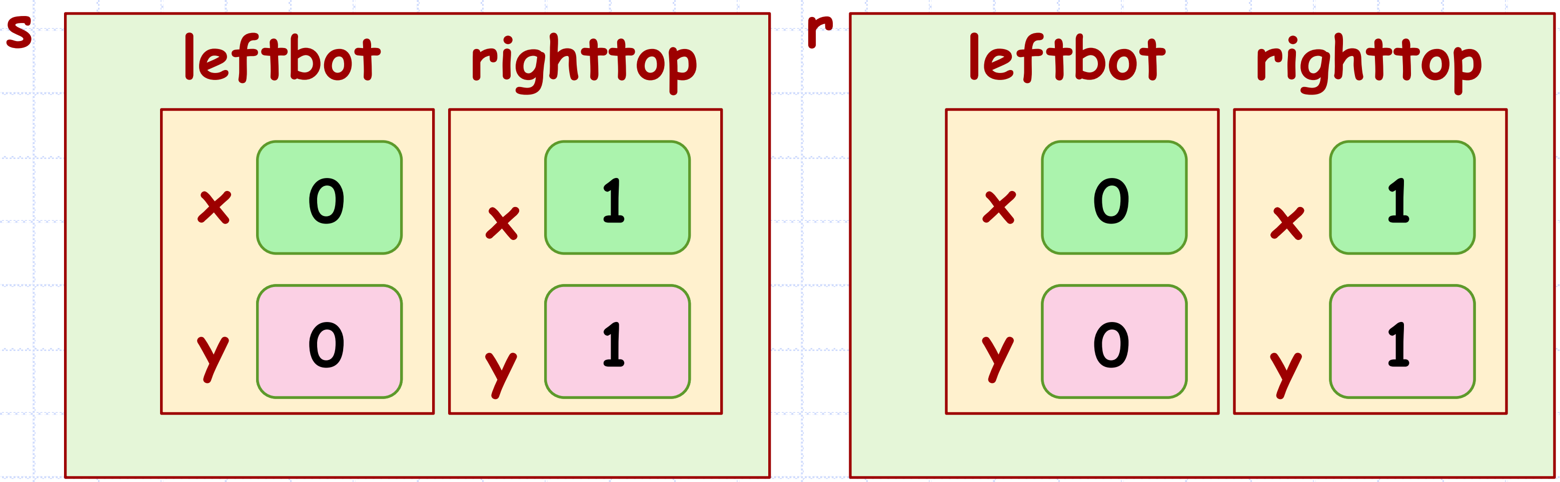
Assigning structure variables

```
struct rect r,s;
r.leftbot.x = 0;
r.leftbot.y = 0;
r.righttop.x = 1;
r.righttop.y = 1;
s=r;
```

1. We can assign a structure variable to another structure variable
2. The statement `s=r;` does this.
3. Structures are *assignable* variables, unlike arrays!
4. Structure name is *not* a pointer, unlike arrays.



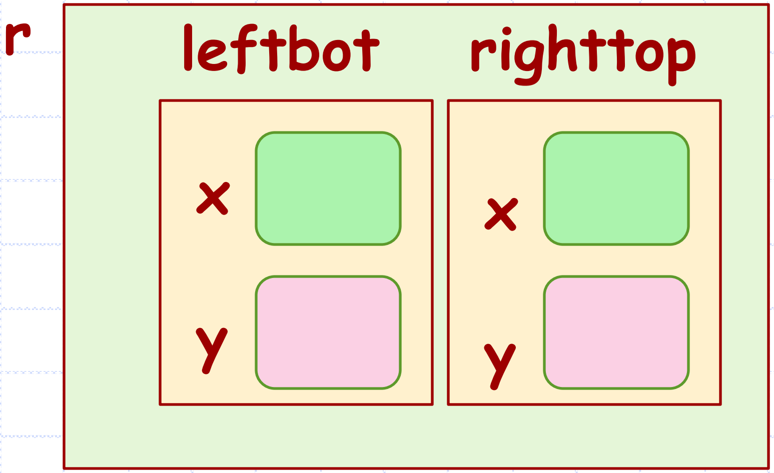
After the assignment

Passing structures..?

```
struct rect { struct point leftbot;
              struct point righttop; };
int area(struct rect r) {
   return
      (r.righttop.x – r.leftbot.x) *
      (r.righttop.y – r.leftbot.y);
}
void fun() {
   struct rect r1 ={{0,0}, {1,1}};
   area(r1);
}
```

We can pass structures as parameters, and return structures from functions, like the basic types int, char, double etc..

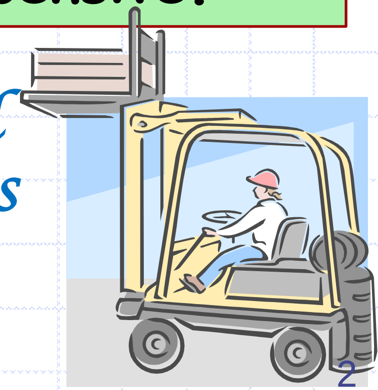
But is it efficient to pass structures or to return structures?



Usually NO. E.g., to pass struct rect as parameter, 4 integers are copied. This is expensive.

Same for returning structures

So what should be done to pass structures to functions?



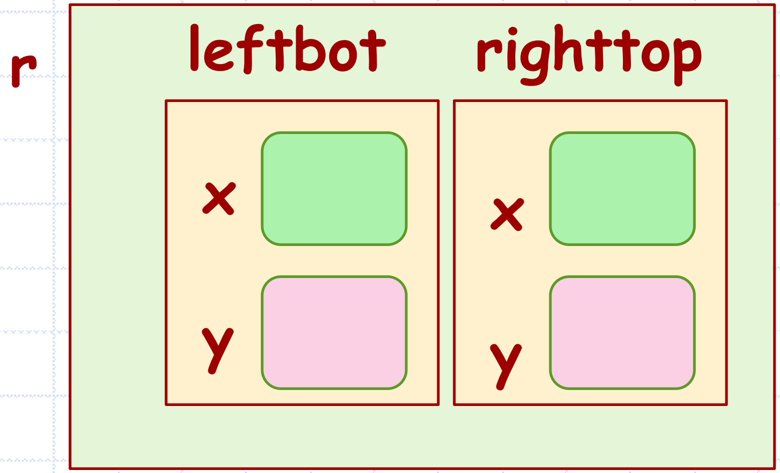
Passing structures..?

```
struct rect { struct point leftbot;
              struct point righttop;};
int area(struct rect *pr) {
  return
 ((*pr).righttop.x – (*pr).leftbot.x) *
 ((*pr).righttop.y – (*pr).leftbot.y);
}
void fun() {
   struct rect r ={{0,0}, {1,1}};
   area (&r);
 }
```

Instead of passing structures, pass pointers to structures.

area() uses a pointer to struct rect pr as a parameter, instead of struct rect itself.

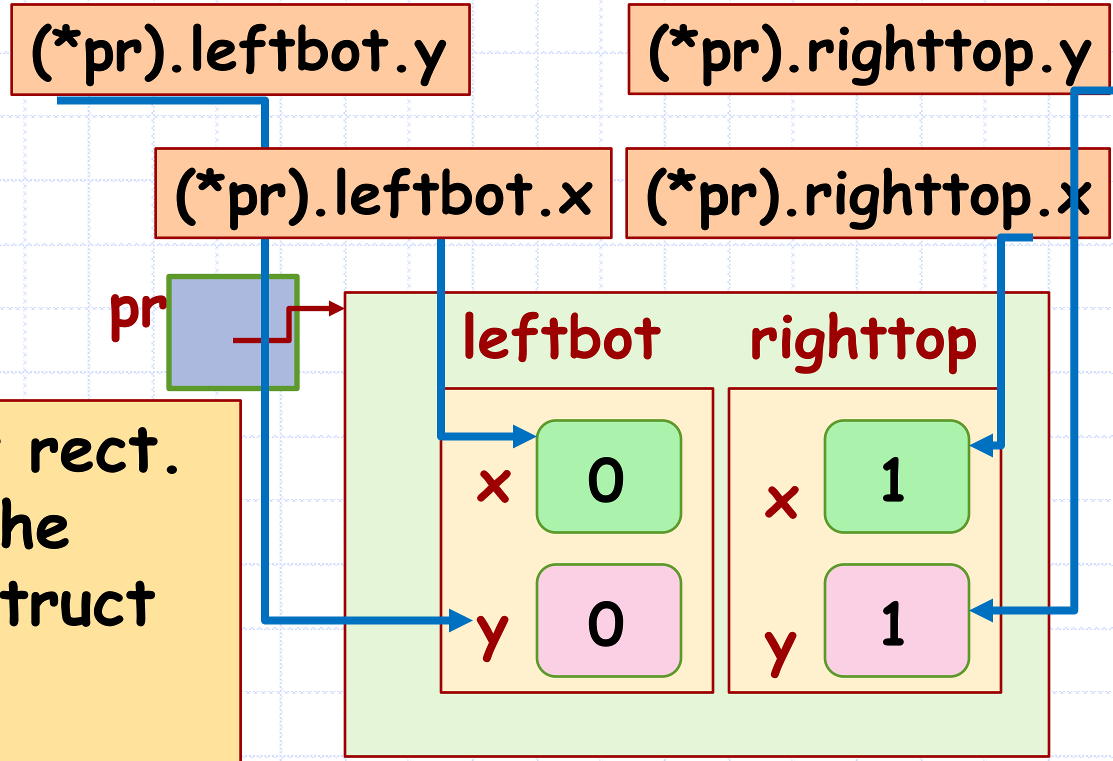
Now only one pointer is passed instead of a large struct.



Same for returning structures

Structure Pointers

```
struct point {
      int x; int y;};
struct rect {
   struct point leftbot;
   struct point righttop;
};
struct rect *pr;
```

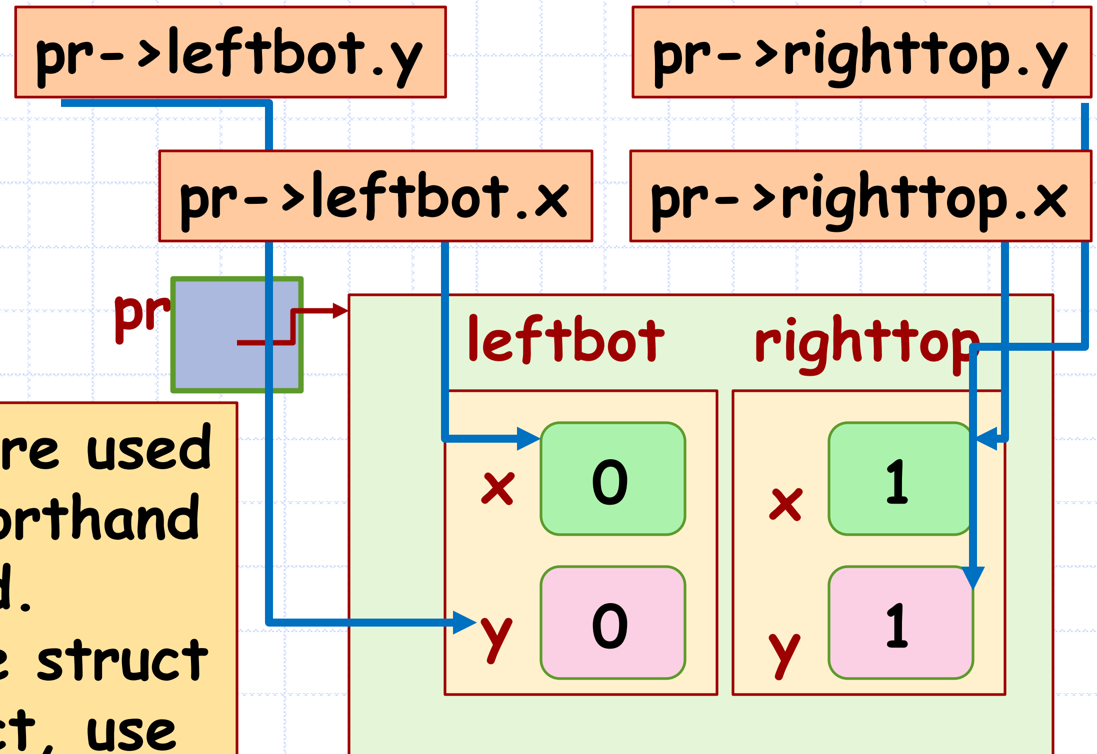


1. pr is pointer to struct rect.
2. To access a field of the struct pointed to by struct rect, use
   (*pr).leftbot
   (*pr).righttop
3. Bracketing (*pr) is essential here. * has lower precedence than .
4. To access the x field of leftbot, use (*pr).leftbot.x

Addressing fields via the structure's pointer

```
struct point {
      int x; int y;};
struct rect {
   struct point leftbot;
   struct point righttop;
};
struct rect *pr;
```

Structure Pointers



1. Pointers to structures are used so frequently that a shorthand notation (->) is provided.
2. To access a field of the struct pointed to by struct rect, use
   pr->leftbot
3. -> is one operator. To access the x field of leftbot, use
   pr->leftbot.x
3. -> and . have same precedence and are left-associative. Equivalent to (pr->leftbot).x

pr->leftbot is equivalent to (*pr).leftbot

Addressing fields via the structure's pointer (shorthand)

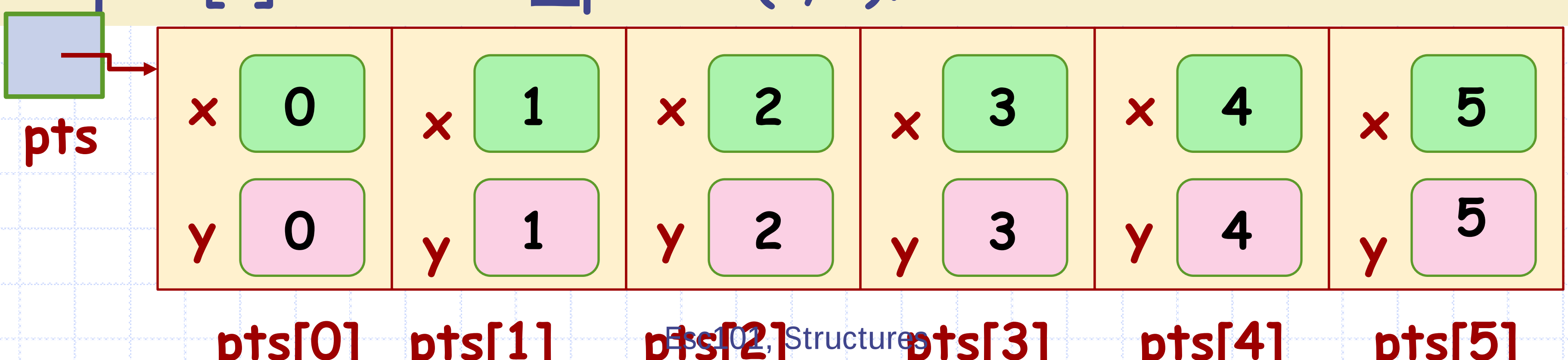
Passing struct to functions

- When a **struct** is passed directly, it is passed by copying its contents
  - Any changes made inside the called function are lost on return
  - This is same as that for simple variables
- When a **struct** is passed using pointer,
  - Change made to the contents using pointer dereference are visible outside the called function

Dynamic Allocation of struct

- Similar to other data types
- sizeof(…) works for struct-s too

```
struct point* pts;
int i;
pts = (struct point*) malloc(6 * sizeof(struct point));
for (i = 0; i < 6; i++)
    pts[i] = make_point(i, i);
```



(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;
bitmap Signature_scan; // user defined type bitmap
```

};

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

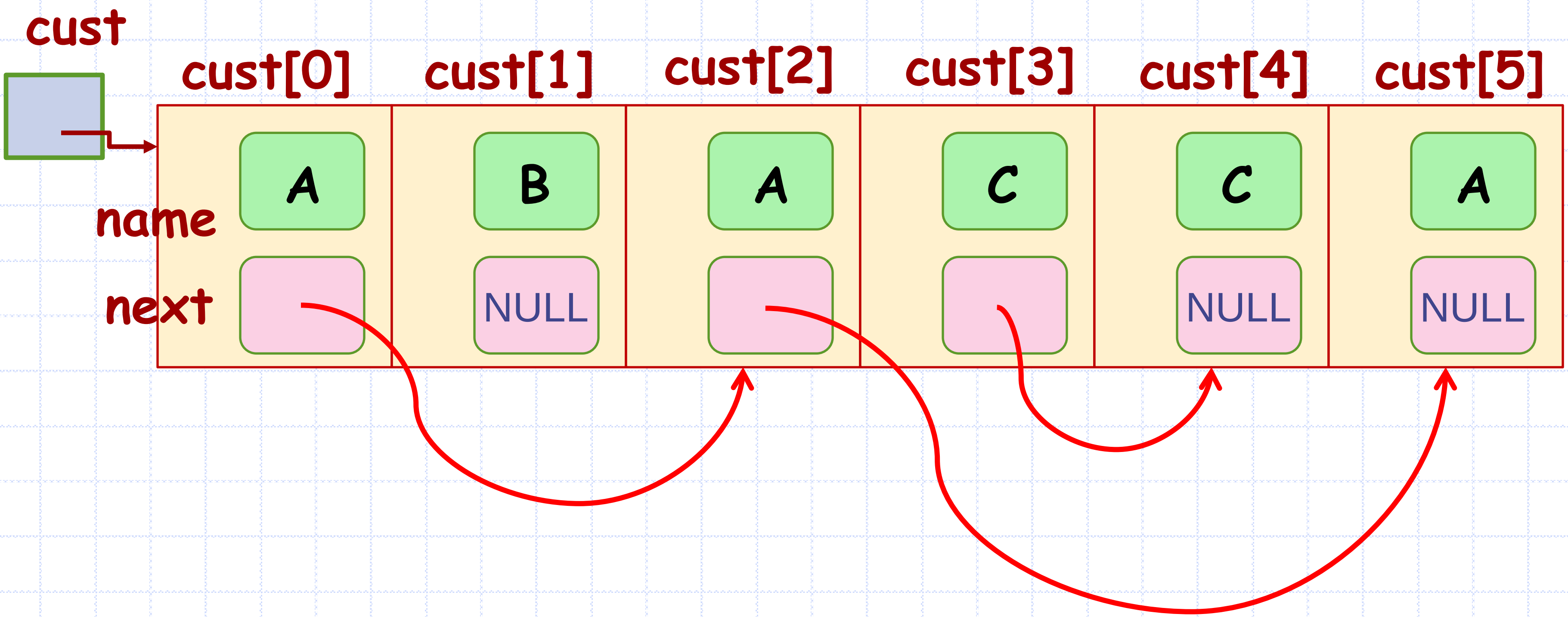
Customer Information : Updated

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

```
    int Account_Number;
    int Account_Type;
    char *Customer_Name;
    char* Customer_Address;
    bitmap Signature_scan; // user defined type bitmap
    struct cust_info* next_account;
};
```

- Why not (?):
  - struct cust_info next_account;

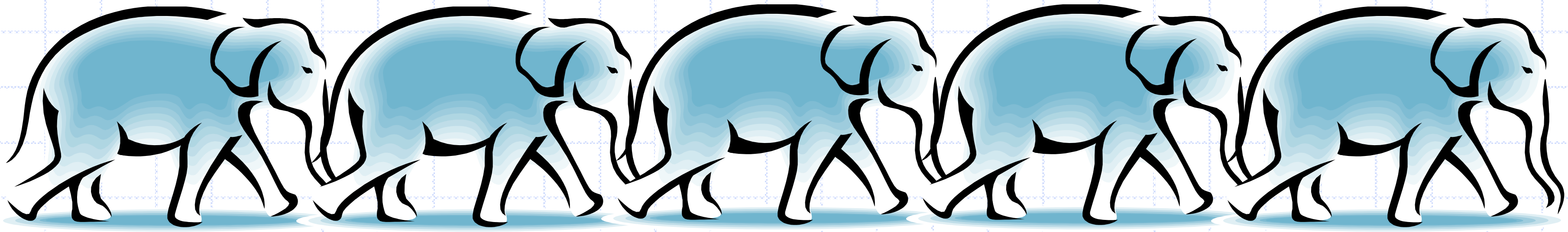
Error: Field next_account has **incomplete type**



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

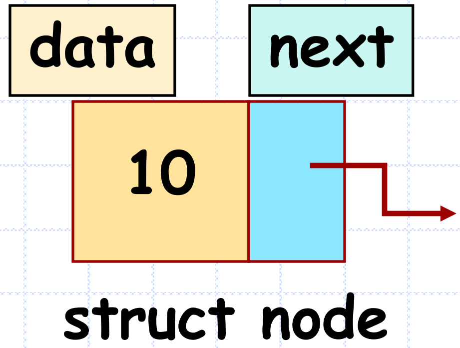
- A linear, dynamic data structure, consisting of nodes. Each node consists of two parts:
  - a "data" component, and
  - a "next" component, which is a pointer to the the next node (the last node points to nothing).



Linked List : A Self-referential structure

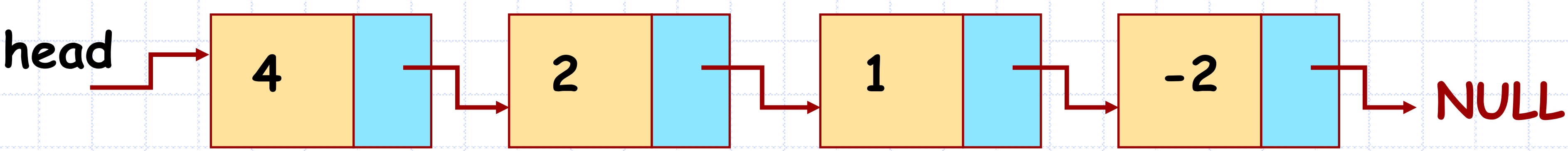
Example:

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



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 next 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".