Ceng 140

Linked Lists

An abstract data type: Lists

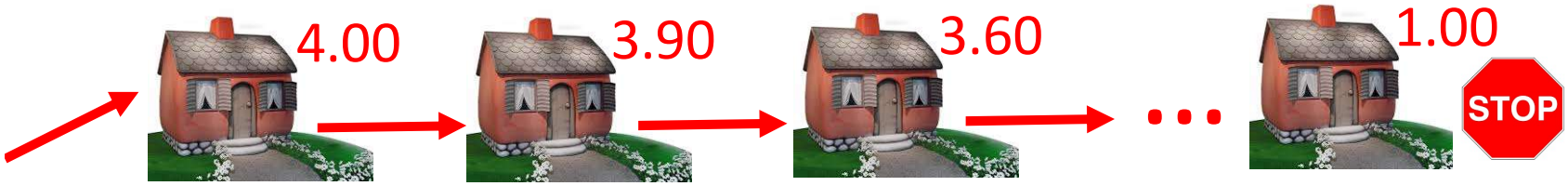
- A list is a sequence of zero or more elements of a given type.
  - $x_1, x_2, x_3, \ldots, x_n$
- **Arrays** can be used to i[illegible]ut:
  - Example: Keep list of CGPAs

4.00	3.90	3.60	3.20	2.80	2.00	1.90	1.00

  - insert/delete is hard – requires shifting elements
  - may waste space or not fit!

Lists

- In the linked list implementation of lists, pointers are used to link successive list elements.

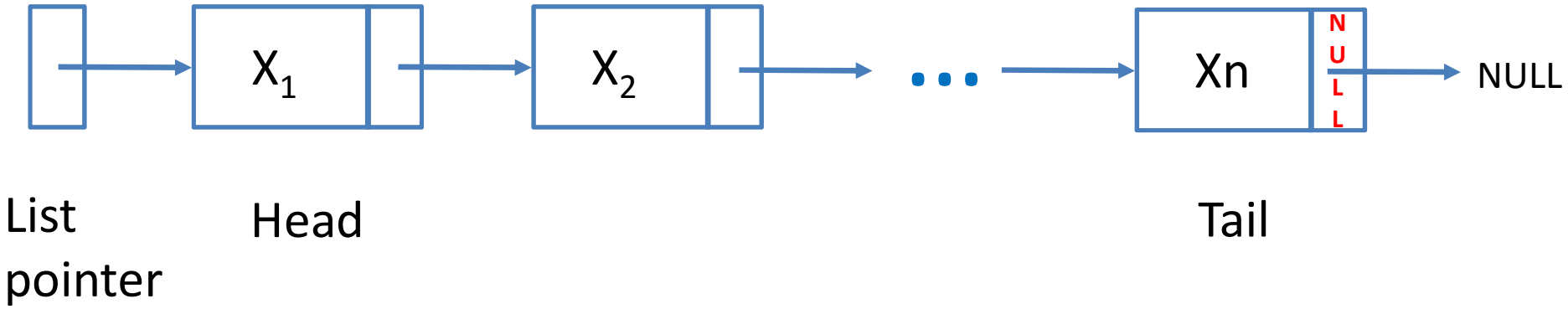


Lists

- In the linked list implementation of lists, pointers are used to link successive list elements.

- A singly linked list is made up of **nodes**, each consisting of an element on the list, and a pointer to the next node!

- A singly linked list is made up of nodes, each consisting of an element on the list, and a pointer to the next node!
  - $x_1, x_2, x_3, \ldots, x_n$



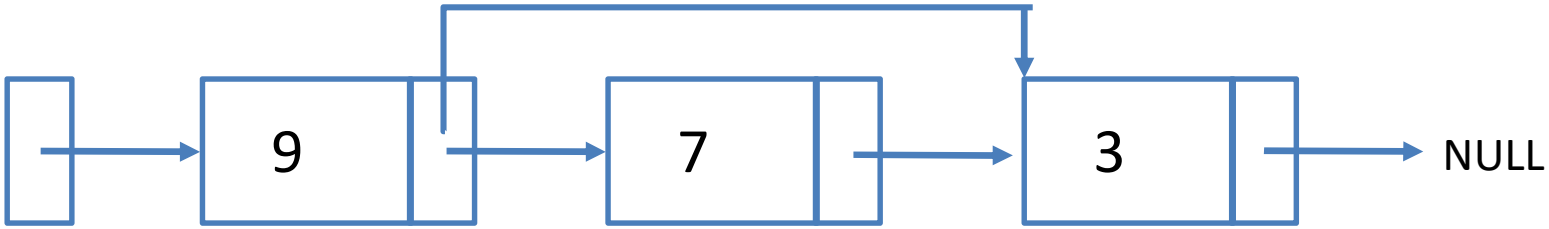
(Singly) Linked Lists

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

Let's create a list that is kept in descending order of the data values, e.g., 9, 7, 4, 3, 1…

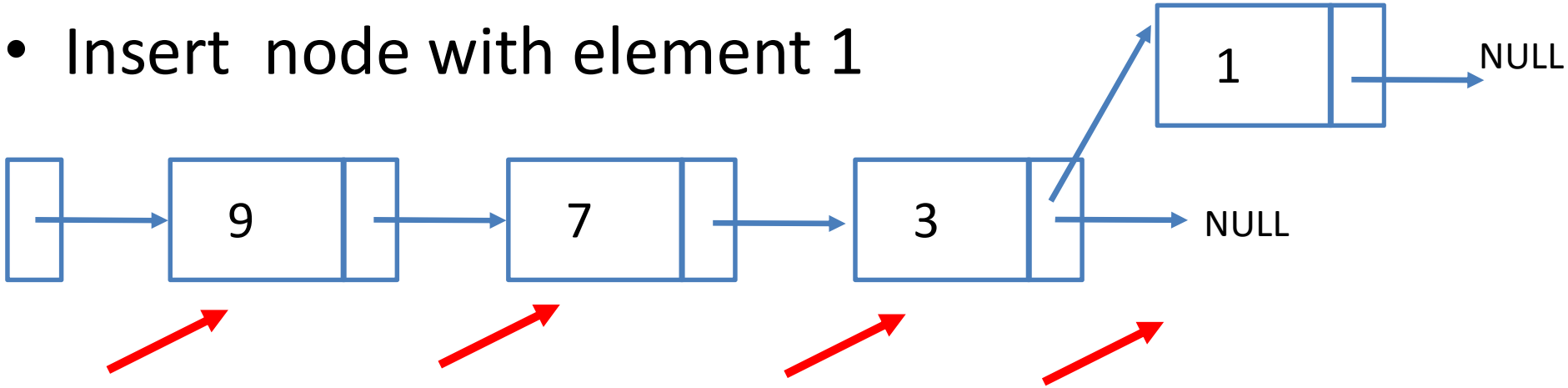
Example

- Deleting node with element 7

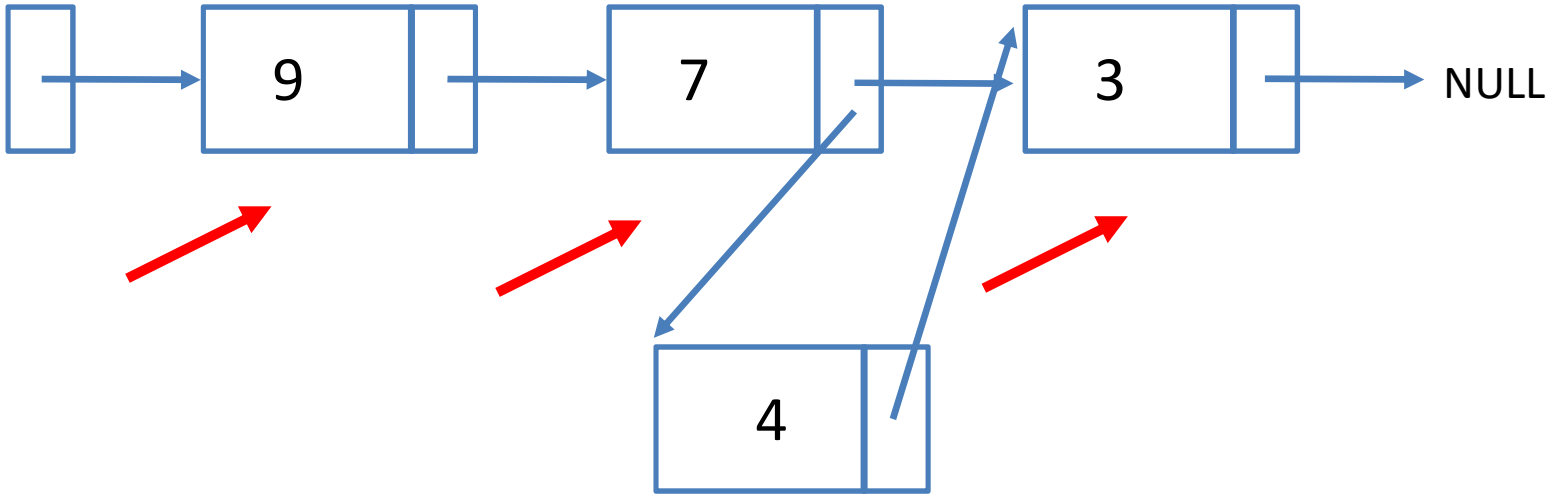


Example

- Insert node with element 1

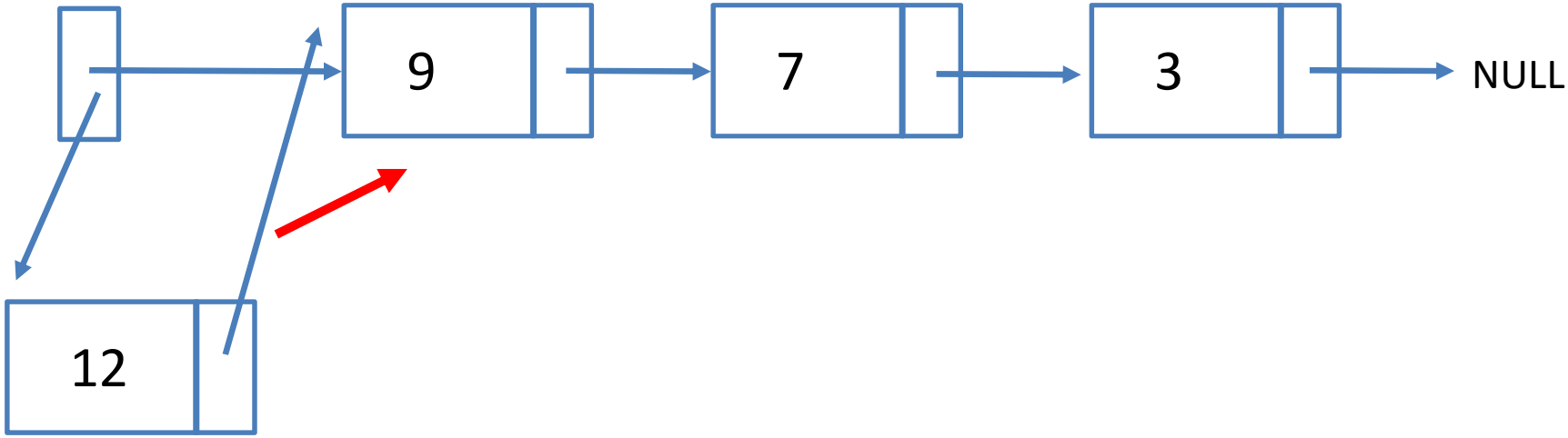


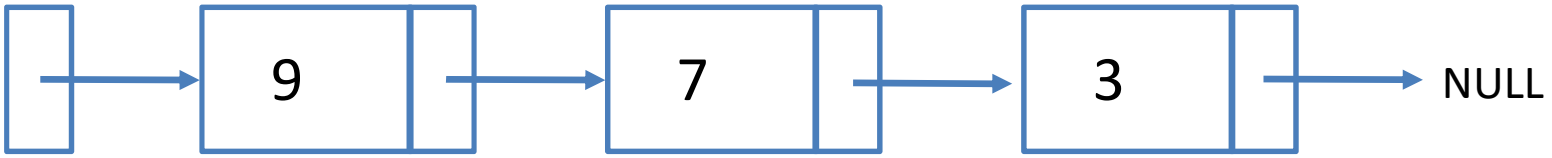
- Insert node with element 4



Example

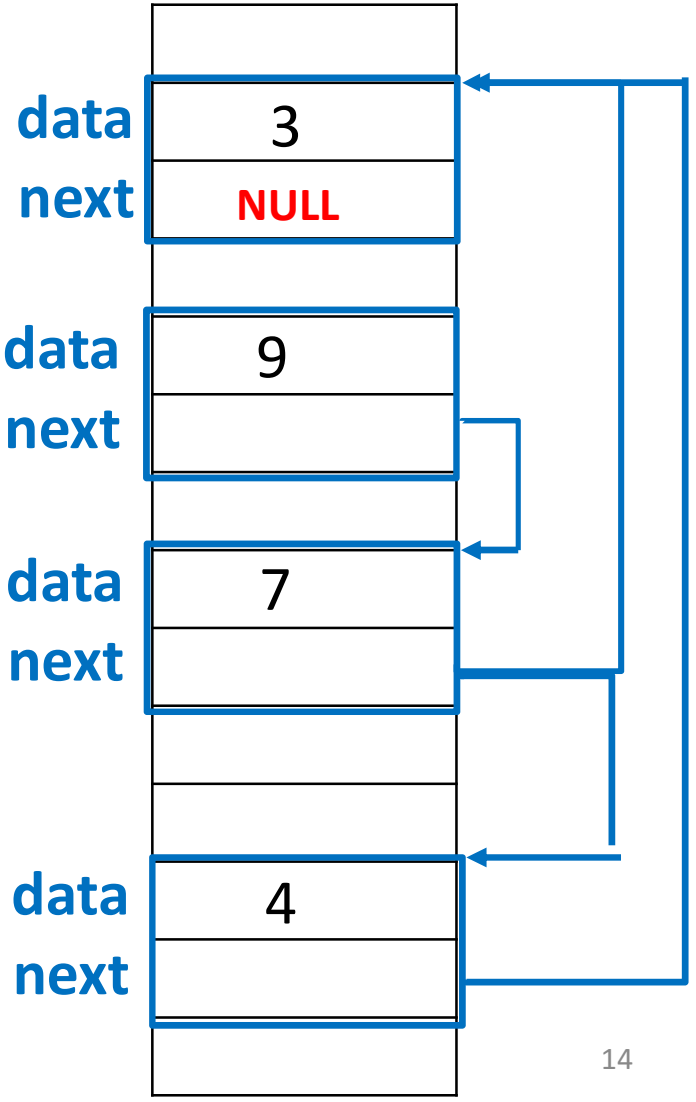
- Insert node with element 12






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

- Insert node with element 4

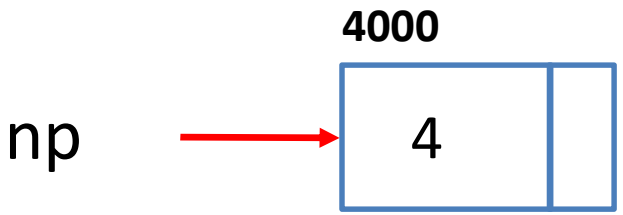


Lists

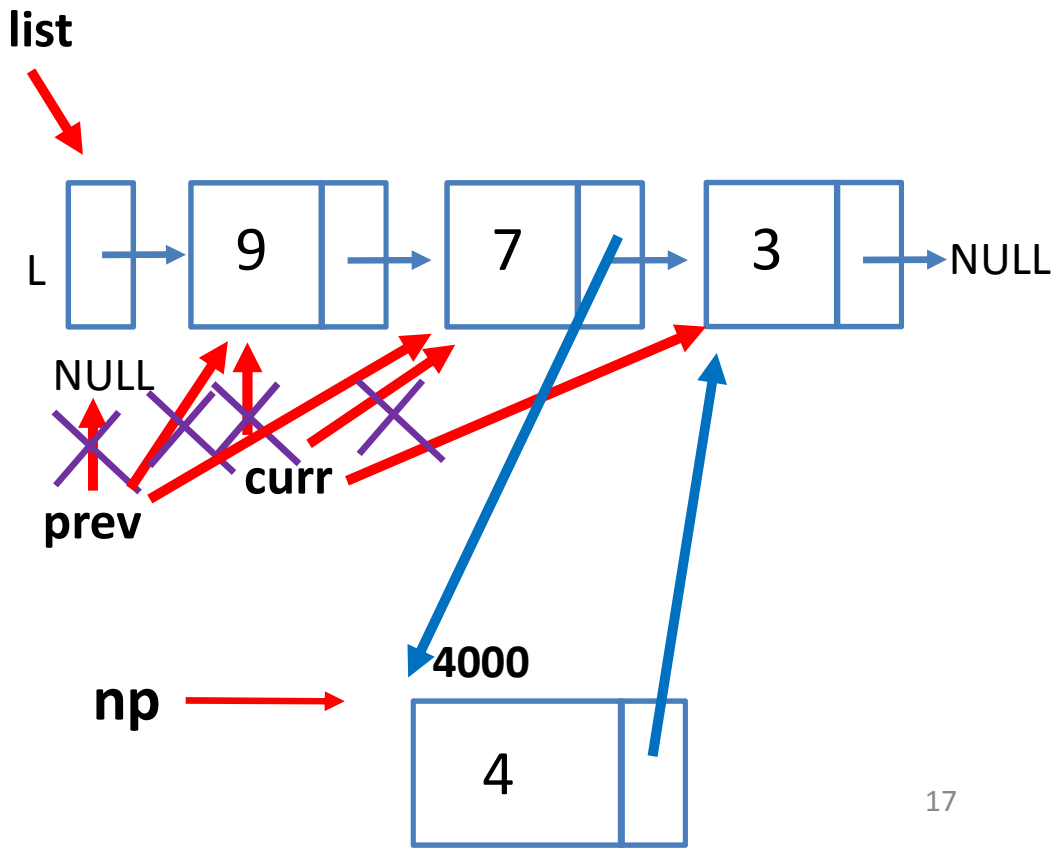
- In the linked list implementation of lists, pointers are used to link successive list elements.
- A singly linked list is made up of nodes, each consisting of an element on the list, and a pointer to the next node!
  - insert/delete is easy
  - can alloc/dealloc storage dynamically!

(Singly) Linked Lists

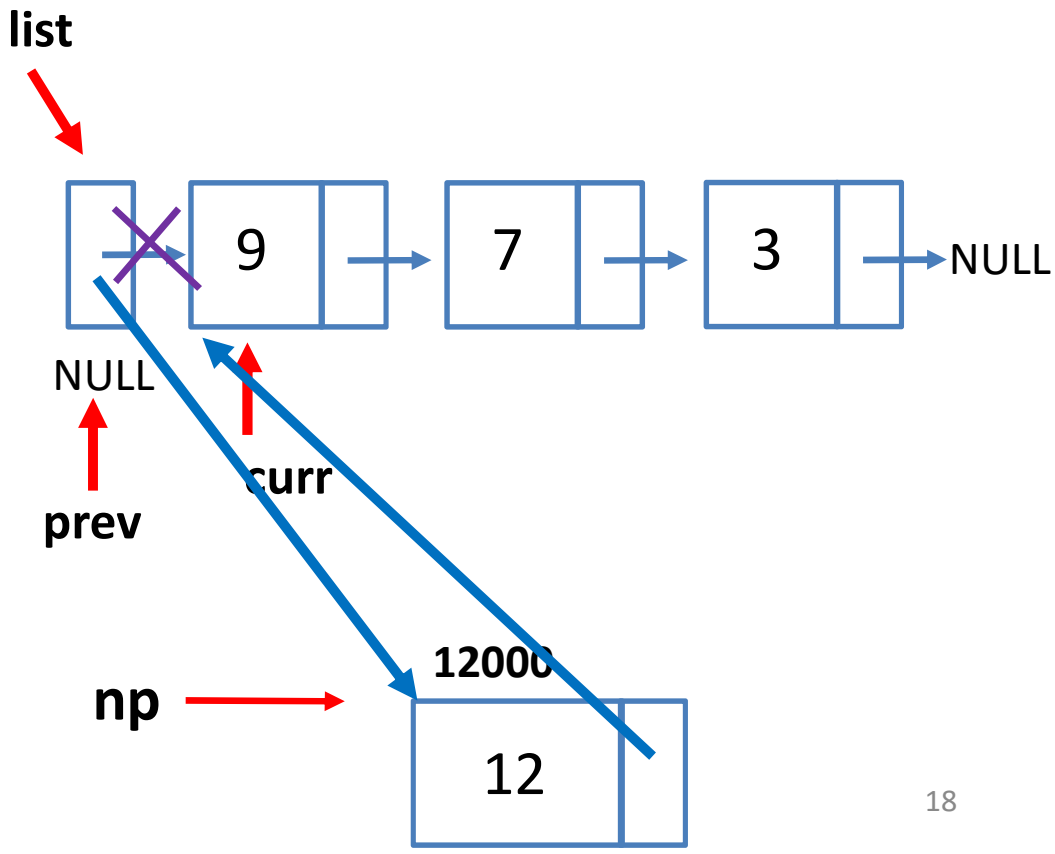
```
struct node * mknode(int data)
{ struct node *np;
   np = (struct node *) malloc ( sizeof(struct node));
   if (np)
   { np → data = data;
     np → next = NULL; }
   return np;
}
```



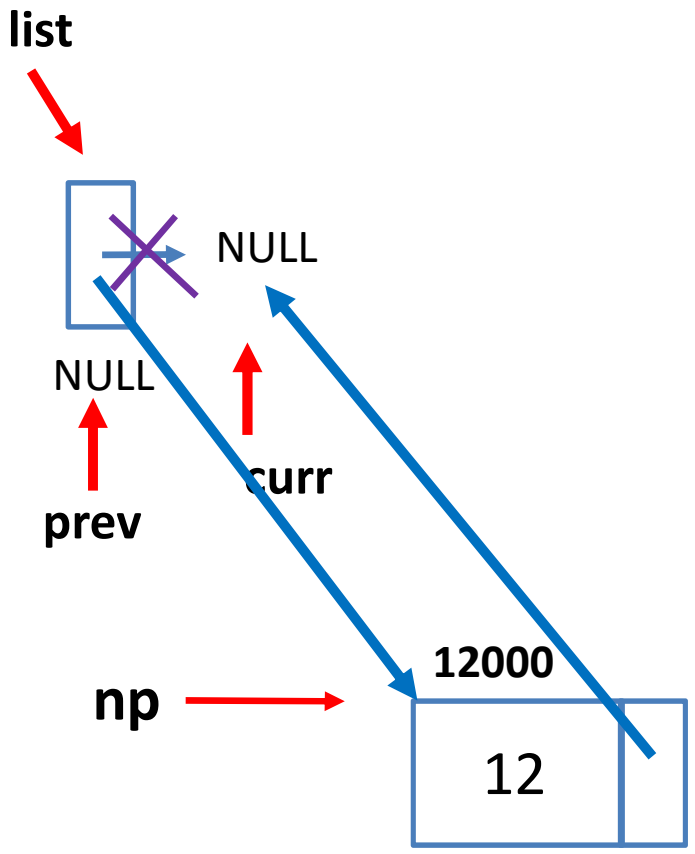
```
struct node * insert(struct node **list, int data)
{ struct node *np;
  if (np = mknode(data))
  { struct node *curr, *prev;
    curr = *list; prev = NULL;
    while (curr && data < curr → data)
    { prev = curr;
      curr = curr → next; }
    np→ next = curr;
    if (prev)
      prev → next = np;
    else
      *list = np; }
return  np; }
```



```
struct node * insert(struct node **list, int data)
{ struct node *np;
  if (np = mknode(data))
  { struct node *curr, *prev;
    curr = *list; prev = NULL;
    while (curr && data < curr → data)
    { prev = curr;
      curr = curr → next; }
    np→ next = curr;
    if (prev)
      prev → next = np;
    else
      *list = np; }
return  np; }
```

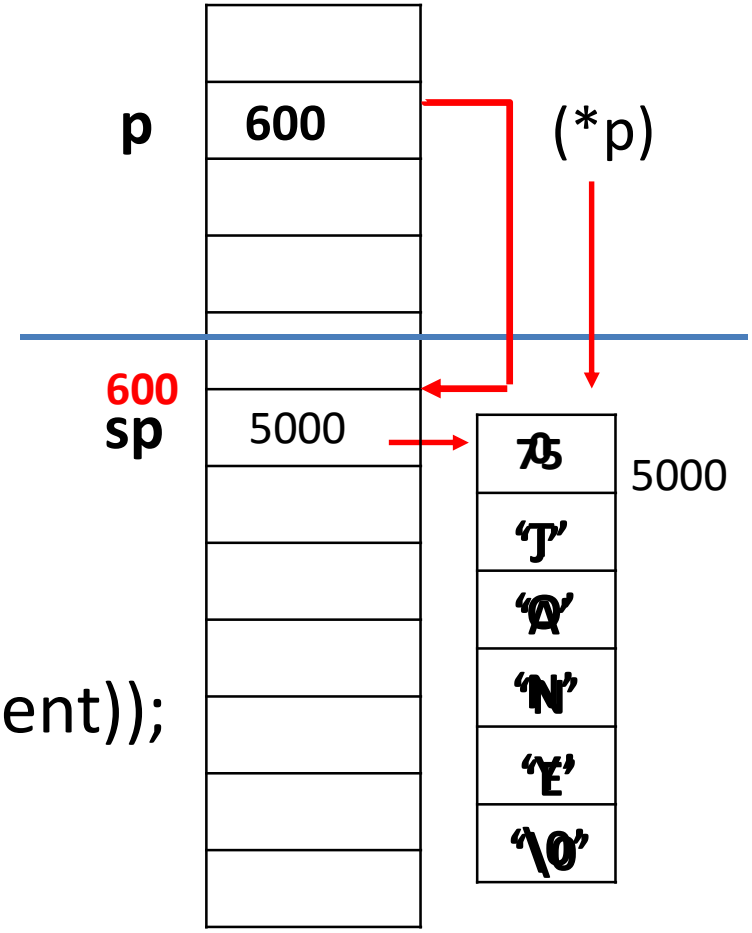


```
struct node * insert(struct node **list, int data)
{ struct node *np;
  if (np = mknode(data))
  { struct node *curr, *prev;
    curr = *list; prev = NULL;
    while (curr && data < curr → data)
    { prev = curr;
      curr = curr → next; }
    np→ next = curr;
    if (prev)
       prev → next = np;
    else
       *list = np; }
return  np; }
```

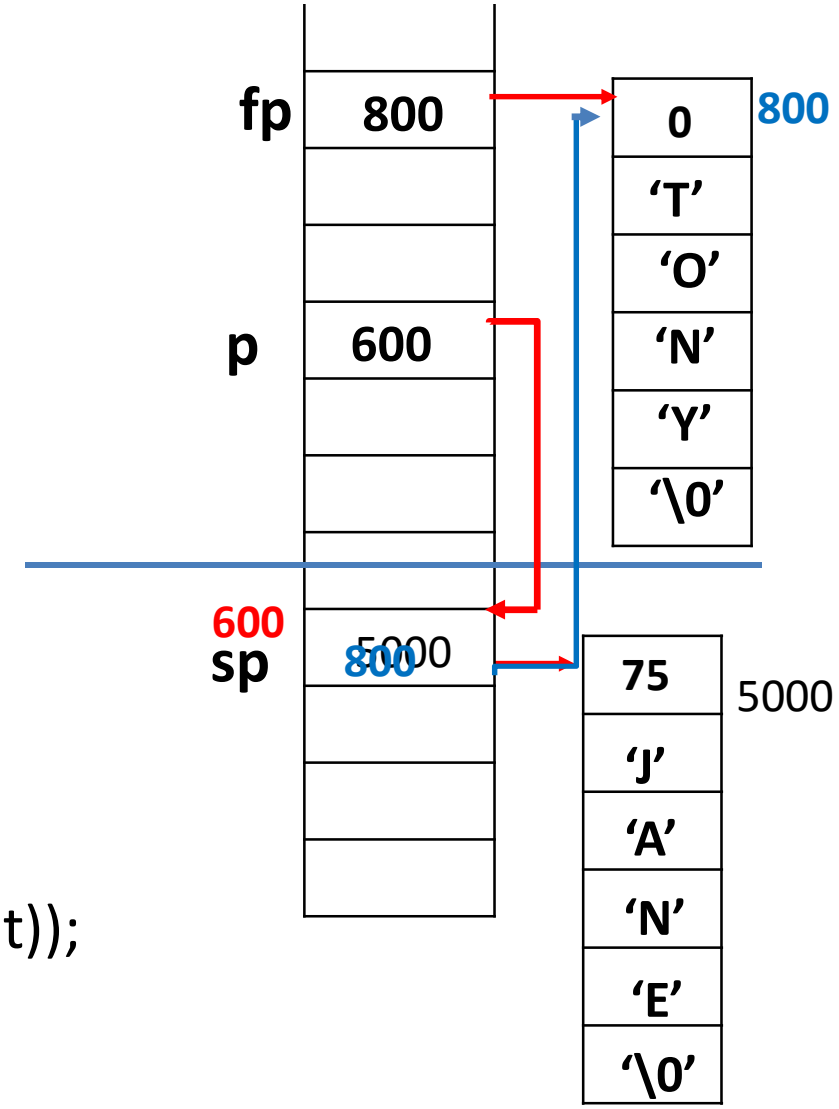


```
void g(struct student **p)
{(*p) → grade = 0;
   strcpy( (*p) → name, "TONY");}

int main()
{ struct student *sp;
  sp = (struct student *)
                malloc (sizeof(struct student));
  sp -> grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp → name,sp → grade); // Output?
  g(&sp);
 printf("see %s %d\n", sp → name,sp → grade); // Output?}
```



```
void g(struct student **p)
{ struct student *fp;
  fp = (struct student *) malloc (...));
   fp → grade = 0;
   strcpy( fp → name, "TONY");
 *p = fp; }
int main()
{ struct student *sp;
  sp = (struct student *)
                malloc (sizeof(struct student));
  sp -> grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp → name,sp → grade); // Output?
  g(&sp);
 printf("see %s %d\n", sp → name,sp → grade); // Output?}
```



Time for the Course Evaluations!!!

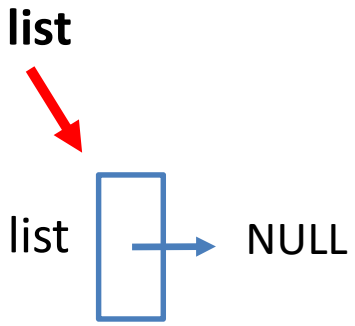
Deadline: 23 January, 2022

student.metu.edu.tr

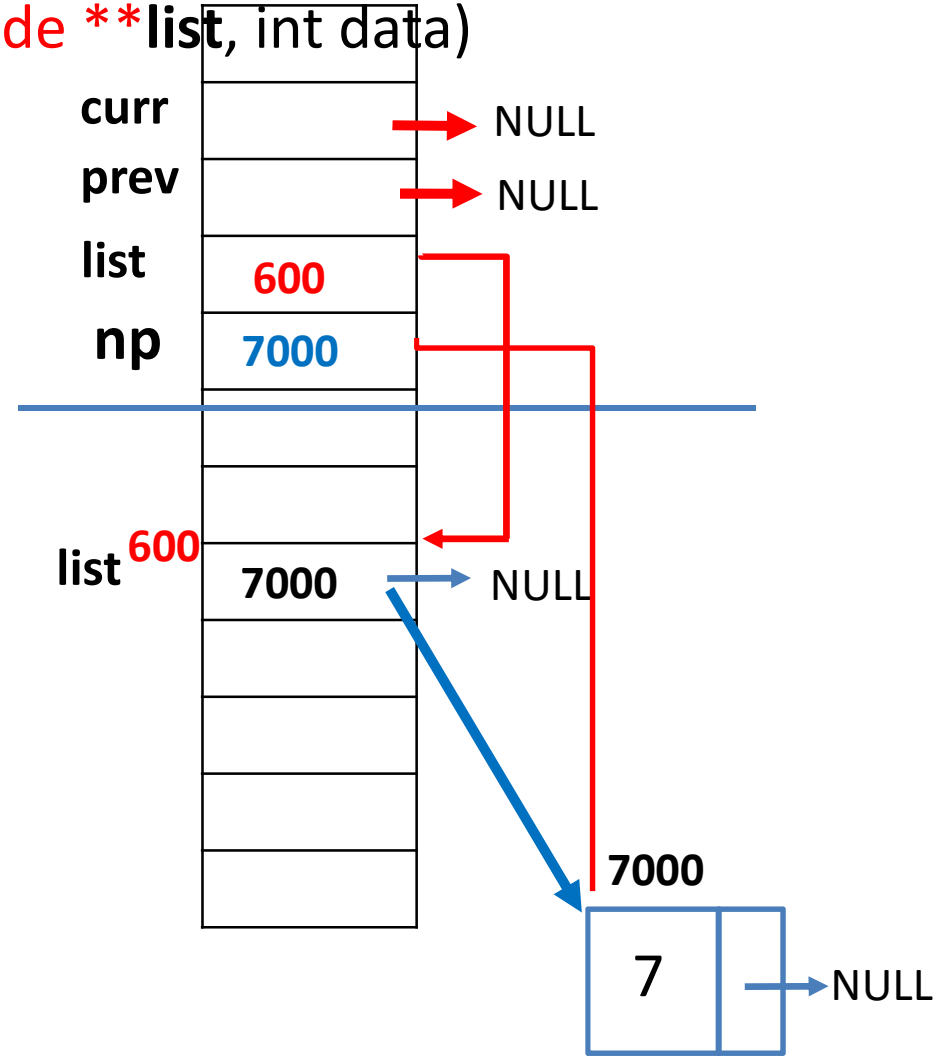
```
struct node * insert(struct node **list, int data)
{ struct node *np;
  if (np = mknode(data))
  { struct node *curr, *prev;
    curr = *list; prev = NULL;
    while (curr && data < curr → data)
    { prev = curr;
      curr = curr → next; }
    np→ next = curr;
    if (prev)
        prev → next = np;
    else
        *list = np; }
return  np; }
```

Hey…This is… insertion sort!

```
void sort()
{ struct node *list = NULL;
   int i;
   while (scanf("%d", &i) !=EOF && insert(&list , i))
       ;
   print(list);
}
```



```
struct node * insert(struct node **list, int data)
{ struct node *np;
  if (np = mknode(data))
  { struct node *curr, *prev;
    curr = *list; prev = NULL;
   while (curr && data < curr → data)
    { prev = curr;
      curr = curr → next; }
    np→ next = curr;
    if (prev)
        prev → next = np;
    else
        *list = np; }
 }
```

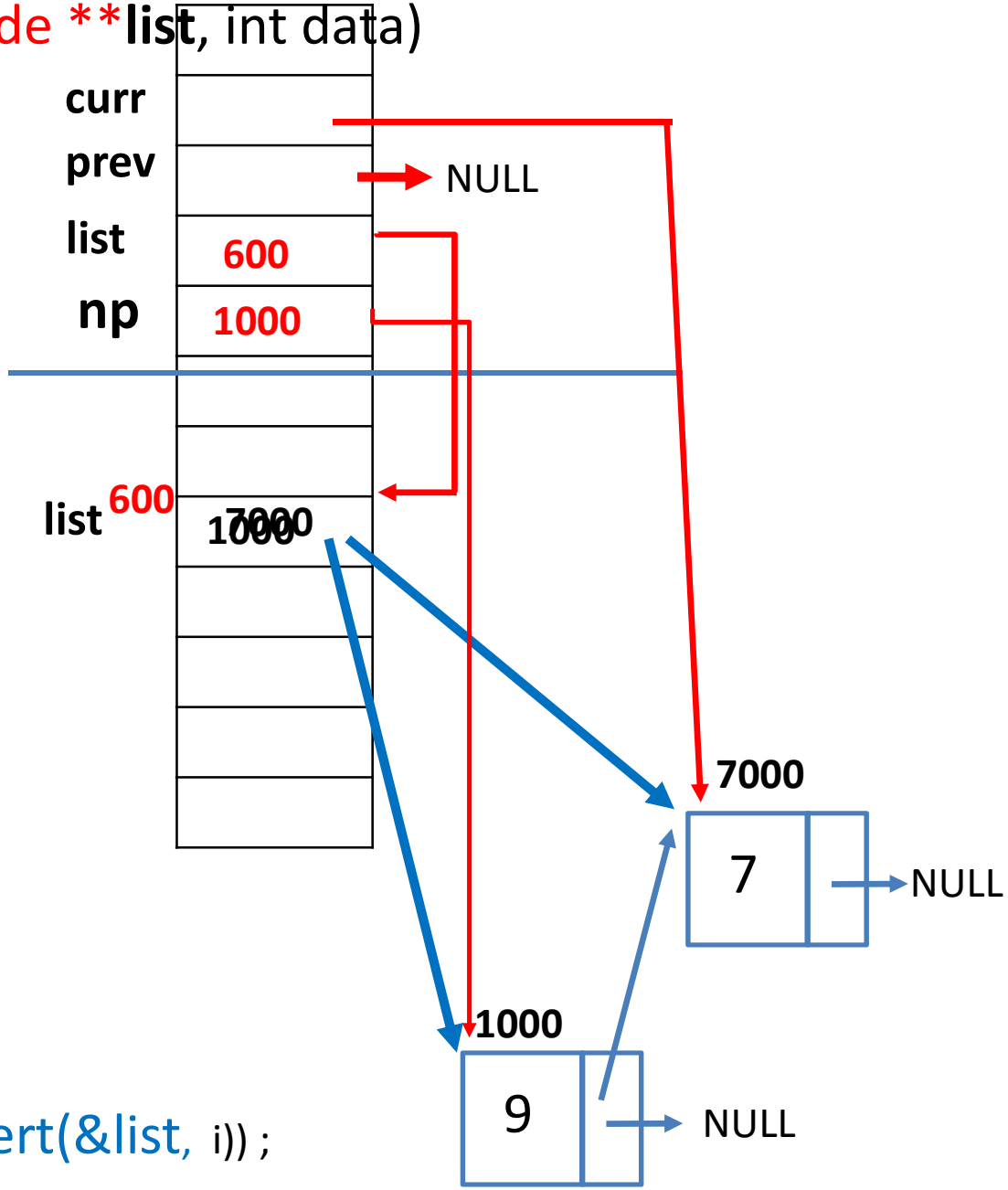



```
void sort()
{ struct node *list = NULL;
  int i;
  while (scanf("%d", &i) !=EOF && insert(&list, i)) ;

}
```

```
struct node * insert(struct node **list, int data)
{ struct node *np;
  if (np = mknode(data))
  { struct node *curr, *prev;
    curr = *list; prev = NULL;
   while (curr && data < curr → data)
    { prev = curr;
      curr = curr → next; }
    np→ next = curr;
    if (prev)
        prev → next = np;
    else
        *list = np; }
}
```

```
void sort()
{ struct node *list = NULL;
  int i;
  while (scanf("%d", &i) !=EOF && insert(&list, i)) ;
}
```

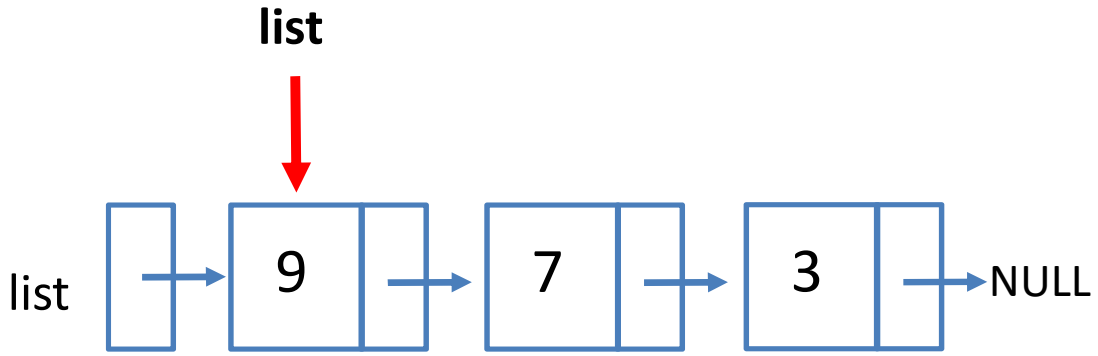


How can we have **an insert function** that is **not** taking a "ptr to ptr" as parameter?

HW-1

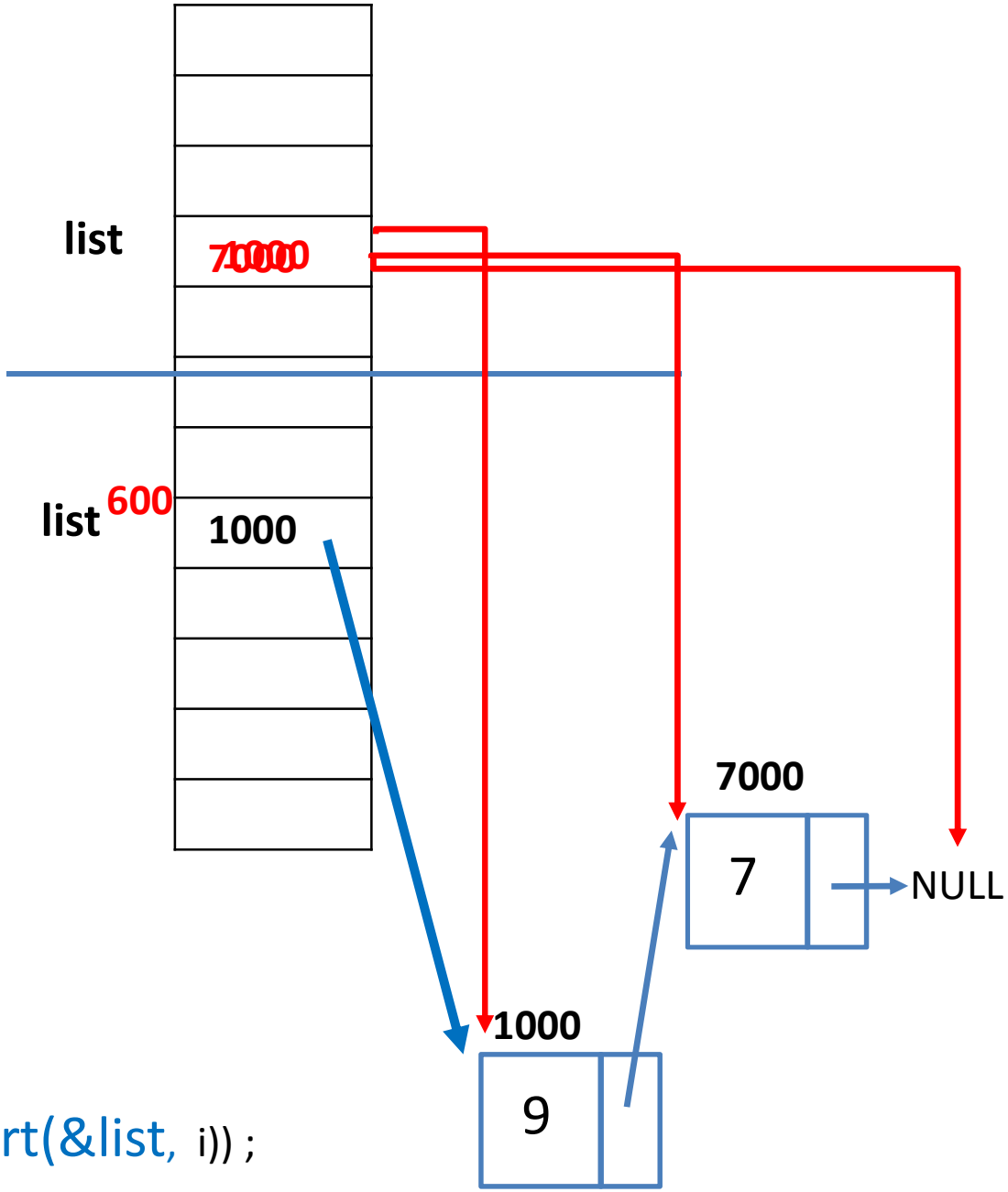
- Try to solve yourself until the next lecture!

```
void print(struct node *list)
{  while (list)
    {  printf ("%d", list → data);
       list = list → next
    }
}
```




```
void sort()
{ struct node *list = NULL;
  int i; while (scanf("%d", &i) !=EOF && insert(&list,  i)) ;
  print(list);
}
```

```
void print(struct node *list)
{  while (list)
    {  printf ("%d", list → data);
       list = list → next;
     }
}
```

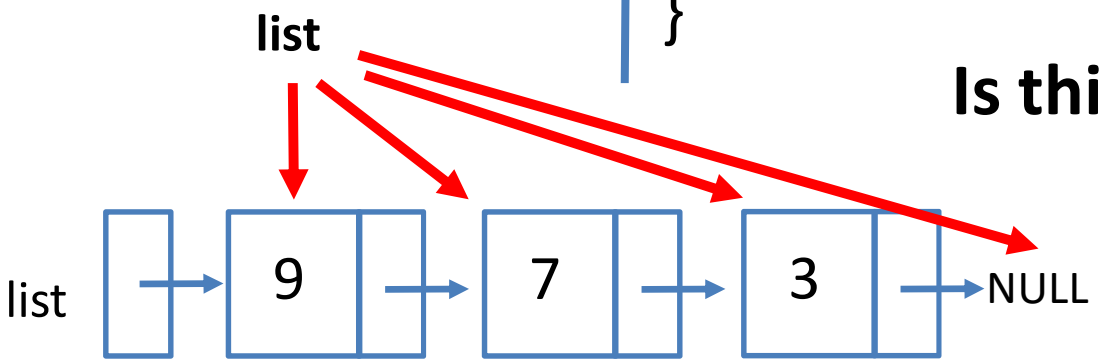



```
void sort()
{ struct node *list = NULL;
  int i;
  while (scanf("%d", &i) !=EOF && insert(&list, i)) ;
 print(list);
}
```

A recursive print

```
void print(struct node *list)
{  while (list)
   {  printf ("%d", list → data);
      list = list → next;
   }
}
```

```
void rec_print(struct node *list)
{   if (list)
    {  printf ("%d", list → data);
       rec_print(list → next );
    }
}
```



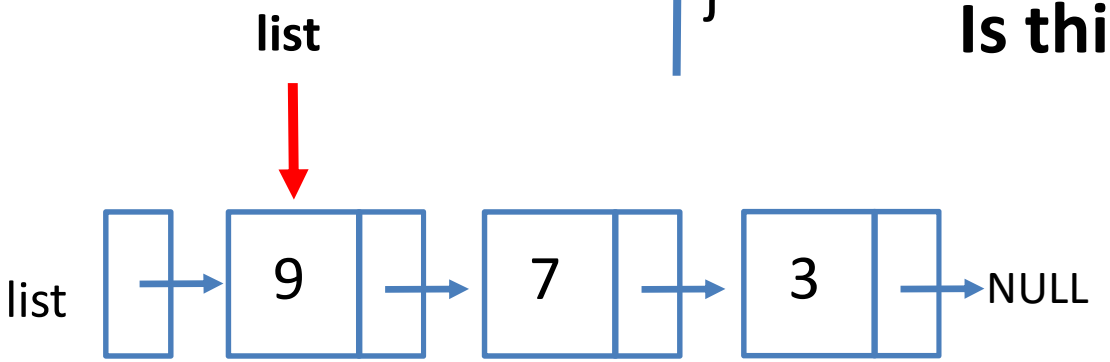
Is this tail recursive?

Print in reverse order?

```
void rec_print(struct node *list)
{   if (list)
    {  printf ("%d", list → data);
       rec_print(list → next);
    }
}
```

```
void rev_print(struct node *list
{   if (list)
    {rec_print(list → next);
     printf ("%d", list → data);
    }
}
```

Is this tail recursive?



Print in reverse order: Iterative & Constant Space Usage

HW-2

- Try to solve yourself until the next lecture!

typedef

- Allows to define synonyms for existing data types. Declaration:
  - **typedef** existing_typename **declarator**;
  - **typedef** float **REAL**;
  - **typedef** unsigned short int **BOOL, BOOLEAN;**
  - **typedef** char **STRING**[MAX];
    // for type char arrays of MAX elements
  - **typedef** int * **FUNC**(short, short)
    // synonym for type func taking shortparams and returning an integer pointer

typedef

- For defining a synonym for a type:

  1) write as if you are declaring a variable of that type,

  2) substitute the synonym in place of the variable name

  3) precede with keyword typedef

  char s[MAX]; // 1) declare var

  char **STRING**[MAX]; // 2) substitute the synonym

  typedef char **STRING**[MAX]; // 3) preceding typedef

typedef

- The synonym may appear anywhere a type specifier is allowed:

```
typedef unsigned short int BOOL, BOOLEAN;
typedef char STRING[MAX];
typedef int * FUNC(short, short)

BOOLEAN flag; // unsigned short int flag;
STRING s;     // char s[MAX];
FUNC f;       // int * f(short, short);
```

But cannot be mixed with other type specifiers

```
typedef short int SMALLINT;
unsigned SMALLINT s;
```

typedef

- Again, to define synonyms for existing data types:

```
typedef struct person {
                char name[10];
                int id; } PERSON, HUMAN;
```

- struct person, PERSON, HUMAN refer to the same type!

typedef

- Resembles #define, but…!

#define BYTE char *

BYTE cp1, cp2;

// preprocessor handles macro & replaces textually

//char * cp1, cp2; so cp2 is of type char here!

versus:

typedef char * BYTE;

BYTE cp1, cp2;

// compiler handles typedef, both are type BYTE

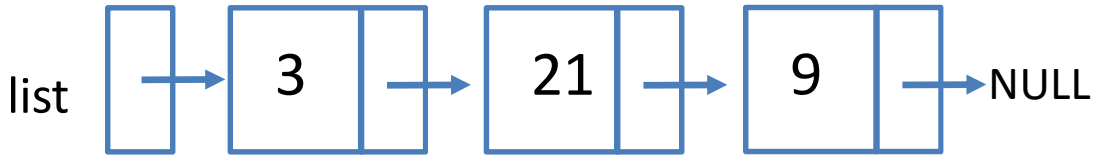
// and hence, char *

```
typedef struct node {
        int data;
        struct node *next; } ListElem, *List;
List mknode(int data)
{ List np;
  np = List malloc ( sizeof( ListElem ));
  if (np)
  { np → data = data;
    np → next = NULL; }
  return np;
}
```

```
typedef struct node {
        int data;
        struct node *next; } ListElem, *List;
void insert(List *L, int data)
{
```

This time my list is not sorted, but I will add the data to the list only if it is NOT already in the list…

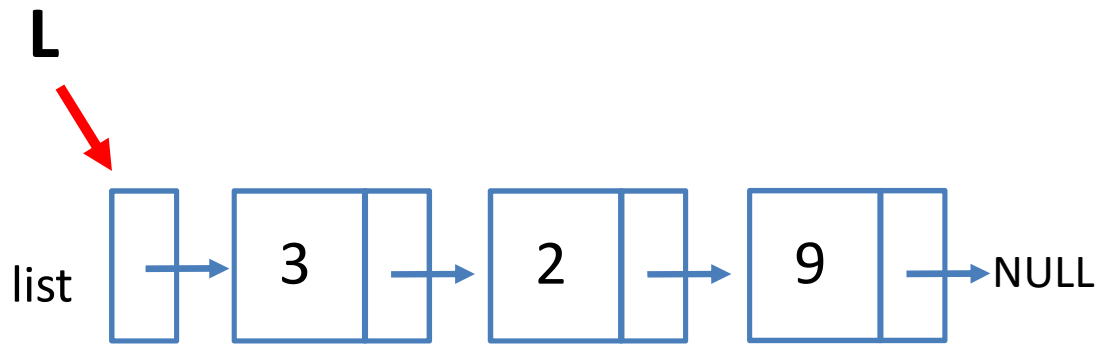
- Iteratively? HW-3



- Recursively?

```
}
```

```
typedef struct node {
        int data;
        struct node *next; }
      ListElem, *List;
```



list is of type **List**

```
void insert(List *L, int data)
{ if ( (*L) == NULL)
  {  (*L) = (List) malloc (sizeof(ListElem));
     (*L) → data = data;
     (*L) → next = NULL; }
  else if ( (*L) → data != data)
  // call insert recursively,what points to rest of the list?
      insert(   & ((*L) → next)   , data);   }
```

Best practice: A good use of macros with arguments

```
typedef struct node {
        int data;
        struct node *next; } ListElem, *List;

#define DATA(p) ((p)->data)
#define NEXT(p) ((p)->next)
```

Btw, our slides show arrow operator like this → but in the real C program, it is just -> (no space between – and >