                for(y = pos; y < L->count ; y++){
                    L->elem[y] = L->elem[y+1];
                }
                L-> count --;
            }
            return (pos <= L->count)? TRUE: FALSE;
```

≈≈≈≈≈ A SEGWEY ≈≈≈≈≈ This is an introductory pag 2/9/2023

[A] Where in memory is the string "Hello" stored in each of the following:

a. `char str1[10] = "Hello";` STACK

b. `char *str2 = "Hello";` "STRING POOL"

c. `char *str3 = (char*) malloc (sizeof(char) * 10);` HEAP( POINTER in STACK)

```
if (str3 != NULL){
    strcpy (str3, "Hello");
}
```

d. `char str4[] = "Hello";` STACK

Yeap, they related, bro. Cuz JAVA terms.

"Immutable" → cannot be changed in Java

# Cursor Based Implementation of List

Let's compare Arrays & Linked List First

NOTE: This is a generalized thing, if you need to traverse then, the running time is O(N)

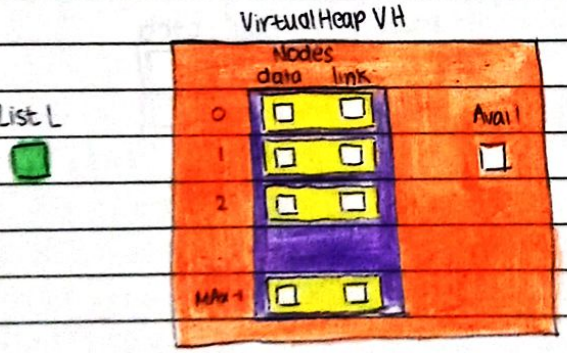
| Array Implementation | vs | Linked List |
|---|---|---|
| » fixed size | | » InsertFirst() => O(1) (Constant Time) |
| » finite elements | | » deleteLast() or deletePOS () => O(N) |
| » Insert: Check space | | |
| » Direct access | | |
| » Elements are stored in contiguous cells of the array | | |

CURSOR BASED → aims to lessen memory managed by programmer

So now, let's compare Linked list and Cursor Based

| Linked List | vs | Cursor Based |
|---|---|---|
| Pointer to Node | | int |
| Pointer to Pointer to Node | | int* |
| NULL | | -1 |
| Pointer in stack, node in heap | | Pointer in stack, node either in stack or heap |
| temp→data = 'a'; | | arrHeap[temp].data = 'a'; |
| temp→link = NULL; | | arrHeap[temp].link = -1; |

So this is how a cursor based looks like...




```
#define MAX 5

typedef struct {
    int data;
    int link;
} NodeType;

typedef struct {
    NodeType Nodes[MAX];
    int Avail;
} VirtualHeap;

typedef int List;
```

Virtual Heap Management

1.) initHeap

2.) deleteFirst ≈ malloc
≈ allocSpace

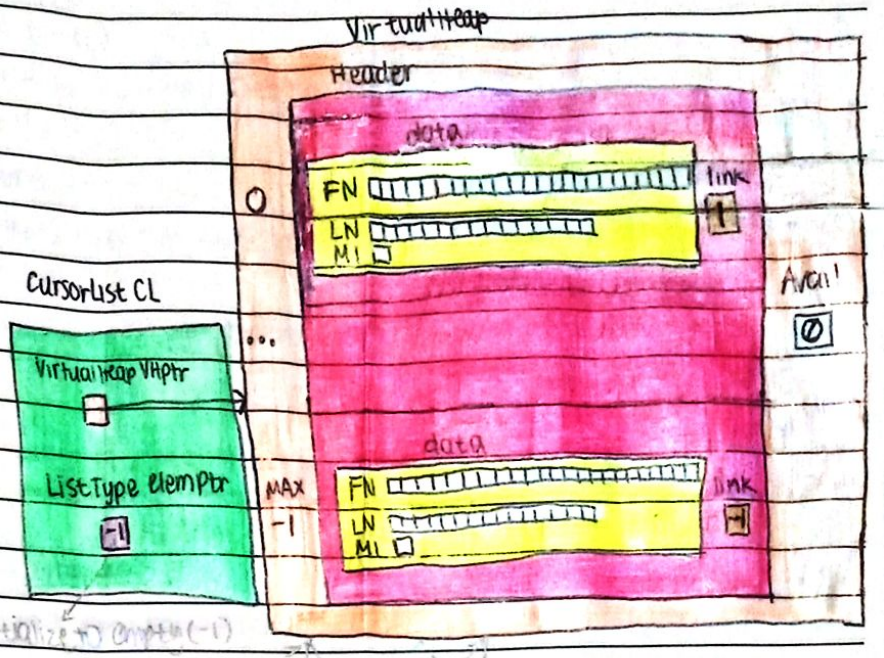
3.) insertFirst ≈ free
≈ deallocSpace

February 10, 2023
[continuation]

# Cursor Based Operations

① Draw variable CL with the Data Structure Definition and assume that the list is initialized to be empty and VHptr is pointing to an initialized Virtual Heap.

```
#define MAX 10

typedef struct {
    char FN[24];
    char LN[16];
    char MI;
} NameType;

typedef struct {
    NameType data;
    int link;
} NodeType;

typedef int ListType;

typedef struct {
    VirtualHeap * VHptr;
    ListType elemptr;
} CursorList;

CursorList CL;
```




```
typedef struct {
    NodeType Header[MAX];
    int Avail;
} VirtualHeap;
```

② Given the cursorlist, function initCursorList() will initialize the cursorlist to be empty (i.e. elemptr is -1) and the VHptr will point to an initialized Virtual Heap.

```
void initCursorList (CursorList * CL) {
    int ctr;
    CL→elemptr = -1;
    CL→VHptr = (VirtualHeap*) malloc (sizeof(VirtualHeap));
    CL→VHptr→Avail = 0;
    for (ctr=0; ctr<MAX-1; ctr++) {
        CL→VHptr→Header[ctr].link = ctr+1;
    }
    CL→VHptr→Header[MAX-1].link = -1;
    // last index of the array is MAX-1, which is not the case for ③
}
```

start index of header is 0 // means that the next of the present node is the present node +1

VirtualHeap VH

| | Header data | Header link | Avail |
|---|---|---|---|
| 0 | | -1 | |
| 1 | | 0 | |
| 2 | | 1 | |
| 3 | | 2 | 9 |
| 4 | | 3 | |
| 5 | | 4 | |
| 6 | | 5 | |
| 7 | | 6 | |
| 8 | | 7 | |
| 9 | | 8 | |

③ Write the code for the function InitVirtualHeap()

```
void initVirtualHeap (VirtualHeap* V){
    int ctr;
    V→Avail = MAX-1;
    for(ctr=MAX-1; ctr>=0; ctr--) {
        V→Header[ctr].link = ctr-1;
    }
    // in here the last index is 0 because its link is -1.
}
```

See how this is different in ②, in here -1 stays

Incorporate no.2 & no.3

```
initCursorList (CursorList * CL){
    CL→elemptr = -1; //Empty the list
    CL→VHptr = (VirtualHeap*) malloc (sizeof(VirtualHeap));
    CL→VHptr→Avail = MAX-1;
    initVirtualHeap (CL→VHptr);
}
```

// The link for ③ is better because you don't need to have an additional one line of code for the end of list na, which is the case in ②

# Cursor Based vs. Linked List Operations

Virtual Heap management functions

① void initVirtualHeap (VirtualHeap * VH)
↳ connects the nodes in the cursor based
↳ sets the avail field

② void insertNode (VirtualHeap * VH, int ndx) ≈ deallocSpace()
↳ free() on linked list
↳ insert node with given index in the available node list

③ int deleteNode (VirtualHeap * VH) ≈ allocSpace()
↳ malloc() on linkedlist
↳ delete a Node from the Available list of nodes and return index to the calling function. Returns -1 if there is no available node

| | LINKED LIST | CURSOR BASED |
|---|---|---|
| initializing | initList<br>↳ set to NULL | initCursorList<br>↳ set to -1 |
| Insert First | 1.) allocate space // check if not NULL<br>2.) populate data<br>3.) set link to head<br>4.) Set head to temp | 1.) int temp = allocSpace // check if not -1<br>2.) CB→VHptr→Nodes[temp].data = elem;<br>3.) CB→VHptr→Nodes[temp].link = CB→elemptr;<br>4.) CB→elemptr = temp; |
| Insert Last (PPN) | void insertLast (list* L, char elem){<br>List *trav;<br>for(trav=L; *trav!=NULL; trav=&(*trav)→link); | void insertLast (CBList* CB, char elem)<br>int * trav;<br>for (trav=&CB→elemptr;<br>*trav != -1;<br>trav=&CB→VHptr→Nodes[trav]<br>→link) |
| Delete Elem | 1.) traverse<br>2.) Set temp to node<br>3.) Set trav to temp→link<br>4.) free temp node | int temp;<br>1.) for( trav=&CB→elemptr; *trav!=-1 &&<br>CB→VHptr→Nodes[*trav].data!=<br>elem; trav=&CB→VHptr→Nodes<br>[*trav].link)<br>2.) temp = *trav;<br>3.) *trav = CB→VHptr→Nodes[temp].link;<br>4. deallocSpace(CB→VHptr, temp); |
| Display (PPN) | void displayList (List L){<br>List *trav;<br>for(trav=L; *trav!=NULL; trav=&(*trav)→link) | void displayList (CBList CB){<br>int *trav;<br>for(trav=&CB→elemptr;<br>*trav!=-1;<br>trav=&CB→VHptr→Nodes[*trav]<br>link) |