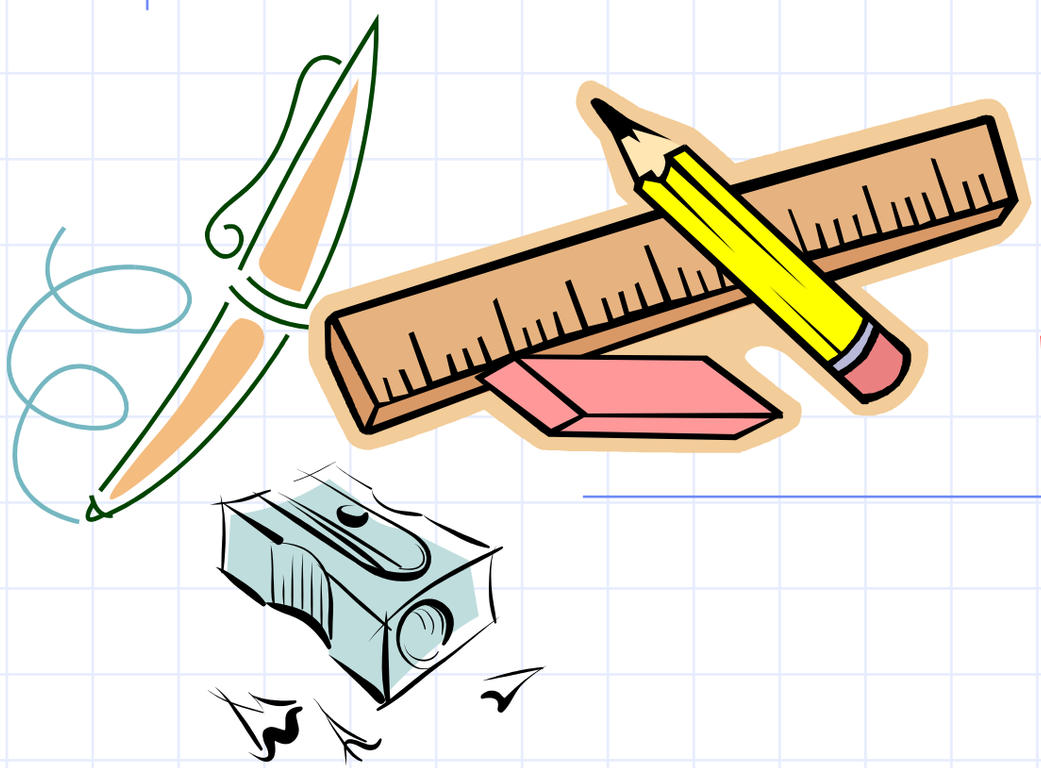
ESC101: Introduction to Computing

Structures



(Re)defining a Type - typedef

- When using a structure data type, it gets a bit cumbersome to write struct followed by the structure name every time.
- Alternatively, we can use the typedef command to set an alias (or shortcut).

```
struct point {
        int x; int y;
};
typedef struct point Point;
struct rect {
    Point leftbot;
    Point righttop;
};
```

- We can merge struct definition and typedef:

```
typedef struct point {
        int x; int y;
} Point;
```

More on typedef

- `typedef` may be used to rename any type
  - Convenience in naming
  - Clarifies purpose of the type
  - Cleaner, more readable code
  - Portability across platforms

Syntax

```
typedef Existing-Type NewName;
```

- **Existing type** is a base type or compound type
- **NewName** must be an identifier (same rules as variable/function name)

More on typedef

```
typedef char* String;
// String: a new name to char pointer
typedef int size_t; // Improved Readability
typedef struct point* PointPtr;

typedef long long int64; // Portability as
it's at least a 64-bit integer
OR
typedef long long int int64;
```

Practical Example: Revisited

- Customer information
- Struct cust_info {

```
int Account_Number;
int Account_Type;
char *Customer_Name;
char* Customer_Address;
};
```

- Customer can have more than 1 accounts
  - Want to keep multiple accounts for a customer together for easy access

Customer Information : Updated

- "Link" all the customer accounts together using a "chain"
- Struct cust_info {

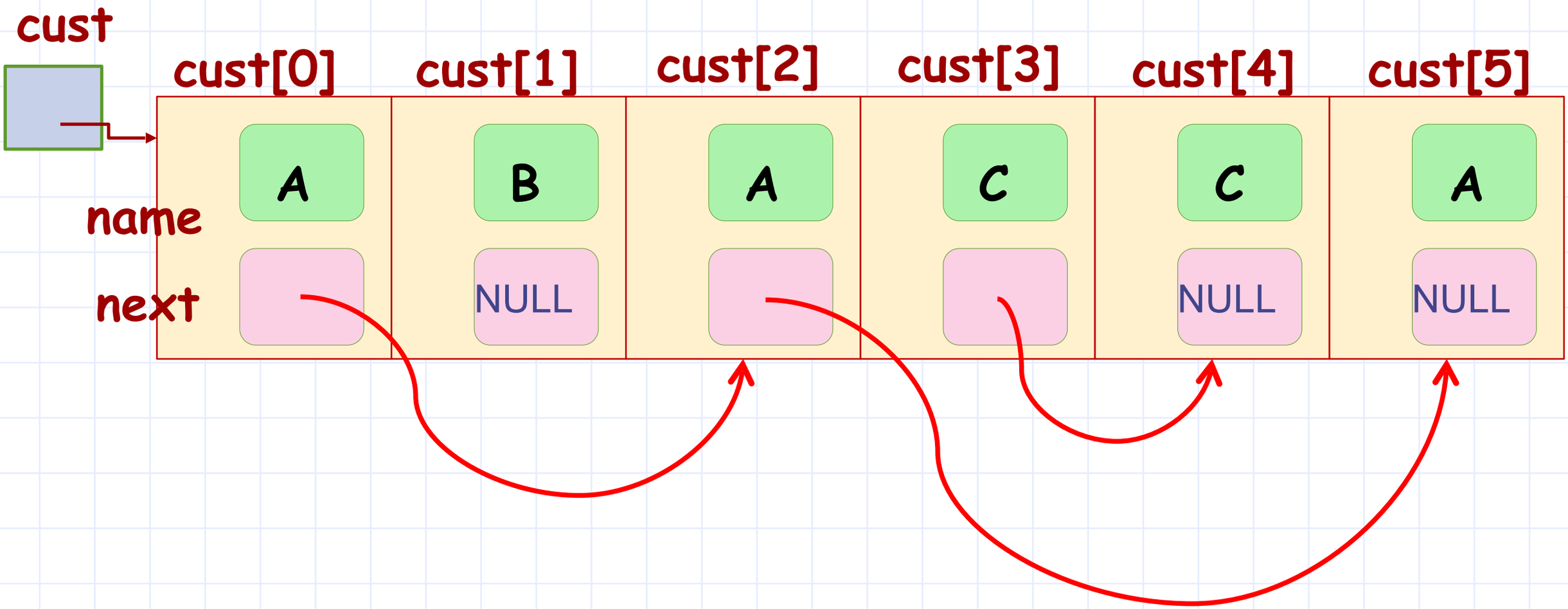
```
int Account_Number;
int Account_Type;
char *Customer_Name;
char* Customer_Address;
struct cust_info next_account;
};
```

Error: Field next_account has **incomplete type**

Customer Information : Updated

- "Link" all the customer accounts together using a "chain-of-pointers"
- Struct cust_info {

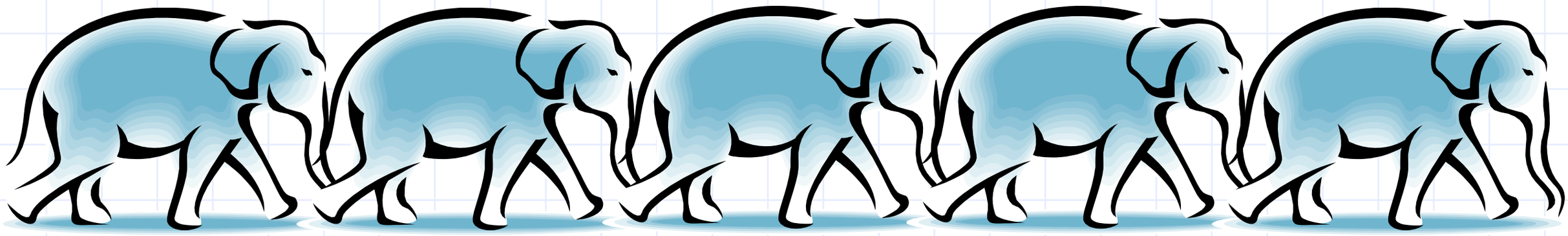
```
    int Account_Number;
    int Account_Type;
    char *Customer_Name;
    char* Customer_Address;
    struct cust_info* next_account;
};
```



cust[i].next, cust[i].next->next,
cust[i].next->next->next etc.,
when **not NULL**, point to the "other"
records of the same customer

Data Structure- Eg. Linked List

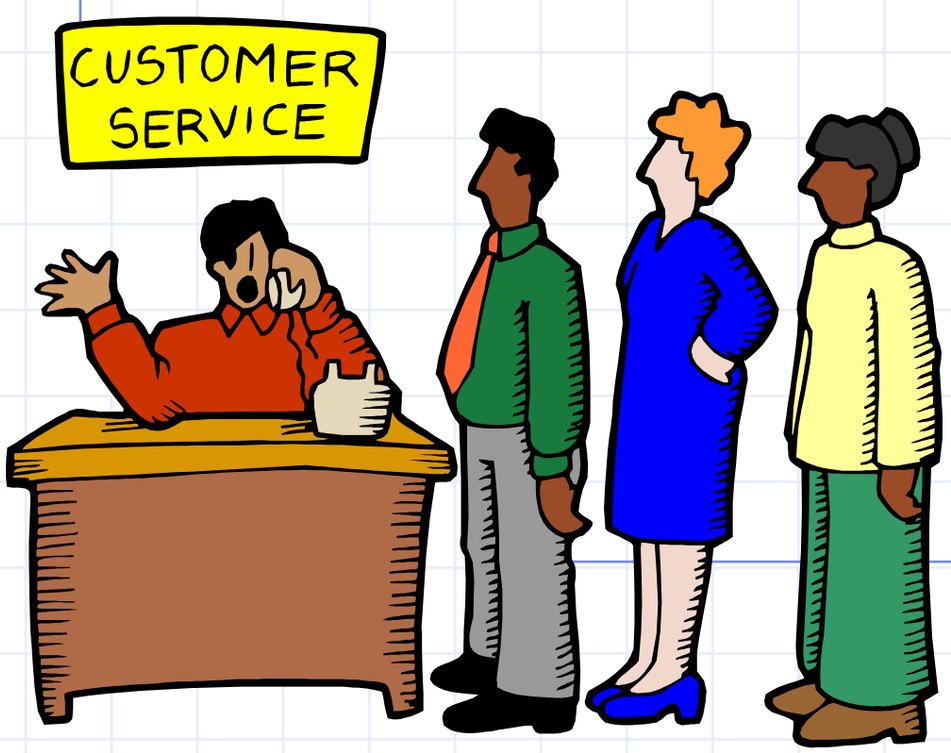
- A linear, dynamic data structure, consisting of nodes. Each node consists of two parts:
  - a "data" component, and
  - a "next" component, which is a pointer to the next node (the last node points to nothing).

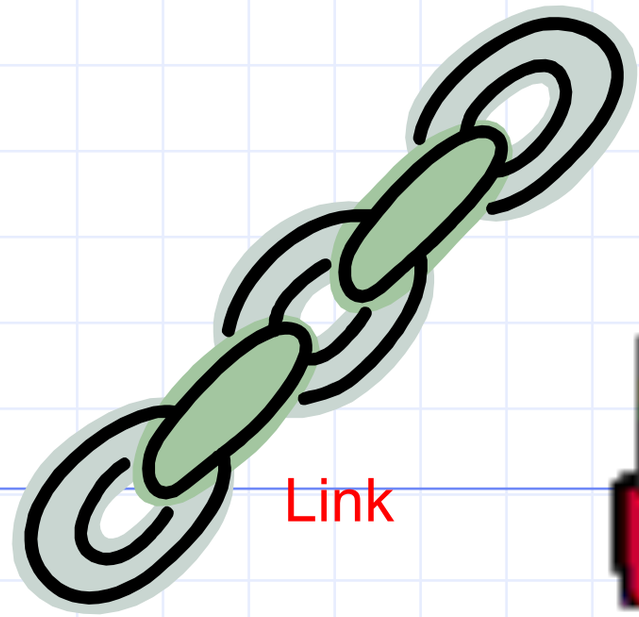


ESC101: Introduction to Computing

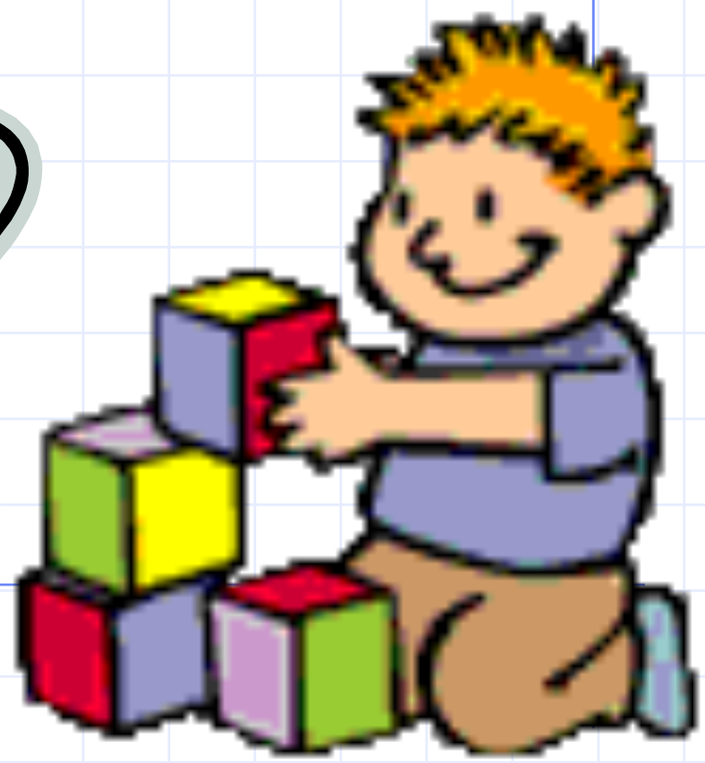
Data Structures

Queue





Link



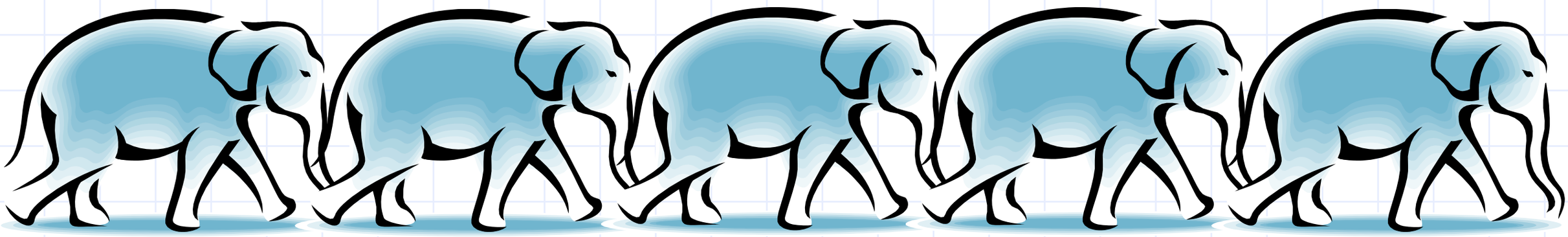
Stack

Data Structure

- What is a data structure?
- According to Wikipedia:
  - … a particular way of storing and organizing data in a computer so that it can be used efficiently...
  - … highly specialized to specific tasks.
- Examples: array, a dictionary, a set, etc.

Linked List

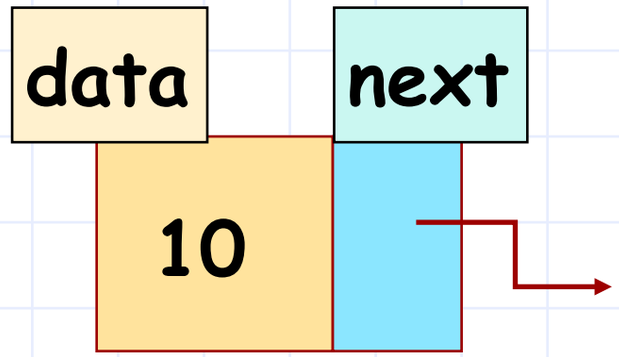
- A linear, dynamic data structure, consisting of nodes. Each node consists of two parts:
  - a "data" component, and
  - a "next" component, which is a pointer to the next node (the last node points to nothing).



Linked List : A Self-referential structure

Example:

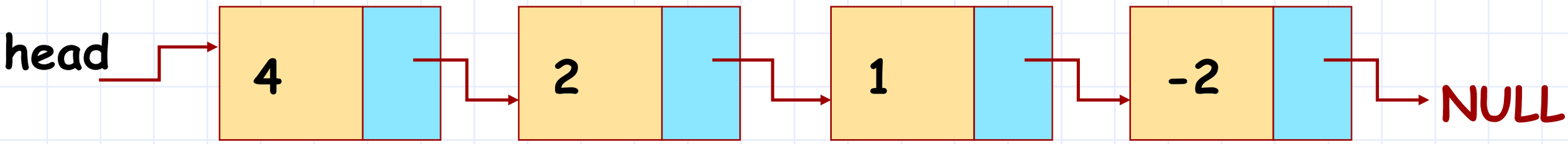
```
struct node {
      int data;
      struct node *next;
};
```



struct node

1. Defines the structure **struct node**, which will be used as a node in a "linked list" of nodes.
2. Note that the field **next** is of type **struct node ***
3. If it was of type struct node, it could not be permitted (recursive definition of unknown or infinite size)

An example of a (singly) linked list structure is:

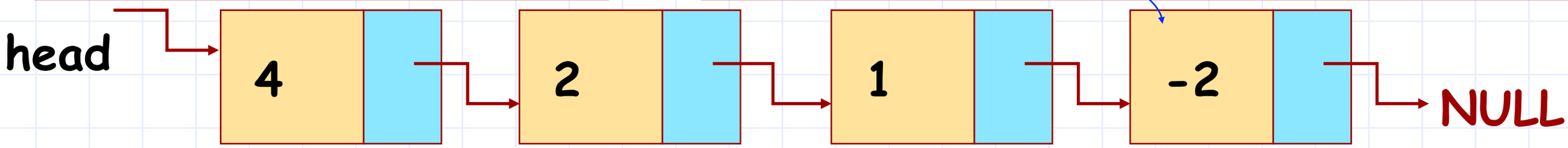


There is only one link (pointer) from each node, hence, it is also called "**singly linked list**".

Linked Lists

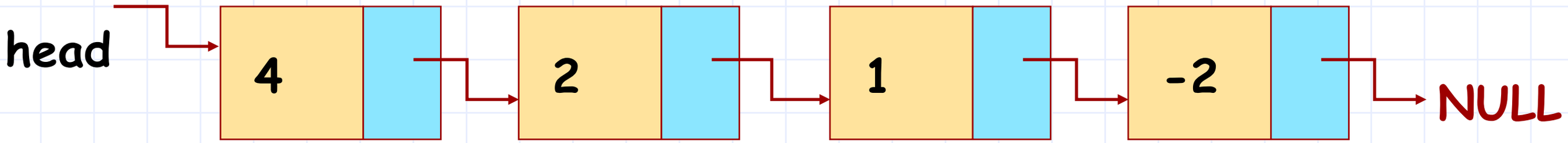
List starts at node pointed to by head

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



1. The list is modeled by a variable called **head** that 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. Note that the name **head** is just a convention – it is possible to give any name to the pointer to first node, but **head** is used most often.

Displaying a Linked List

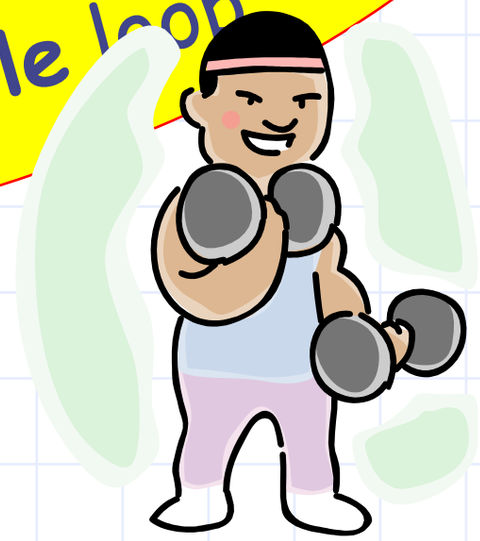



```
void display_list(struct node *head)
{
  struct node *cur = head;
  while (cur != NULL) {
    printf("%d ", cur->data);
    cur = cur->next;
  }
  printf("\n");
}
```

OUTPUT

4 2 1 -2

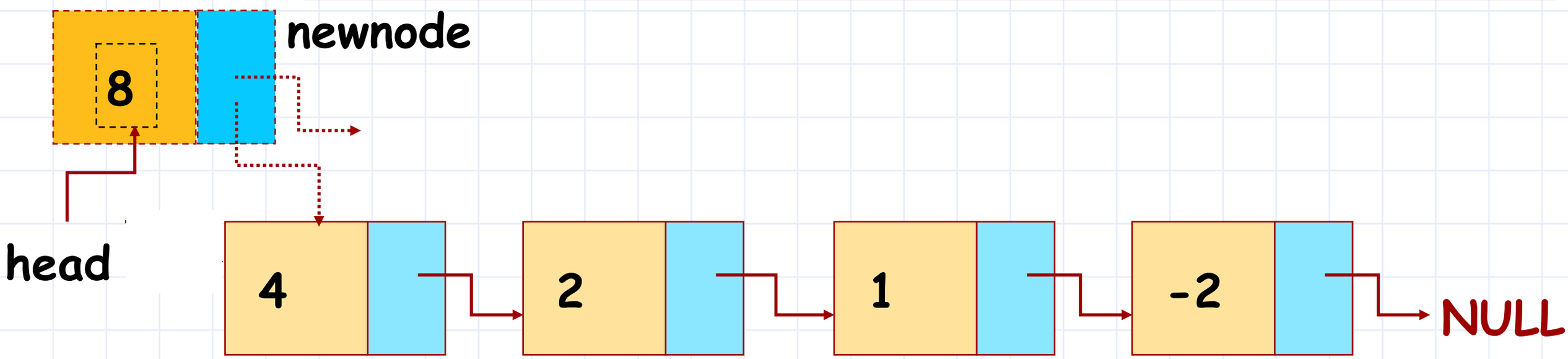
Exercise: Rewrite the code using for loop instead of while loop



Insert at Front

Inserting at the front of the list.

1. Create a new node of type struct node. Set its data field to the value given.
2. "Add" it to the front of the list:
   Make its next pointer point to target of head.
3. Adjust head correctly to point to newnode.

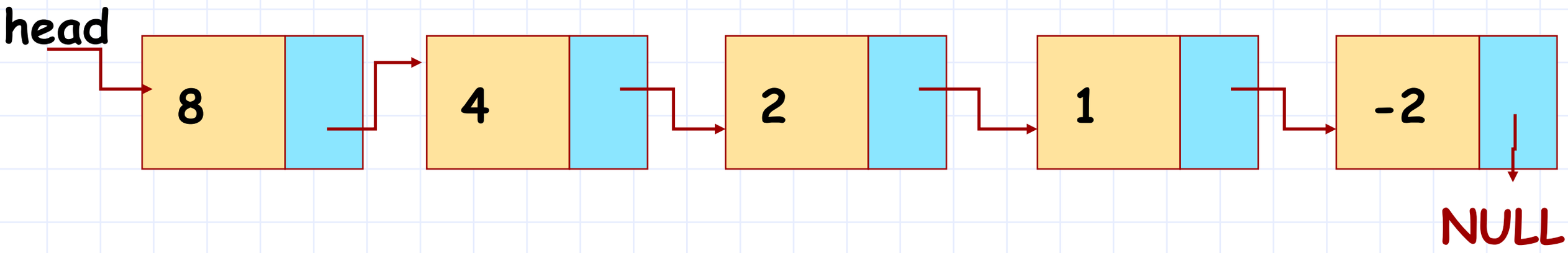


```
struct node * make_node(int val) {
      struct node *nd;
      nd = calloc(1, sizeof(struct node));
      nd->data = val;
      return nd;
}
```

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

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

```
/* Inserts a node with data field val at the head
    of the list currently pointed to by head.
   Returns pointer to the head of new list.
   Works even when the original list is empty,
     i.e. head == NULL */
```



Suppose we want to start with an empty list and insert in sequence -2, 1,2, 4 and 8. The following code gives an example. Final list should be as above.

```
struct node *head =
 insert_front (8,
    insert_front( 4,
         insert_front(2,
              insert_front(1,
                   insert_front(-2,NULL ) ) ) ) );
```

This creates the list from the last node outwards. The innermost call to insert_front gives the first node created.

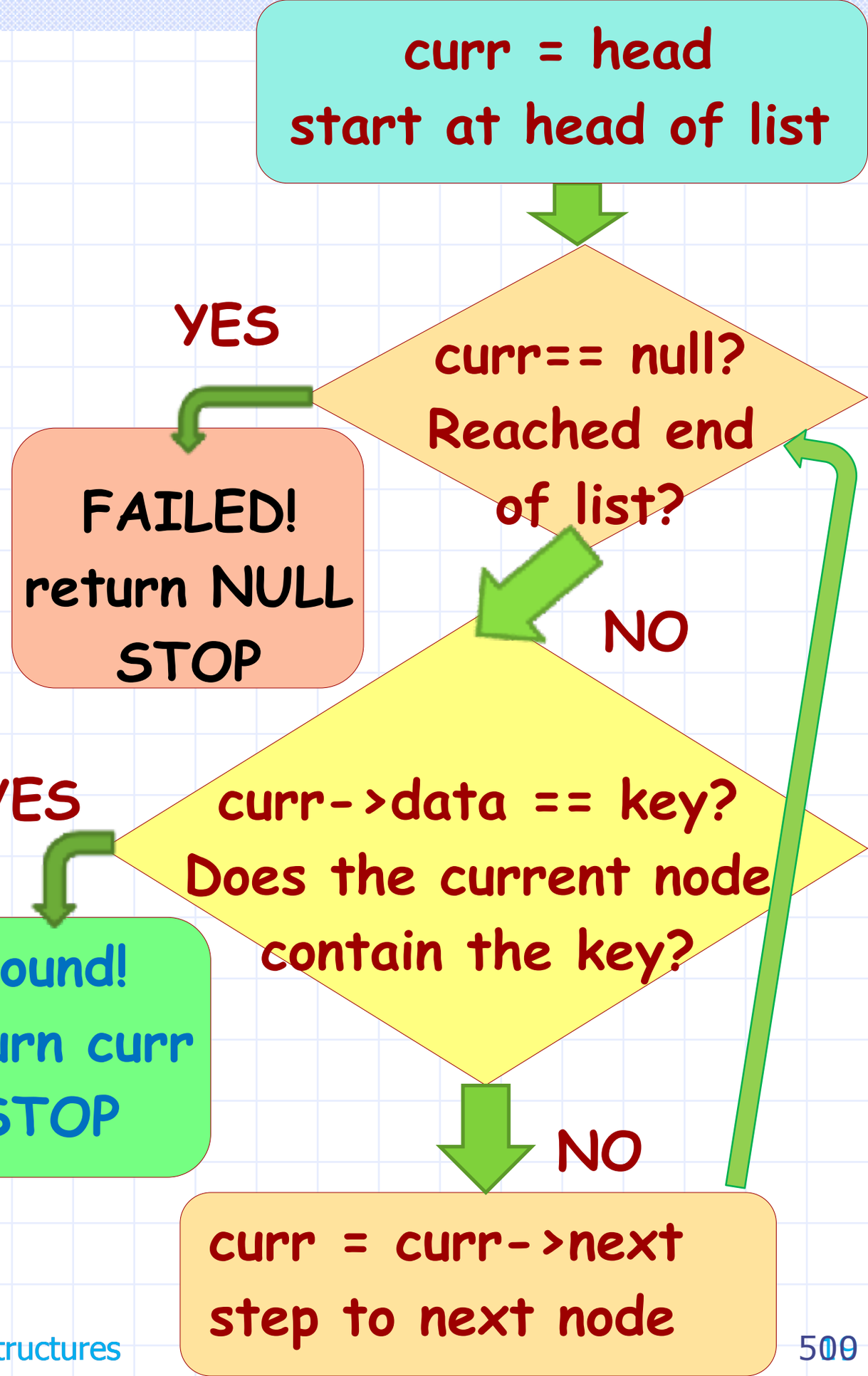
Searching in LL

```
struct node *search(
    struct node *head, int key) {
    struct node *curr = head;
    while
     (curr && curr->data != key)
                curr = curr->next;

    return curr;
}
```

search for key in a list pointed to by head.
Return pointer to the node found or else return NULL.

Disadvantage:
Sequential access only.

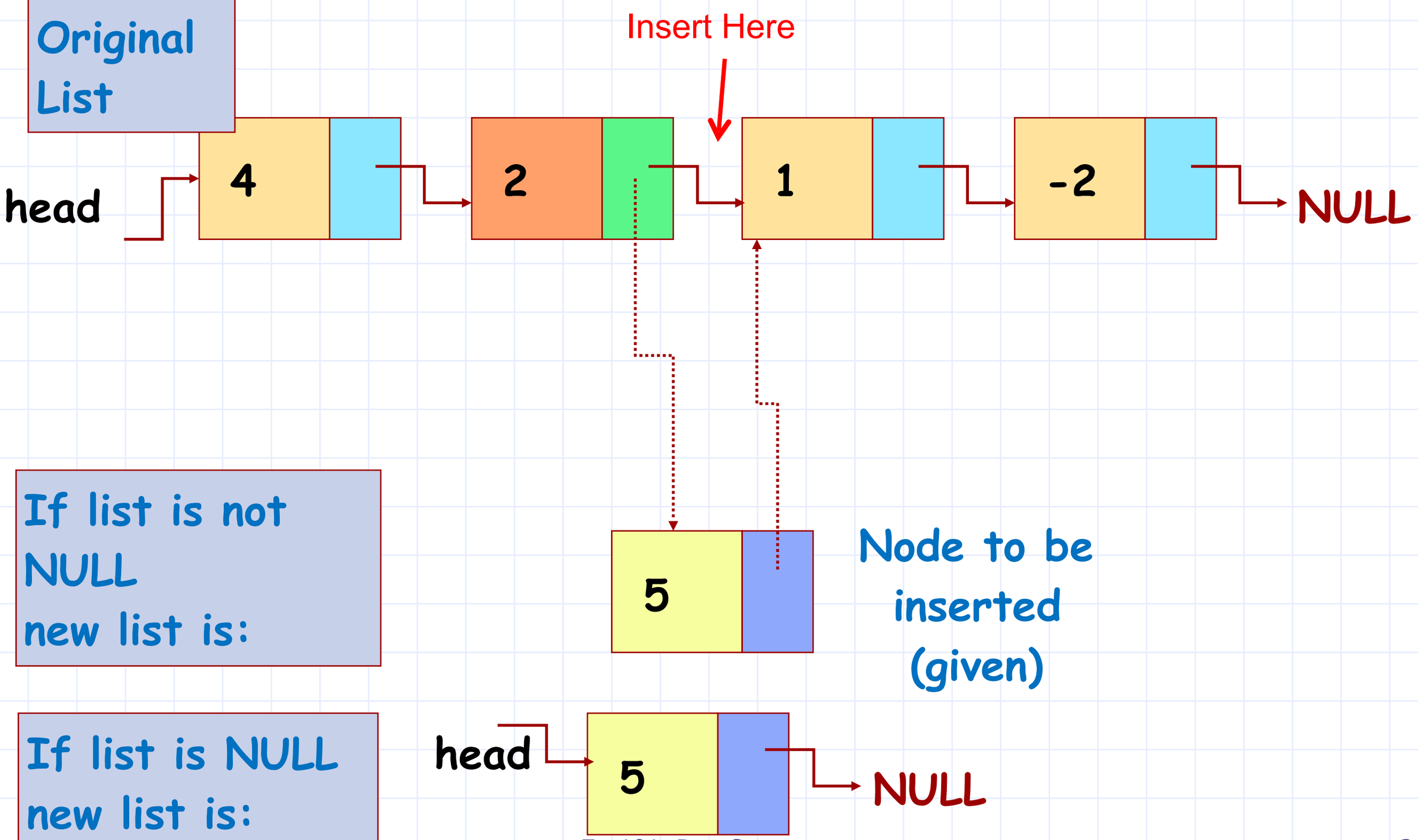


Insertion in linked list

List Insertion

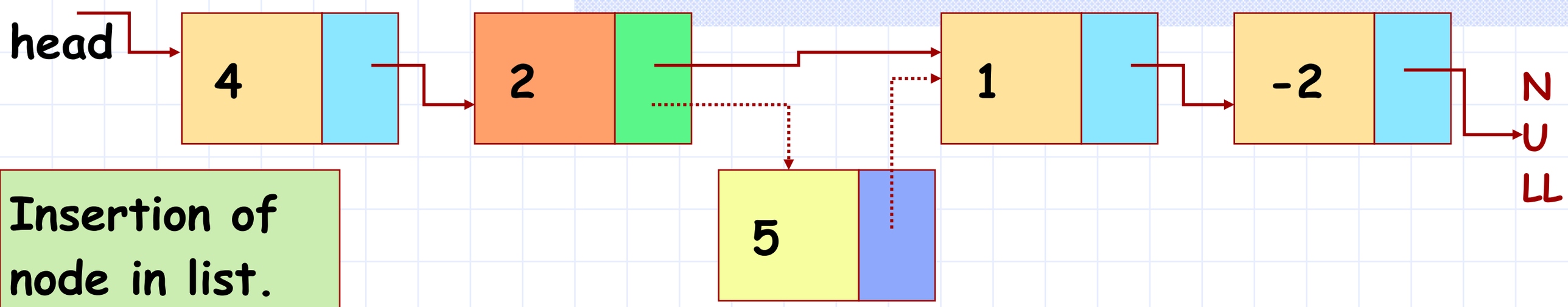
Given a node, insert it after a specified node in the linked list.

Original List



If list is not NULL
new list is:

If list is NULL
new list is:



Insertion of node in list.

Given **pcurr:** Pointer to node after which insertion is 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 stmts is important
        pnew->next = pcurr->next;
        pcurr->next = pnew;
        return pcurr; // return the prev node
    }
    else return pnew; // return the new node itself
}
```

Recap: typedef in C

- Repetitive to keep writing the type struct node for parameters, variables etc.
- C allows naming types— the typedef statement.

Defines a new type Listnode as struct node *

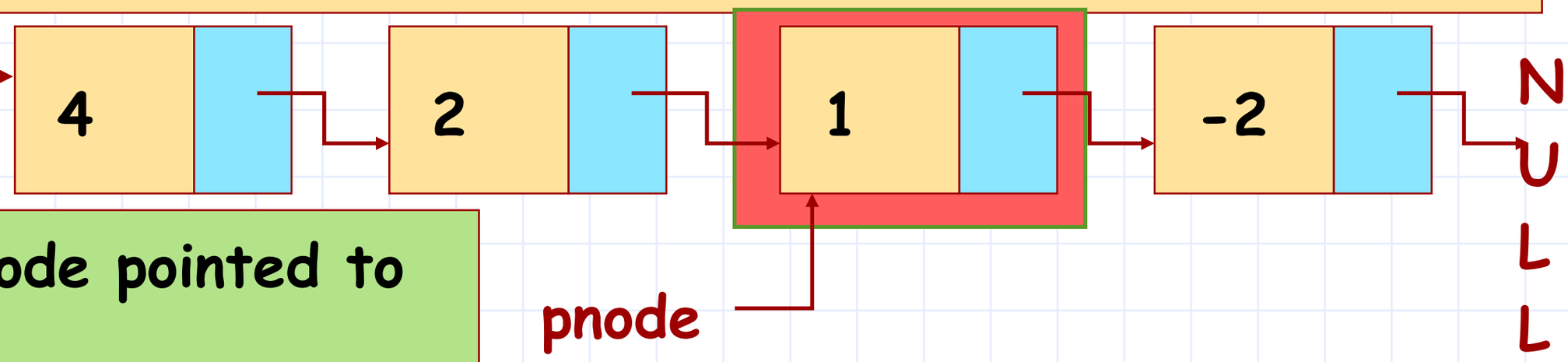
```
typedef struct node * Listnode;
```

Listnode is a type. It can now be used in place of struct node * for variables, parameters, etc..

```
Listnode head, curr;
 /* search in list for key */
Listnode search(Listnode list, int key);
/* insert the listnode n in front of listnode list */
Listnode insert_front(Listnode list, Listnode n);
 /* insert the listnode n after the listnode curr */
Listnode insert_after(Listnode curr, Listnode n);
```

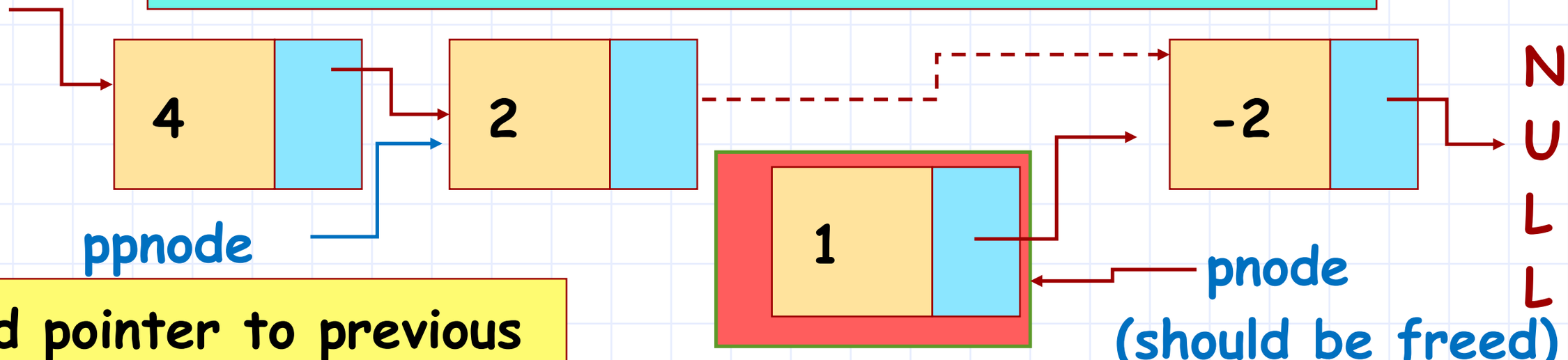
Deletion in linked list

Given a pointer to a node pnode that has to be deleted. Can we delete the node?



E.g, delete node pointed to by pnode

After deletion, we want the following state



Need pointer to previous node to pnode to adjust pointers.

call free() to release storage for deleted node.

prototype

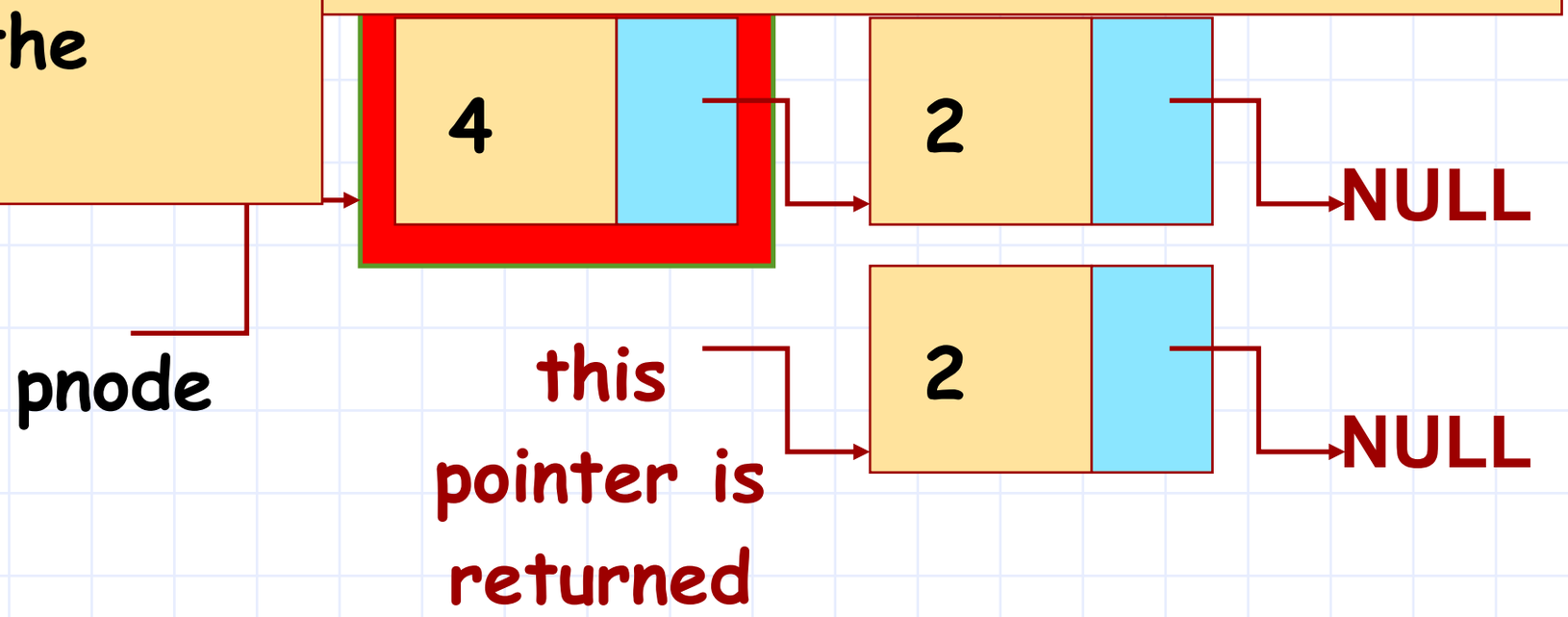
delete(Listnode pnode, Listnode ppnode)

```
Listnode delete(Listnode pnode, Listnode ppnode) {
    Listnode t;
    if (ppnode)
                ppnode->next = pnode->next;
    t = ppnode ? ppnode : pnode->next;
    free (pnode);
    return t;
}
```

Delete the node pointed to by pnode. ppnode is pointer to the node previous to pnode in the list, if such a node exists, otherwise it is NULL.

Function returns ppnode if it is non-null, else returns the successor of pnode.

The case when pnode is the head of a list. Then ppnode == NULL.



Why linked lists

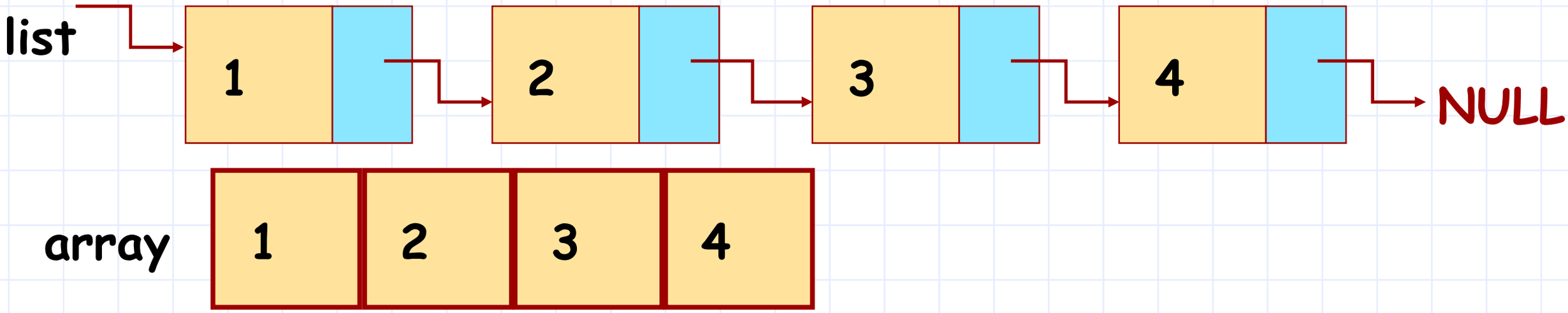
➢ The same numbers can be represented in an array. So, where is the advantage?

1. Insertion and deletion are inexpensive, only a few "pointer changes".
2. To insert an element at position k in array:
   create space in position k by shifting elements in positions k or higher one to the right.
3. To delete element in position k in array:
   compact array by shifting elements in positions k or higher one to the left.

Disadvantages of Linked List

➢ Direct access to kth position in a list is expensive (time proportional to k) but is fast in arrays (constant time).

Linked Lists: the pros and the cons



Operation	Singly Linked List	Arrays
Arbitrary Searching.	sequential search (linear)	sequential search (linear)
Sorted structure.	Still sequential search. Cannot take advantage.	Binary search possible (logarithmic)
Insert key after a given point in structure.	Very quick (constant number of operations)	Shift all array elements at insertion index and later one position to right. Make room, then insert. (linear time)

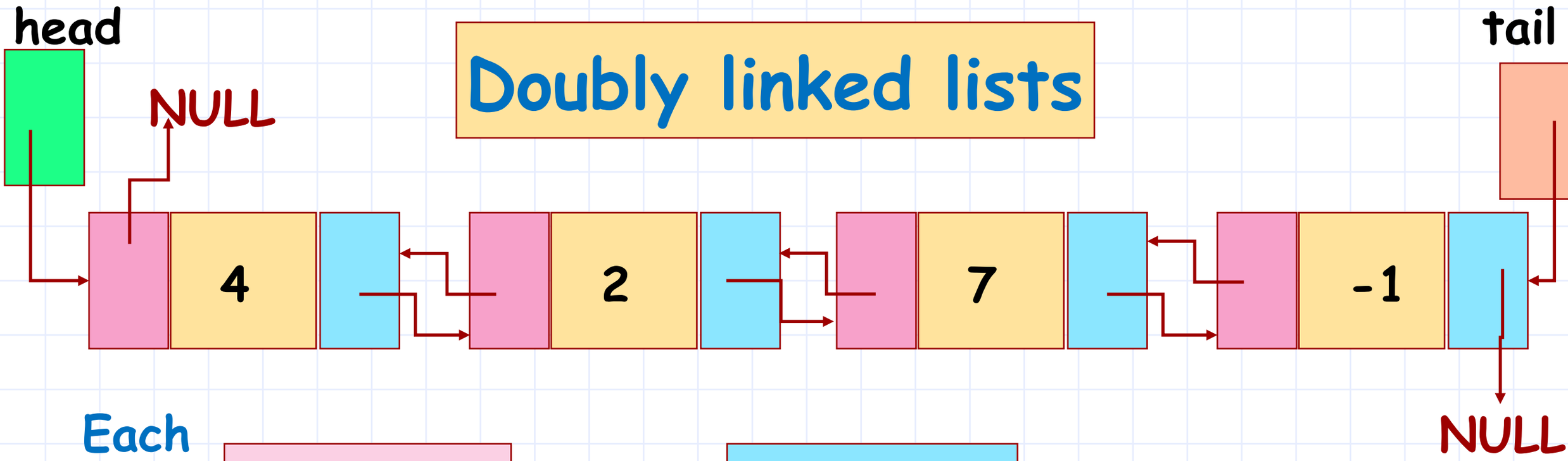
Singly Linked Lists

Operations on a linked list. For each operation, we are given a pointer to a current node in the list.

Operation	Singly Linked List
Find next node	Follow next field
Find previous node	Can't do !!
Insert before a node	Can't do !!
Insert in front	Easy, since there is a pointer to head.

Principal Inadequacy: Navigation is one-way only from a node to the next node.

Doubly linked lists



Each node has 3 fields

(i) pointer to previous node

(ii) data

(iii) pointer to next node

Defining node of Doubly linked list and the Dllist itself.

```
struct dlnode {
        int data;
        struct dlnode *next;
        struct dlnode *prev;
};
typedef struct dlnode *Ndptr;
```

```
struct dlList {
    Ndptr head;/*first node */
    Ndptr tail; /* last node */
};
typedef struct dlList *DlList;
```

Exercise

- Write a program to read in two polynomials and add them.
- Input

  1st Poly terms consisting of *e* exponent and *c* coefficient as integers in descending order -1 -1 indicating end of input

  2nd Poly terms consisting of *e* exponent and *c* coefficient as integers in descending order -1 -1 indicating end of input

Example Input (In descending order)

2 2 1 2 0 1 -1 -1

4 2 3 1 -1 -1

Output (In ascending order)

0 1 1 2 2 2 3 1 4 2

```
#include <stdio.h>
#include <stdlib.h>

struct term
{
        int exp;
        int coeff;
        struct term * next;
};

struct term *make_term(int exp, int coeff)
{
        struct term *t = (struct term *) calloc(sizeof(struct term),1);
        t->exp = exp; t->coeff = coeff;
        t->next = NULL;
        return t;
}

void print_poly(struct term *p)
{
        while(p)
        {
                printf("%d %d ",p->exp, p->coeff);
                p = p->next;
        }
}
```

```
void free_poly(struct term *p)
{
        struct term *t;
        while(p)
        {
                t = p;
                p = p->next;
                free(t);
        }
}
```

```
int main()
{
	struct term *p1=NULL, *p2=NULL;
	struct term *curr;
	int exp,coeff;
	scanf("%d %d",&exp,&coeff);
	while(exp!=-1 && coeff !=-1)
	{
		curr = make_term(exp, coeff);
		curr->next = p1;
		p1 = curr;
		scanf("%d %d",&exp, &coeff);
	}
	scanf("%d %d",&exp,&coeff);
	while(exp!=-1 && coeff !=-1)
	{
		curr = make_term(exp, coeff);
		curr->next = p2;
		p2 = curr;
		scanf("%d %d",&exp, &coeff);
	}
	print_poly(p1); printf("\n");
	print_poly(p2); printf("\n");
	polyadd( p1, p2);

	free_poly(p1);
	free_poly(p2);
	return 0;
}
```

```
void polyadd( struct term *p1, struct term *p2)
{
	while( p1 && p2 )
	{
		if(p1->exp == p2->exp)
		{
			printf("%d %d ",p1->exp, p1->coeff+p2->coeff);
			p1 = p1->next; p2 = p2->next;
		}
		else if(p1->exp > p2->exp)
		{
			printf("%d %d ",p2->exp, p2->coeff);
			p2 = p2->next;
		}
		else {
			printf("%d %d ",p1->exp, p1->coeff);
			p1 = p1->next;
		}
	}
	while( p1 ) {
		printf("%d %d ",p1->exp, p1->coeff);
		p1 = p1->next;
	}
	while( p2 ) {
		printf("%d %d ",p2->exp, p2->coeff);
		p2 = p2->next;
	}
}
```