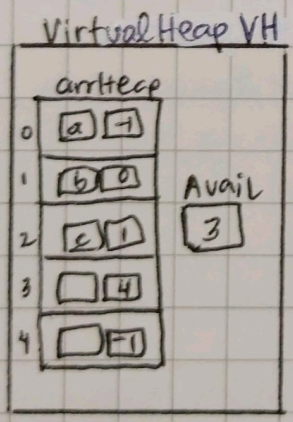
CURSOR BASED IMPLEMENTATION

↳ combines the best of array & linked list
↳ logic is the same as linked list but the data structure is an array.
↳ Although its like an array, no shifting is involved
↳ The point of cursor based is to avoid shifting of elements
↳ Cursor based also limits the space in where the linked list manipulation takes place.

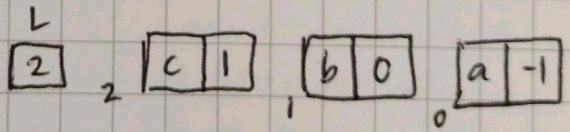
CURSOR BASED ⟶ ARRAY DATASTRUCTURE WITH LINKED LIST MANIPULATION

LINKED LIST	vs.	CURSOR BASED
pointer to node		int (index of the array)
pointer to pointer to node		int * (integer pointer)
NULL		-1

Linked list ⟶ dynamic memory
Cursor Based ⟶ Array cells.

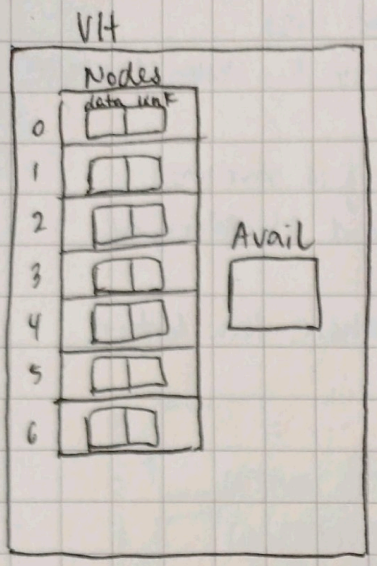


← THIS LIST IN DIFFERENT VIEW (TO HELP WITH LOGIC)



The list looks like a linked list.
Remember: Cursor Based logic Follows linked list logic.

ON ACCESSING:




```
#define MAX 7
typedef struct {
    char data;
    int link;
} nodetype;

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

NAME	DATATYPE	ACCESSING	EXAMPLES
VH	STRUCTURE	.	VH.Nodes
VH.Nodes	Array	[]	VH.Nodes [x-1]
VH.Nodes [x-1]	STRUCTURE	.	VH.Nodes [x-1].Link
VH.Nodes [x-1].Link	INTEGER		

ON CODE EFFICIENCY:

What makes code efficient?

- ★ conciseness
  - ↳ how short the code looks
- ★ Running time
  - ↳ affected by how many variables need to be initialized
- ★ Space
  - ↳ how many allocations does the code need?

The different datastructures manipulate space in different ways.

CURSOR - BASED MANAGEMENT FUNCTIONS

① INITIALIZING THE VIRTUAL HEAP

↳ to link all the nodes (array cells) together

② ALLOC SPACE ≈ malloc() ≈ deleteFirstNode

↳ TAKING THE FIRST AVAILABLE NODE & RETURNING IT TO THE CALLING FUNCTION (the node that was taken from the virtual heap acts as new space allocated to be used by the calling function.)

③ DEALLOC SPACE ≈ free() ≈ insertFirstNode()

↳ THE INDEX OF THE NODE IN THE FUNCTION CALL WILL BE RETURNED TO THE LIST OF AVAILABLE NODES IN THE VIRTUAL HEAP.

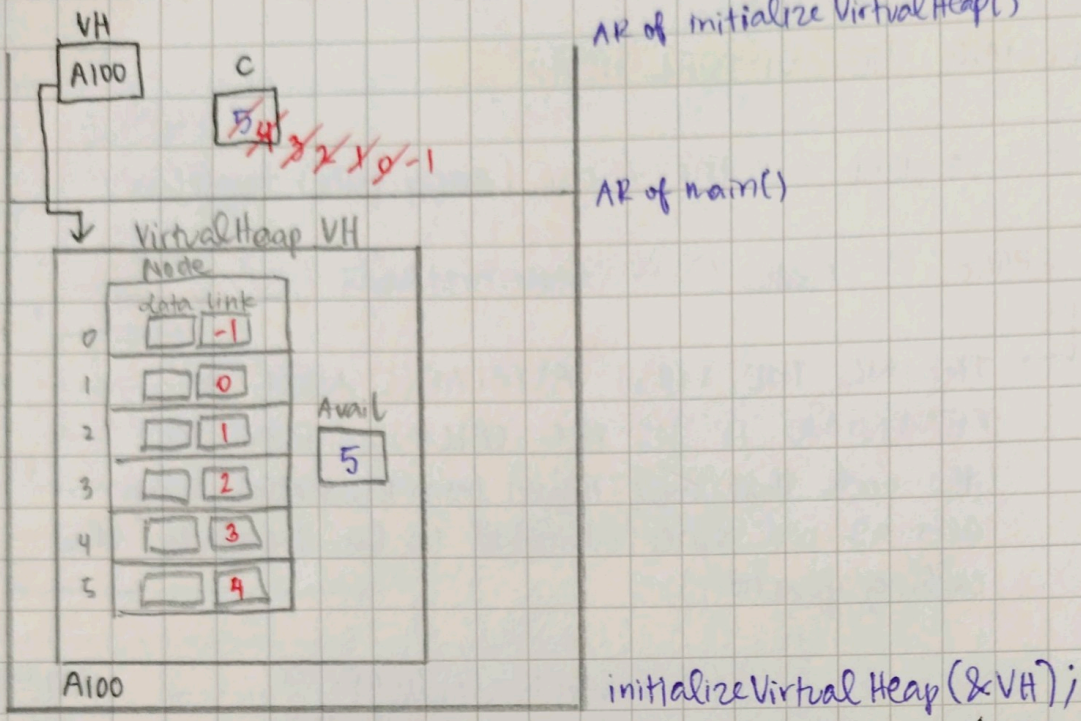
① INITIALIZING THE VIRTUAL HEAP

```
void initializeVirtualHeap (VirtualHeap *VH) { // pass by address
    int c; //initializing a counter                because we're changing values
           to represent the indexes                inside the virtual heap
           of the array
    VH->Avail = MAX-1; // assigning the index of the
                          first available node in the virtual heap
    for(c=VH->Avail; c>=0; c--){
        VH->Nodes[c].link = c-1;
    }
}
```

loop to assign values of the link portion in each array cell to link to the next available cell

initializeVirtualHeap() SIMULATION

MAX = 6



---

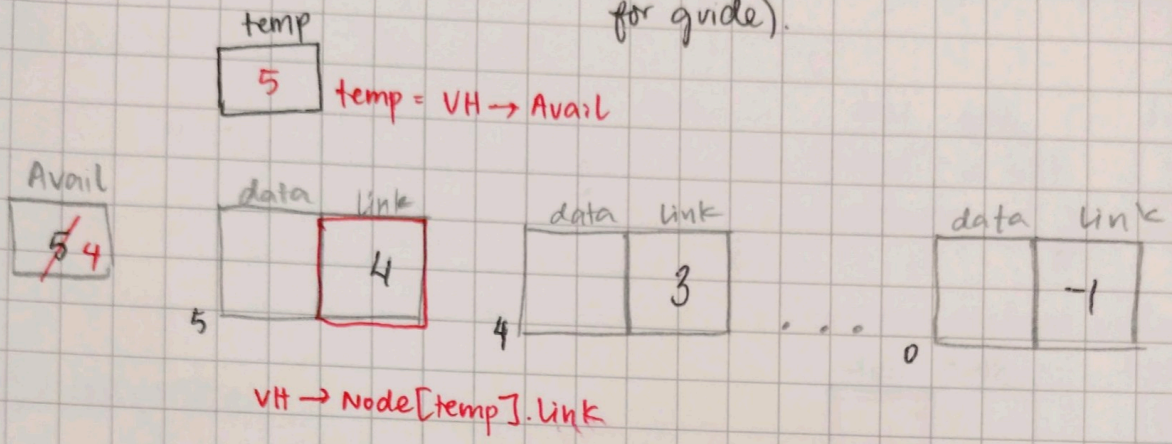
② ALLOCSPACE (DELETING FIRST AVAILABLE NODE)

↳ returns the index of the first available node, which will no longer be part of the available nodes in the virtual heap. Returns -1 (or the sentinel value if list is empty).

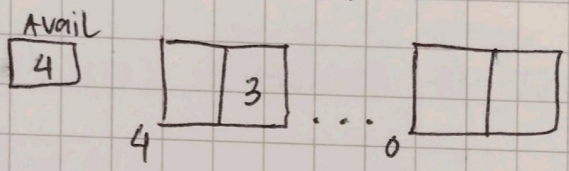
```
int allocSpace (VirtualHeap * VH) { // pass by address, obviously
    int temp = VH → Avail  // temp holds index of node to be deleted
    if (temp != -1) {
        VH → Avail = VH → Node [temp].link;
    }
    return temp; // returns the index of the deleted node,
                 // ready to be used by the calling function.
}
```

- `int` → index or sentinel value
- `if (temp != -1)` → checks if list of Avail is empty
- `VH → Avail = VH → Node [temp].link;` → let Avail point to the next available node, the index stored in the link of the temp index

allocSpace() ILLUSTRATION (not actual representation, strictly for guide).



RESULTING LIST OF AVAILABLE CELLS / NODES



---

③ DEALLOCSPACE (INSERTING A NODE / CELL BACK INTO THE VIRTUAL HEAP)

↳ given an index, it will insert a cell with the same index back into the virtual heap, so that it belongs to the list of available nodes

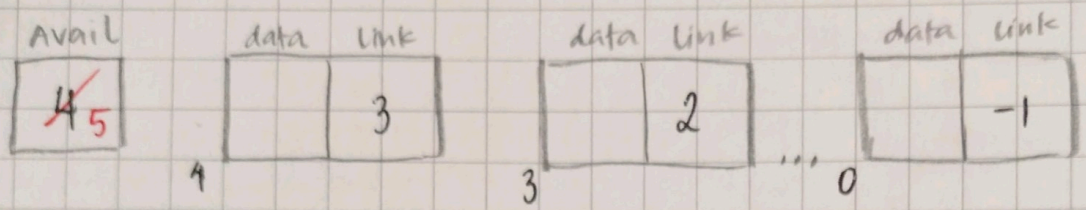
```
void deallocSpace (VirtualHeap * VH, int index) {
    VH → Node [index]. link = VH → Avail;
```

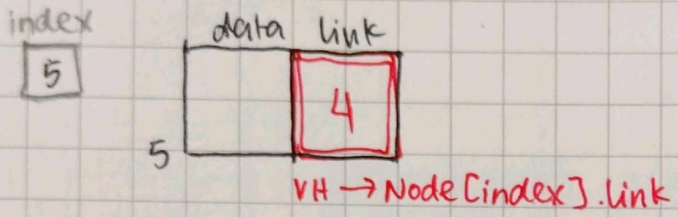
↳ connects new node (cell) to the next available node (which is the index stored in Avail)

```
    VH → Avail = index;
}
```

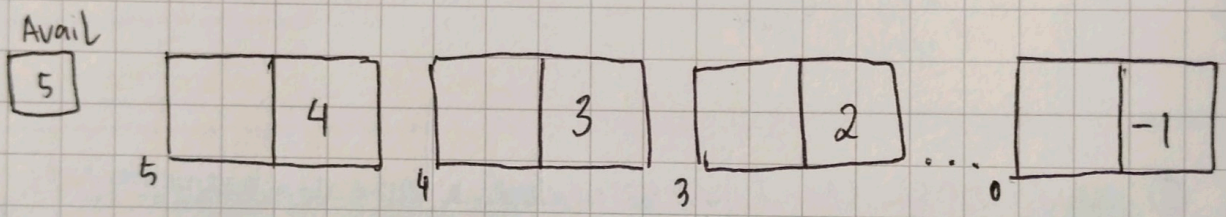
↳ Assigns the index to avail where it can be accessed as part of the list of available nodes.

deallocSpace() ILLUSTRATION





RESULTING LIST OF AVAILABLE CELLS / NODES



ADT LIST OPERATIONS - CURSOR BASED EDITION

① INITIALIZING THE LIST

↳ in cursor implementation, the list "head" is just an integer. In linked list, we initialize a list by pointing it to NULL.
In cursor based, NULL is -1 (or any sentinel value)

② INSERTION

↳ literally a translation from linked list insertion functions. PPN vs. pointer to integer

③ DELETION

↳ also LL translation. PPN vs. pointer to integer

④ DISPLAY

↳ LL translation. PN vs. integer.

ADT LIST OPERATIONS - LL. VS. CURSOR BASED

INSERT

LINKED LIST

- temp = (LIST) malloc (sizeof (struct node))
- temp → data = elem
- temp → link = *trav (pointer-pointer-node)
- *trav = temp;
  dynamically allocated node address

CURSOR BASED

- temp = allocSpace (VH)
- VH → Node [temp]. data = elem
- VH → Node [temp]. link = *trav (pointer to integer)
- *trav = temp;
  index of array

DELETE

LINKED LIST

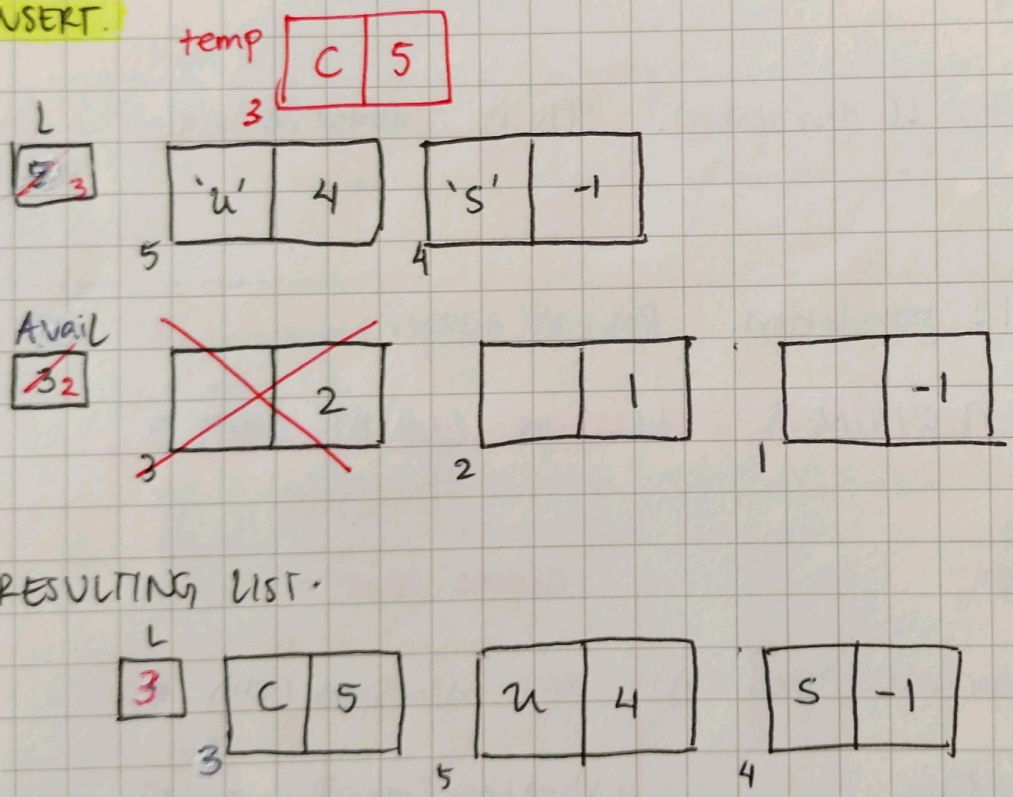
- temp = *trav
  trav = PPN
- *trav = temp → link
- free (temp)
  dynamically allocated location in memory i.e node

CURSOR-BASED

- temp = *trav
  trav = pointer to integer
- *trav = VH → head [temp]. link
- dealloc Space (VH, temp);
  integer, index of array

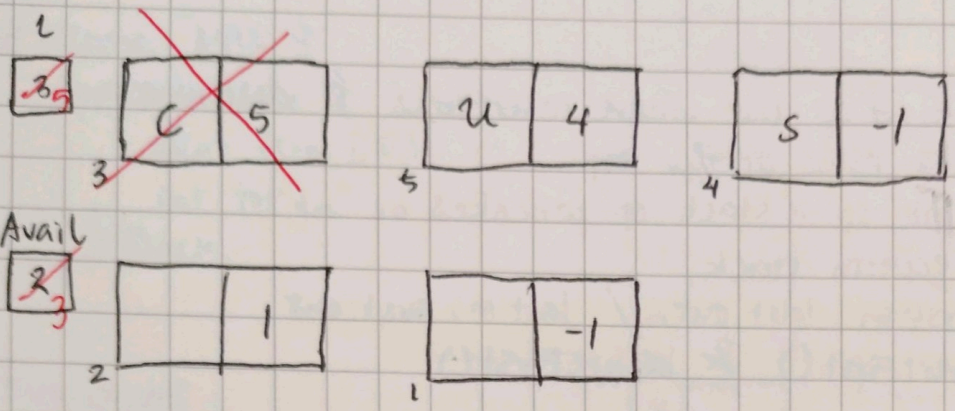
ILLUSTRATIONS

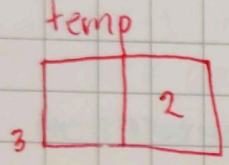
INSERT.



RESULTING LIST.

DELETE.





RESULTING LIST: