Linked Lists

ESC101: Fundamentals of Computing

Nisheeth

Are Arrays the Best?

ADVANTAGES

Allow us to create several variables of a given type

Allow us to give them very convenient names e.g. arr[i]

Can access n-th element very very easily – just use arr[n-1]

Very easy to set up, can also change size using dynamic arrays and realloc

Can have arrays of structures as well

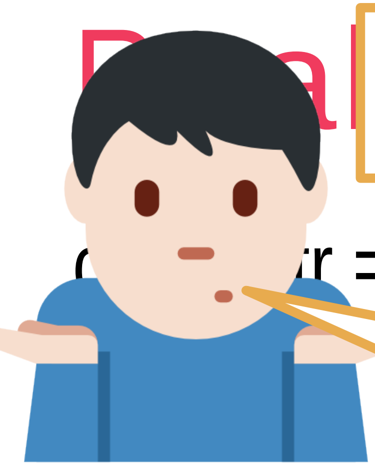
Inserting a new element at the end of the array simple

DISADVANTAGES

Inserting in the middle/beginning of array tedious - need to shift elements one location to make space – can be time consuming too!

Realloc is an expensive procedure – Mr C has to find new space for the enlarged array, allocate that space and then copy all old elements one by one

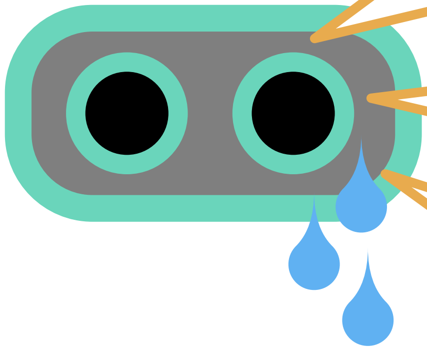
Sometimes if there is not enough memory, realloc may just fail and return a NULL pointer



```
r = (char*)malloc(15 * sizeof(char));
```

So close! We did have space for 18 characters, just not contiguous space

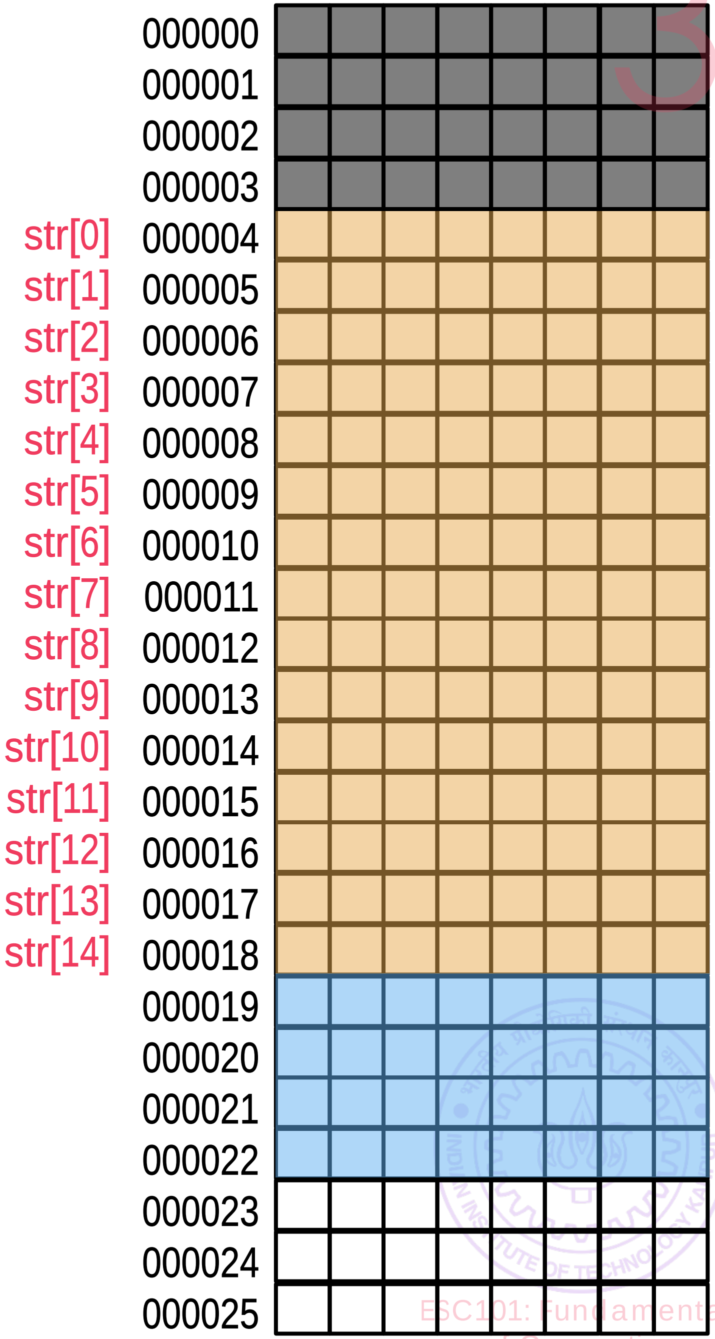
Can't we create a 3 length array and link the two arrays together?



Yes, you can – using structures and linked lists

No more memory left for me to create a char array of length 18

I only have enough space to create a char array of length 3



	000000
	000001
	000002
	000003
str[0]	000004
str[1]	000005
str[2]	000006
str[3]	000007
str[4]	000008
str[5]	000009
str[6]	000010
str[7]	000011
str[8]	000012
str[9]	000013
str[10]	000014
str[11]	000015
str[12]	000016
str[13]	000017
str[14]	000018
	000019
	000020
	000021
	000022
	000023
	000024
	000025

Linked Lists

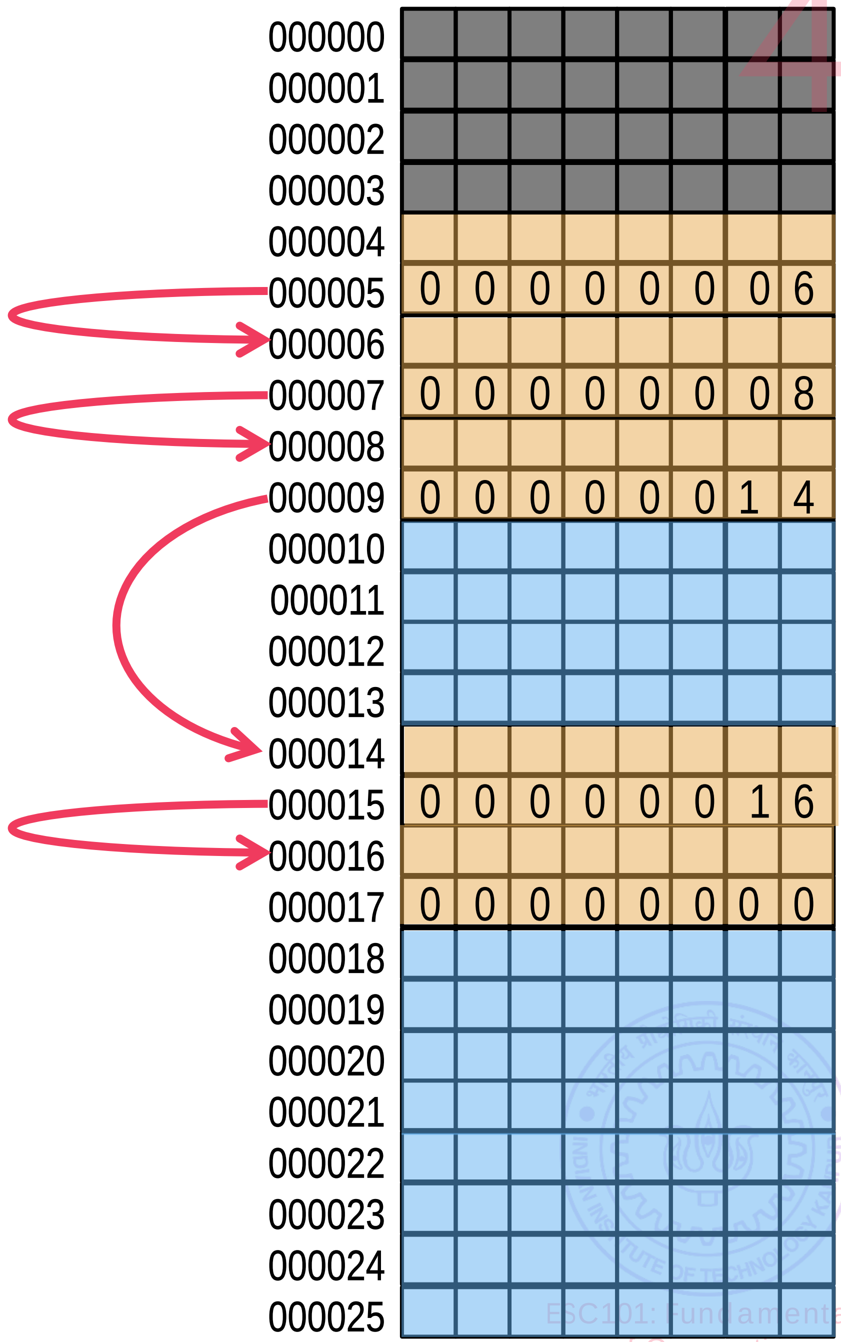
Allow for more efficient usage of space

ADVANTAGES

- Allow as many elements as you want
- Do not require contiguous space to be available – pack things better
- Can expand without calling realloc
- Inserting in the middle very simple (we'll see later)

DISADVANTAGES

- No convenient "names" for elements
- Accessing n-th element slow – require going through first n-1 elements
- Setting them up requires more work (basically linking of many struct nodes)

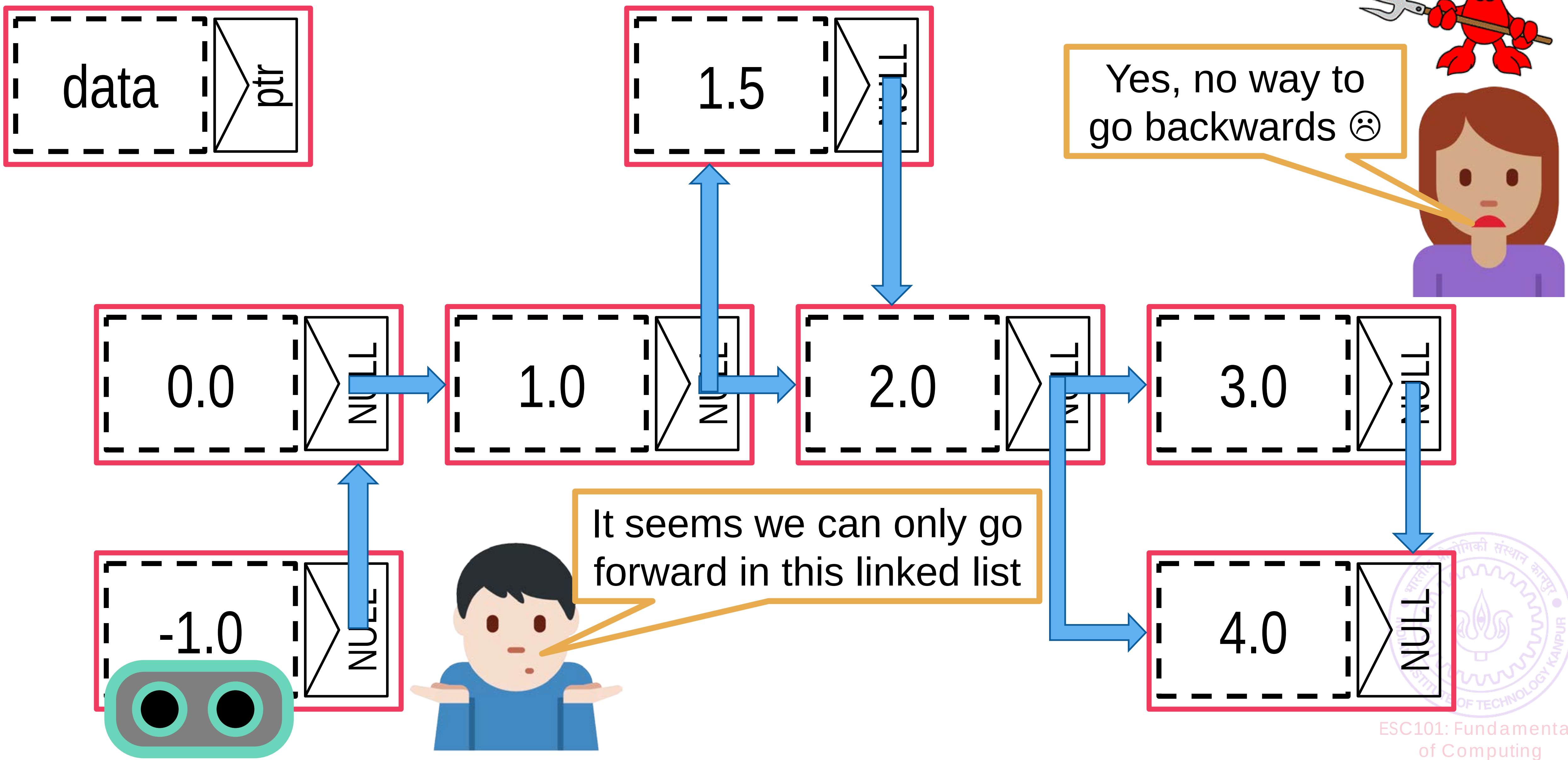


000000								
000001								
000002								
000003								
000004								
000005	0	0	0	0	0	0	0	6
000006								
000007	0	0	0	0	0	0	0	8
000008								
000009	0	0	0	0	0	0	1	4
000010								
000011								
000012								
000013								
000014								
000015	0	0	0	0	0	0	1	6
000016								
000017	0	0	0	0	0	0	0	0
000018								
000019								
000020								
000021								
000022								
000023								
000024								
000025								

A Cartoon of Linked Lists

Fear not – a simple solution

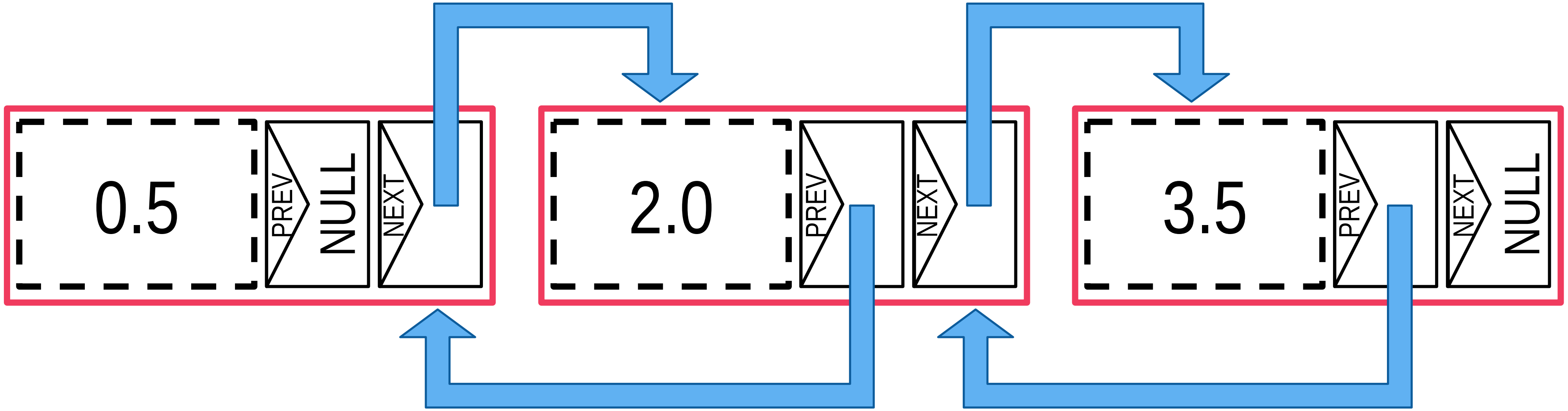
Yes, no way to go backwards ☹



It seems we can only go forward in this linked list

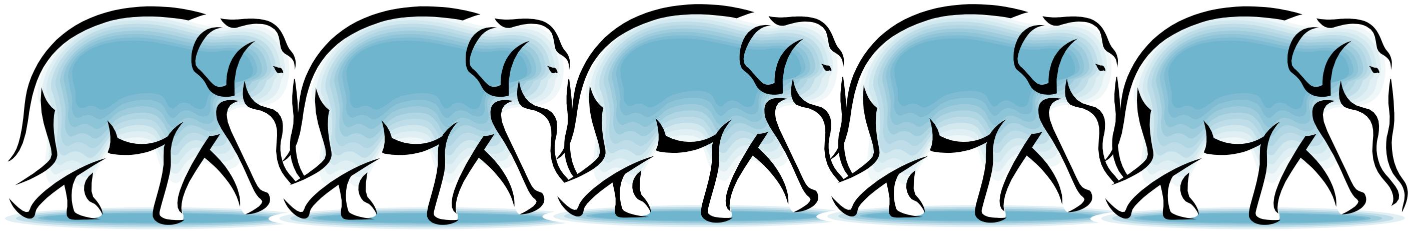
Doubly Linked Lists

Allows traversal both ways.
However more code needed



Linked List - more details..

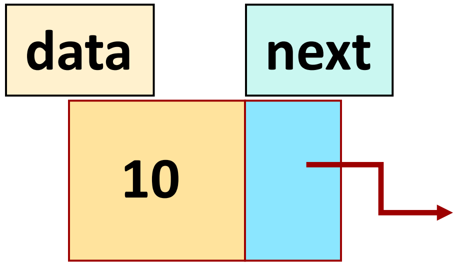
- A linear, dynamic data structure, consisting of nodes. Each node consists of two parts:
  - a “data" component, and
  - a “next" component, which is a pointer to the next node (the last node points to nothing).



- Can use a structure with to create each node of a linked list

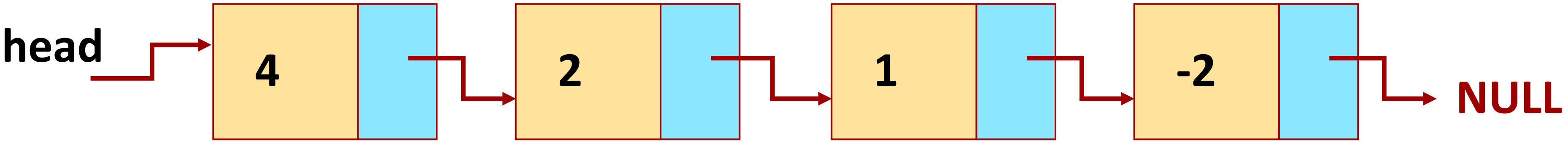
Linked List : A Self-referential structure

```
struct node {
    int data;
    struct node *next;
};
```



struct node

1. Defines struct node, used as a node (element) in the "linked list".
2. Note that the field next is of type struct node *
3. next can't be of type struct node,
   (since it will mean a recursive definition, of unknown or infinite size).

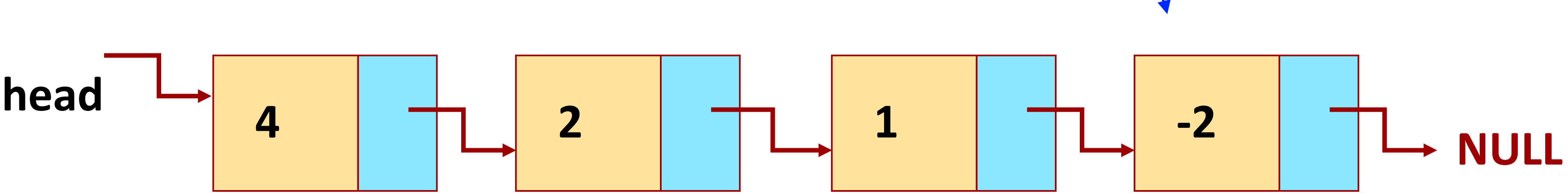


The above list has only one link (pointer) from each node, hence it is a "singly linked list".

Linked List

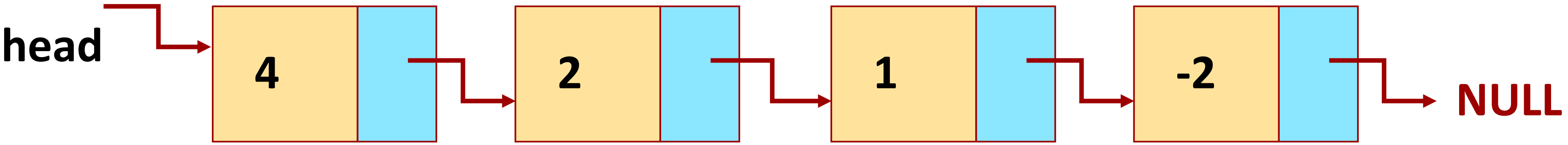
List starts at node pointed to by **head**

next field == NULL pointer indicates the last node of the list



1. The list is modeled by a variable (**head**): points to the first node of the list.
2. **head == NULL** implies empty list.
3. The next field of the **last** node is **NULL.**
4. Name **head** is just a convention – can give any name to the pointer to first node, but **head** is used most often.

Displaying/Traversing a Linked List



```
void display_list(struct node *head)
{
  struct node *cur = head;
  while (cur != NULL) {
    printf("%d ", cur->data);
    cur = cur->next;
  }
  printf("\n");
}
```

OUTPUT

```
4 2 1 -2
```

Can also use recursion (try doing it using recursion)

Creating a new node

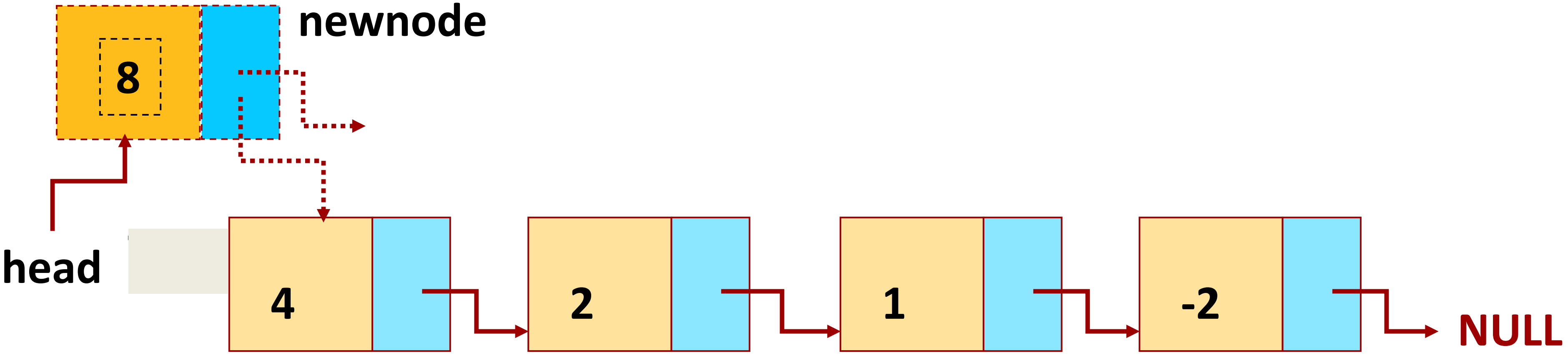
/* Allocates new node pointer and sets the data field to val, next field is NULL */

```
struct node * make_node(int val) {
    struct node *nd;
    nd = (struct node *)calloc(1, sizeof(struct node));
    nd->data = val;
    nd->next = NULL;
    return nd;
}
```

Inserting a node at the front of a linked list

Inserting at the front of the list.

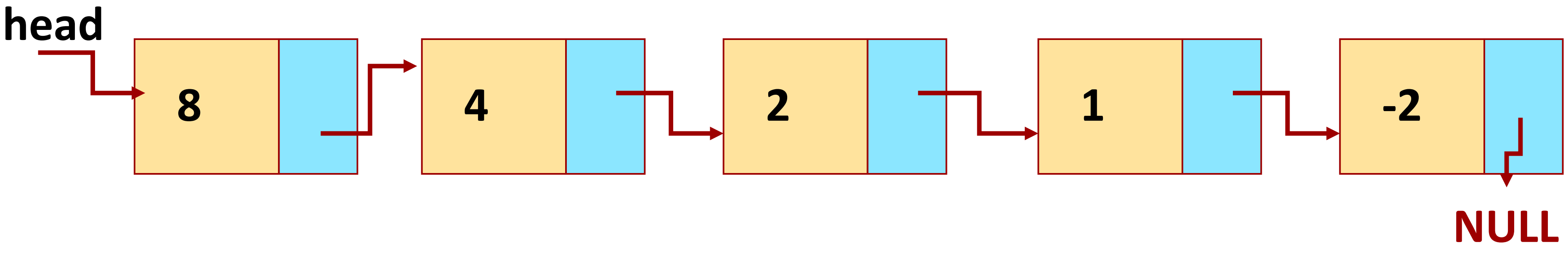
1. Create a new node of type struct node. Data field set to the value given.
2. "Add'' to the front: its next pointer points to target of head.
3. Adjust head to newnode.



Inserting a node at the front of a linked list

```
struct node *insert_front(int val, struct node *head)
{
    struct node *newnode= make_node(val);
    newnode->next = head;
    head = newnode;
    return head;
}
```

Inserts newnode at the head of the list (pointed by head).
Returns pointer to the head of new list.
Works even when list is empty, i.e. head == NULL



Let’s start with an empty list and insert in sequence -2, 1,2, 4 and 8, given by user. Final list should be as above.

```
struct node *head = NULL;
int val; scanf (“%d”, &val);
while (val != -1) {
  insert_front (val, head);
  scanf (“%d”, &val);
}                                   INPUT: -2 1 2 4 8 -1
```

Creates list in the reverse order: head points to the last element inserted.

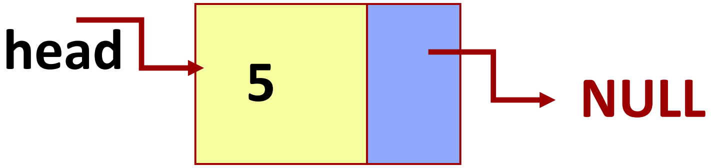
How to create list in the same order as input? Think

Generic Insertion in linked list

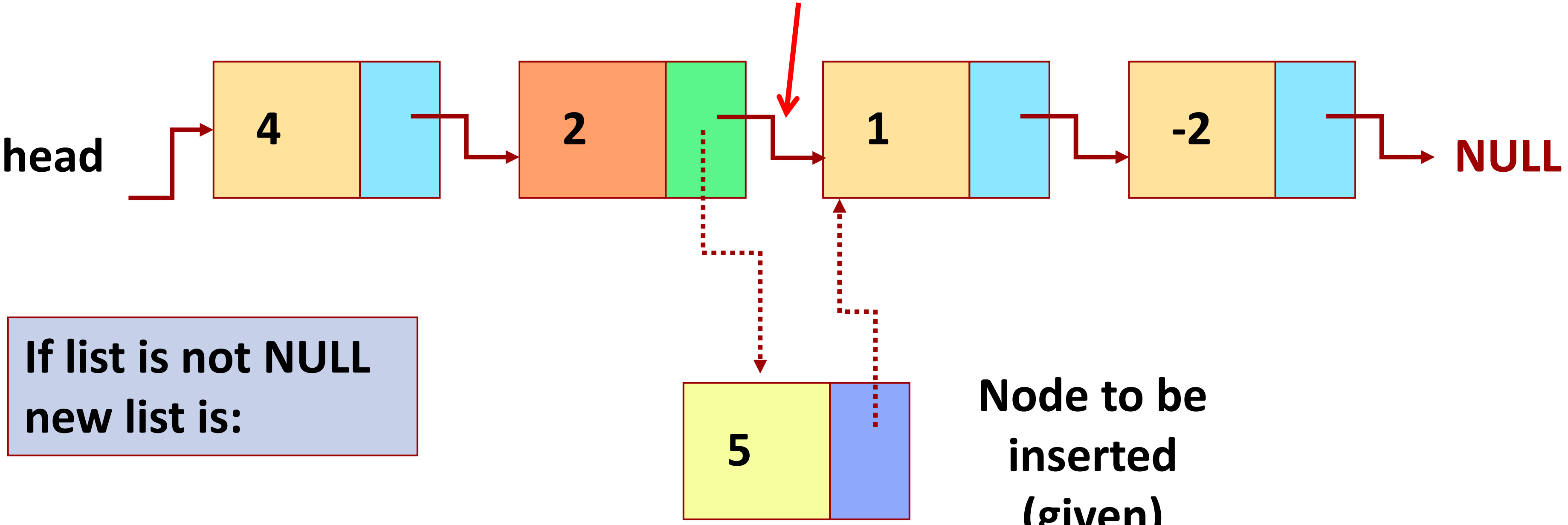
List Insertion

Given a node, insert it after a specified node in the linked list.

If list is NULL new list is:

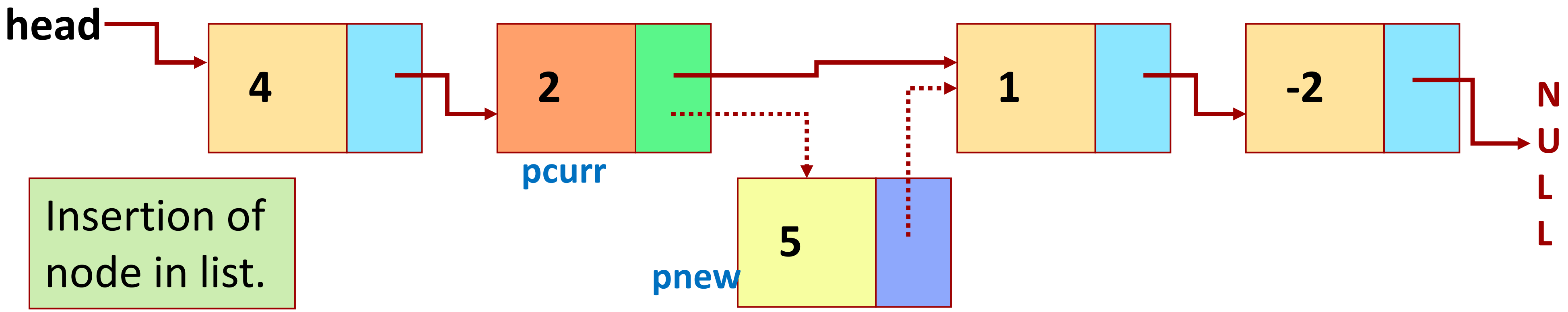


Suppose we want to insert Here



If list is not NULL new list is:

Node to be inserted (given)



Insertion of node in list.

Given

pcurr: Pointer to node after which insertion to be made

pnew: Pointer to new node to be inserted.

```
struct node *insert_after_node (struct node *pcurr, struct node *pnew) {
   if (pcurr != NULL) {
      // Order of next two statements is important
      pnew->next = pcurr->next;
      pcurr->next = pnew;
      return pcurr; // return the prev node
   }
   else return pnew; // return the new node itself
}
```