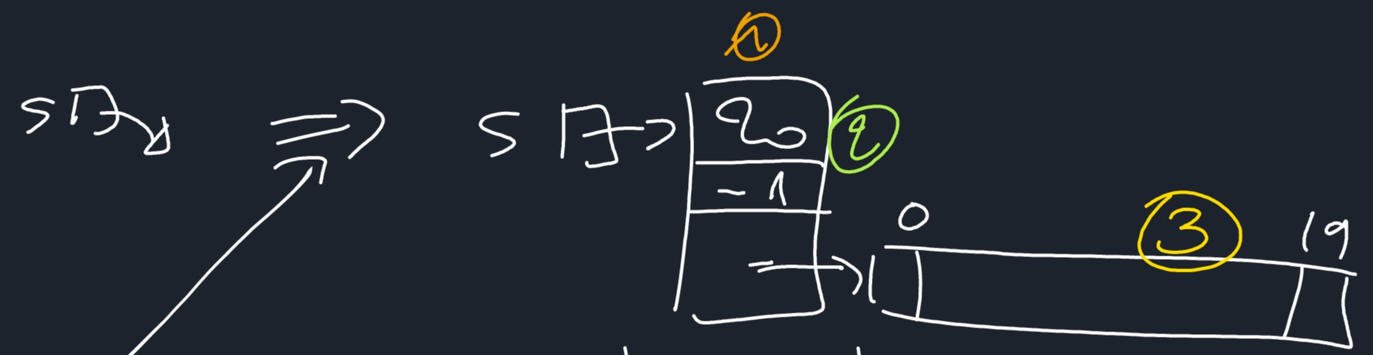
Stack.h

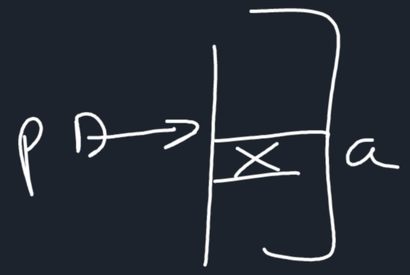
tStack.c

```
Stack *s = createStack (8);
if (!res->elements) {
    free(res);
    return 0;
}
return res;
}
```




```
Stack* createStack(int capa) {
    Stack* res = (Stack*)
        malloc(sizeof(Stack));
    if (!res) return 0;
    res->top = -1;
    res->capacity = capa;
    res->elements = (int*)
        malloc(capa * sizeof(int));
```

s.a

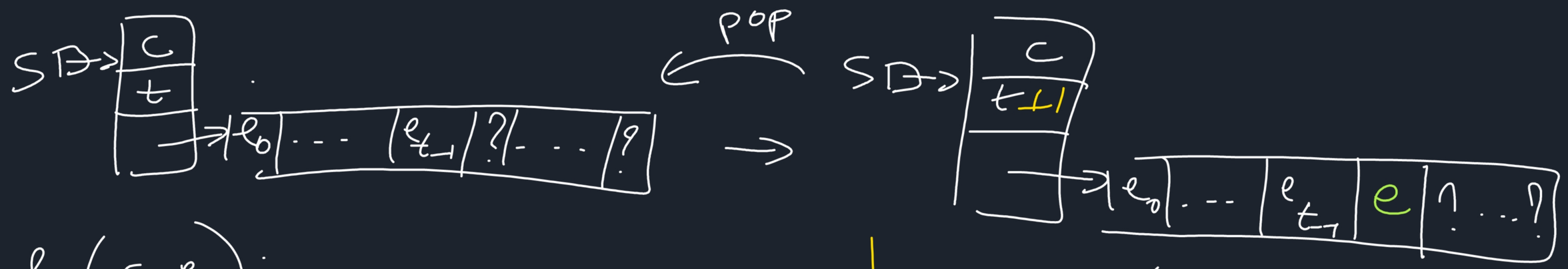


(*p).a

p -> a

((*(*p).a).b).c.d

p -> a -> b -> c -> d

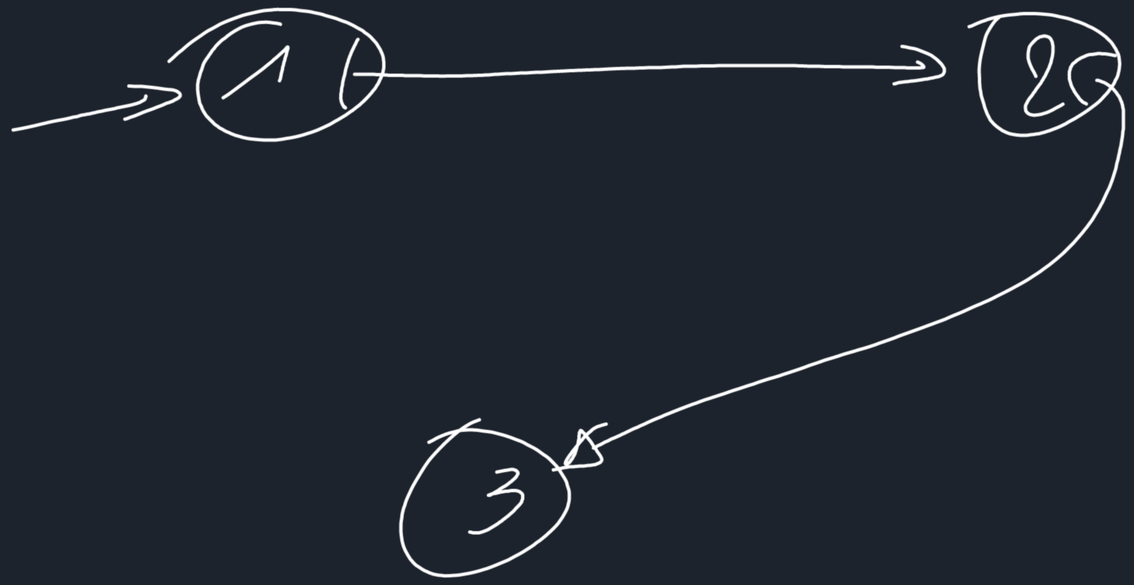


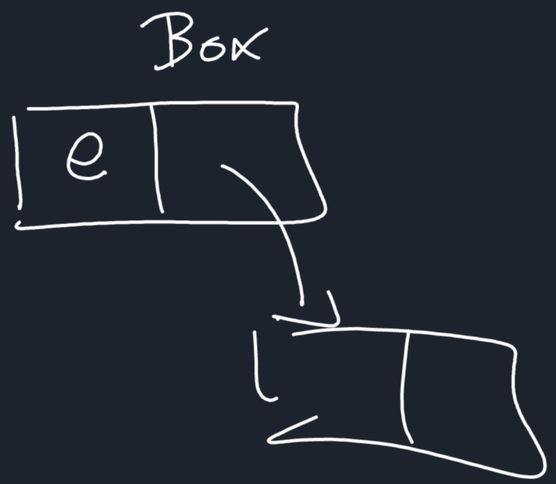

```
push(s, e);
void push(Stack * s, int e){
    s->elements[s->top] = e;
    s->top++;
}
```

```
void pop(Stack * s){
    s->top--;
}

int isEmpty(Stack * s){
    return s->top == -1;
}
```

```
int top(Stack * s){
    return s -> elements[s -> top];
}
void destroyStack(Stack*s){
    free(s->elements);
    free(s);
}
```


3
2
1


1
2
3


Box
e

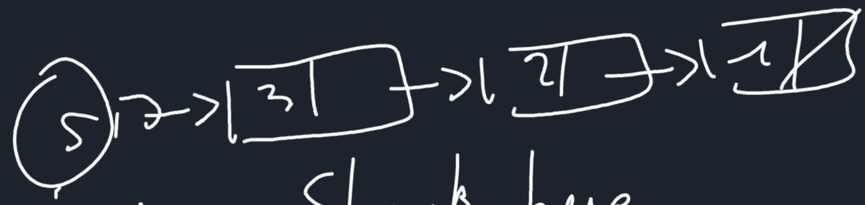
/* Stack.h */

```
typedef struct Box {
    int value;
    struct Box * next;
} Box, * Stack;
```

$\Longleftrightarrow$

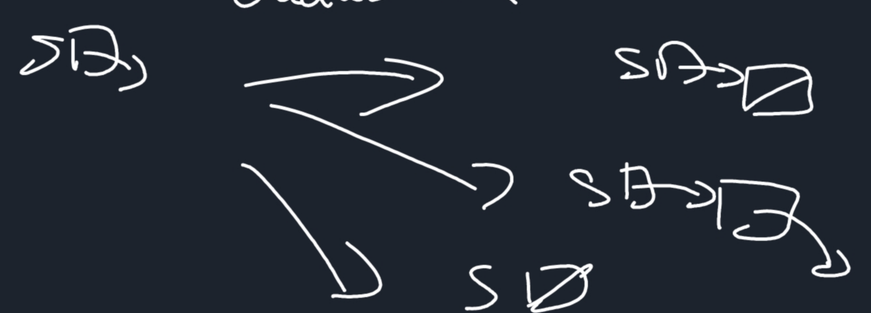
```
typedef Box * Stack;
```

same function prototypes (except createStack())



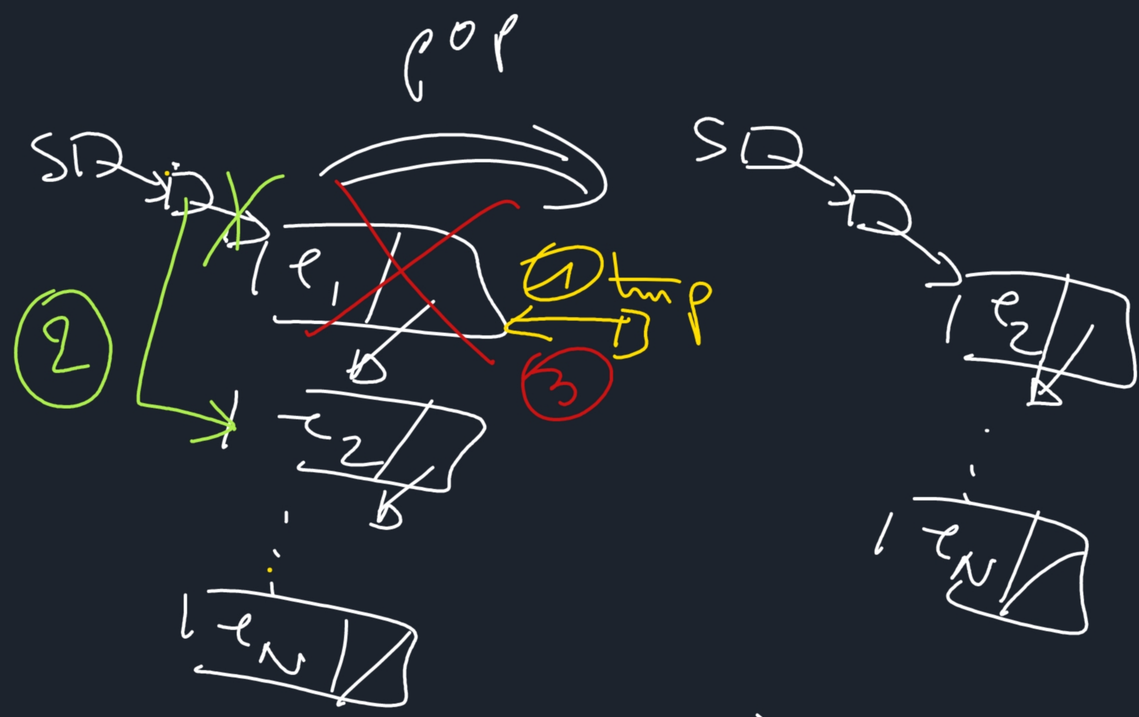
s is a Stack here, not a Stack* !

```
/* Stack.c */

#include "Stack.h"

Stack* createStack(){
    Stack* res = (Stack*)malloc(sizeof(Stack));
    if(!res) return 0;
    *res = 0;
    return res;
}

int isEmpty(Stack* s){ return *s == 0; }
```



```
void push(Stack* s, int e){
    Box* tmp = (Box*)malloc(sizeof(Box));
    if (!tmp) return;
    tmp->value = e;
    tmp->next = *s;
    *s = tmp;
}
```

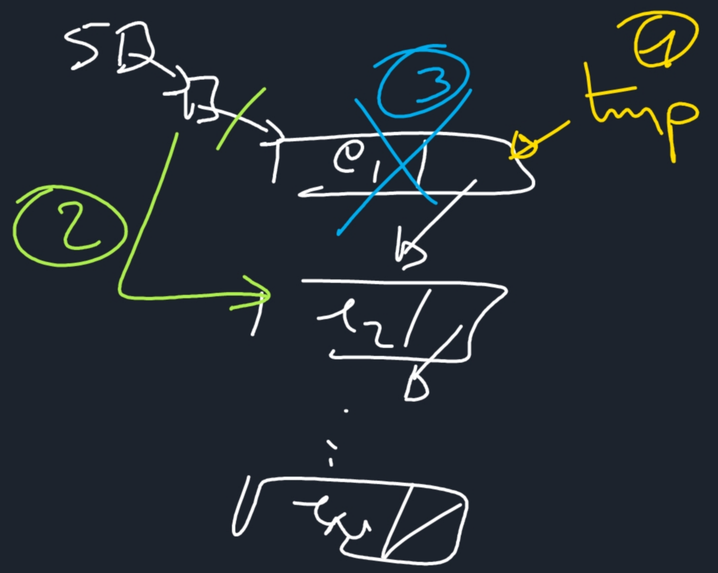
```
void pop(Stack * s){
    Stack tmp = *s;
    *s = tmp->next;
    free(tmp);
}
int isEmpty(Stack*s){
    return *s==0;
}
```




```
int top(Stack* s){
    return (*s)->value;
}
```

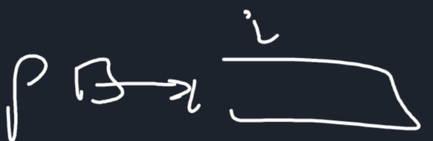
```
void destroyStack(Stack *s){
    while(!isEmpty(s))
        pop(s);
    free(s);
}
```

```
while(*s){
    Box *tmp = *s;
    *s = tmp->next;
    free(tmp);
}
free(s);
```



Pointers to functions

int i; ⟶ int * p; p = & i;




```
int add( int x, int y) { return x+y; }
int mult( int x, int y) { return x * y; }

int (* p)( int x, int y);
p = &add;
printf("%d\n", (*p)(3,4)); /* 7 */
p = &mult;
printf( ____________ ); /* 12 */
```

Convention: functions vs pointers to function

f OR P (...) ⟶ f OR P is called

f OR P ⟶ ____ is considered as a pointer: you get the address of the function

```
int add( ... ) { ... }
int mult( ... ) { ... }
int (*p)(int, int);

p = add;
printf("%d\n", p(3, 4));
p = mult;
printf
```

```
void sortArray(int* a, int size, int (*comp)(int, int)){
    int l, r;
    for(l=0; l<size-1; l++)
        for(r=l+1; r<size; r++)
            if(a[l] > a[r]){
                int tmp = a[l];
                a[l] = a[r];
                a[r] = tmp;
            }
}
```

comp(a[l], a[r])

```
int lessThan(int a, int b){return a<b;}
int greaterThan(int a, int b){return a>b;}
int (*comp)(int, int);

sortArray(a, 10, lessThan);
                 greaterThan
```

/* ↘ */

/* ↗ */

to achieve complete generality, we can "untype"
the array element by typing the array as "void *"

```
void sortArray ( int * a, int size, int (*comp)(int, int));
```

```
void sortArray ( void * a, int size, int eltSize, int (*comp)(void*, void*));
```

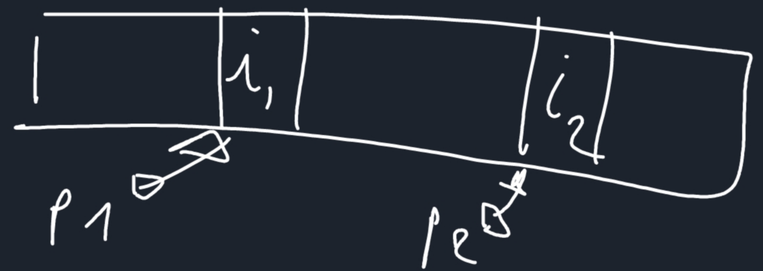
```
void sortArray(void* a, int size, int eltSize, int (*comp)(void*, void*))
{
  int l, r;
  int end = size * eltSize;
  void* tmp = malloc(eltSize);
  for(l = 0; l < end - eltSize; l += eltSize)
    for(r = l + eltSize; r < end; r += eltSize)
      if(comp(a+l, a+r) > 0) {
        memcpy(tmp, a+l, eltSize);
        memcpy(a+l, a+r, eltSize);
        memcpy(a+r, tmp, eltSize);
      }
  free(tmp);
}
```

memcpy

exercise : use this sortArray generic function to sort

- an array of int (decreasing)
- an array of strings (increasing)

(sortTest.c : compInt(), compString() and main())




```
int compInt(void *p1, void *p2){
    return *(int*)p2 - *(int*)p1;
}
```