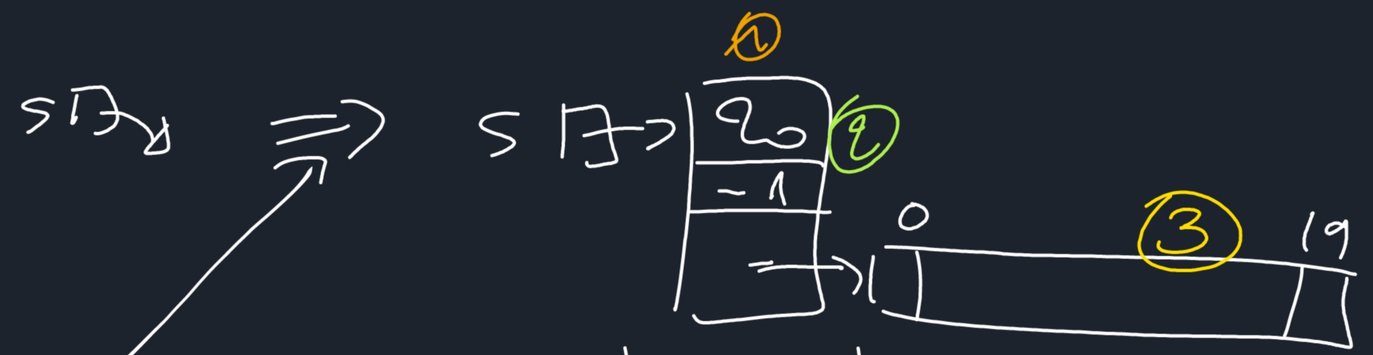
Stack.h

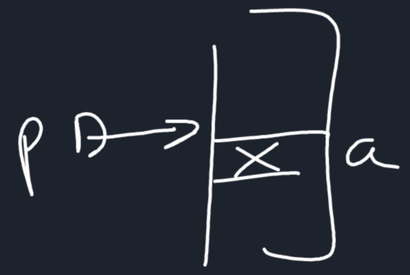
tStack.c

```
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

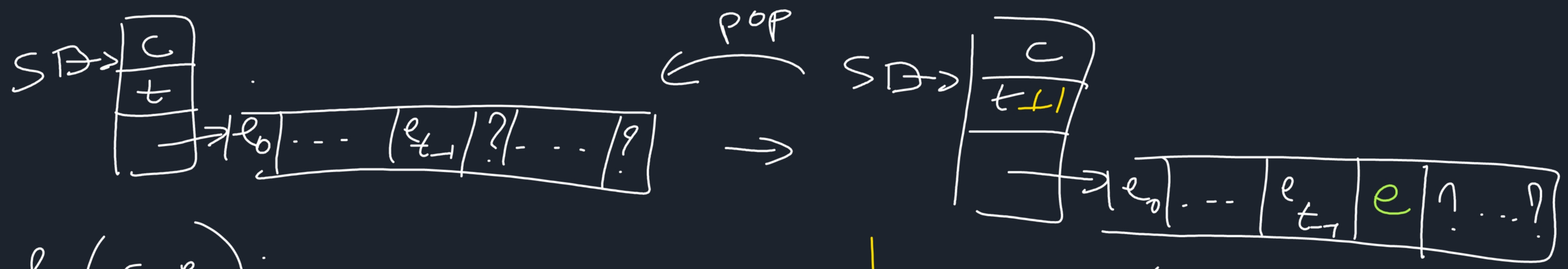


(*p).a

p -> a

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

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

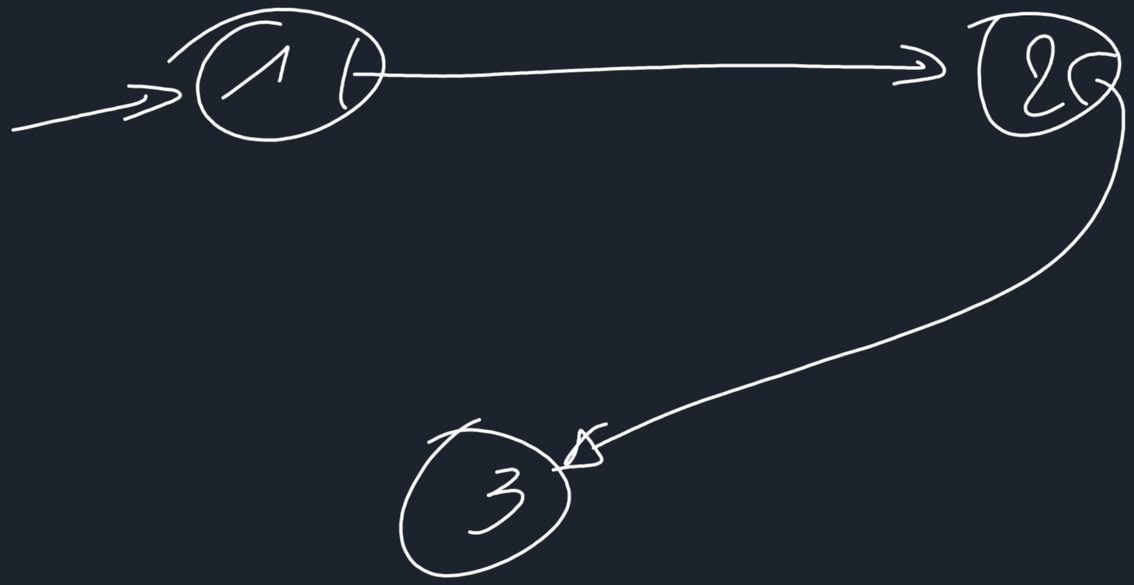


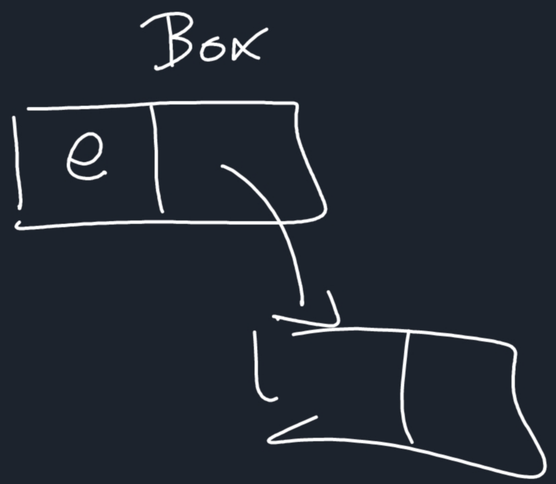

```
push(s, e);
void push(Stack * s, int e){
    s->elements[s->top] = e;
    s->top++;
}
```

```
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

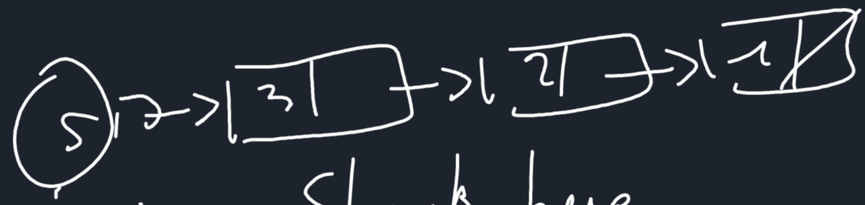
/* Stack.h */

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

$\Longleftrightarrow$

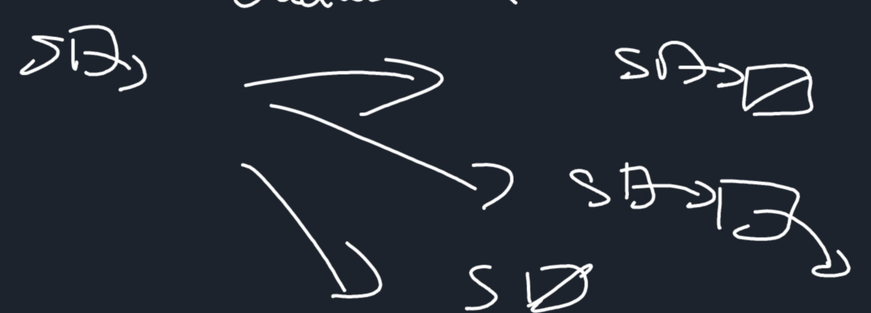
```
typedef Box * Stack;
```

same function prototypes (except createStack())



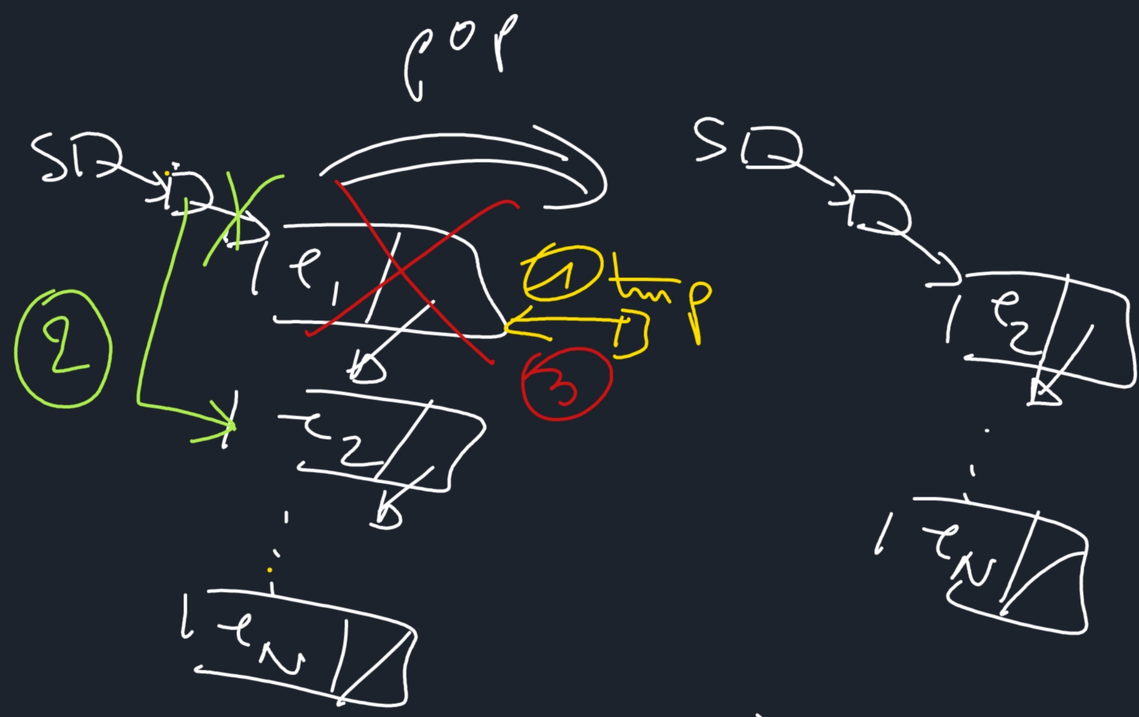
s is a Stack here, not a Stack* !

```
/* 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;
}
```

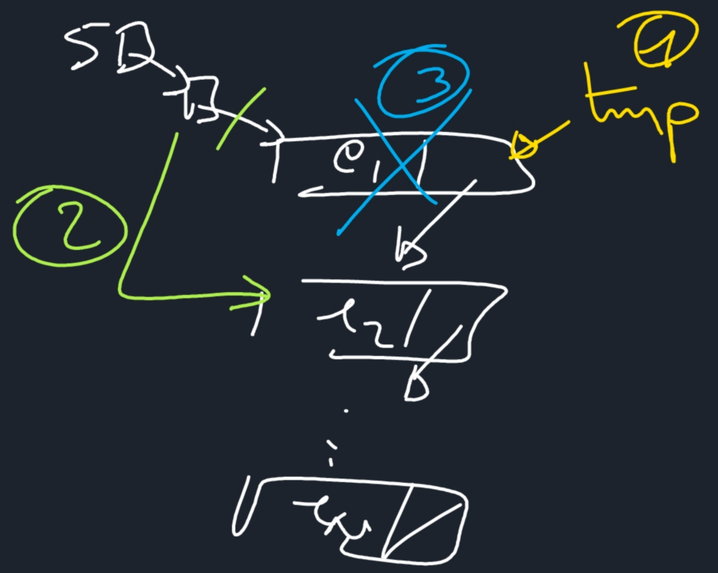
```
void pop(Stack * s){
    Stack tmp = *s;
    *s = tmp->next;
    free(tmp);
}
int isEmpty(Stack*s){
    return *s==0;
}
```




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

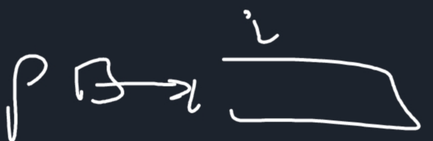
```
void destroyStack(Stack *s){
    while(!isEmpty(s))
        pop(s);
    free(s);
}
```

```
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*));
```

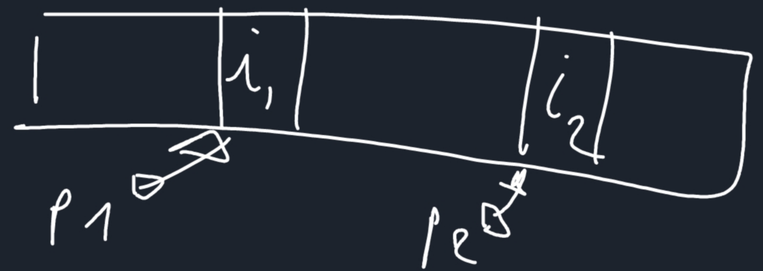
```
void sortArray(void* a, int size, int eltSize, int (*comp)(void*, void*))
{
  int l, r;
  int end = size * eltSize;
  void* tmp = malloc(eltSize);
  for(l = 0; l < end - eltSize; l += eltSize)
    for(r = l + eltSize; r < end; r += eltSize)
      if(comp(a+l, a+r) > 0) {
        memcpy(tmp, a+l,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

```
int main() {
    int i;
    int a[] = { 2, 8, 5, 7, 1, 4 };
    char* b[] = { "un coup", "du dio", "jamais", "n'abdina", "le hasard" };
    sort(a, sizeof(a)/sizeof(int), sizeof(int), compInt);
    displayArray(a, ____________);
    sort(b, sizeof(b)/sizeof(char*), sizeof(char*), compString);
    for(i=0; i<____________; i++)
        printf("%s\n", b[i]);
    return 0;
}
```

```
#include <string.h>
int strcmp(const char* s1, const char* s2);
```

- 0 if s1 == s2
- <0 if s1 before s2 in the dictionary
- >0 ——— after ———

```
int compString(void* p1, void* p2){
    return strcmp(*(char**)p1, *(char**)p2);
}
```

size($\times 10^4$)	t(sort)	t(qsort)
1	0.4	0.003
2	1.6	0.002
4	6.4	0.004
8	25.0	0.008
16		0.016
32		0.04
64		0.09
128		0.18
256	$25 \cdot 10^3$	0.34

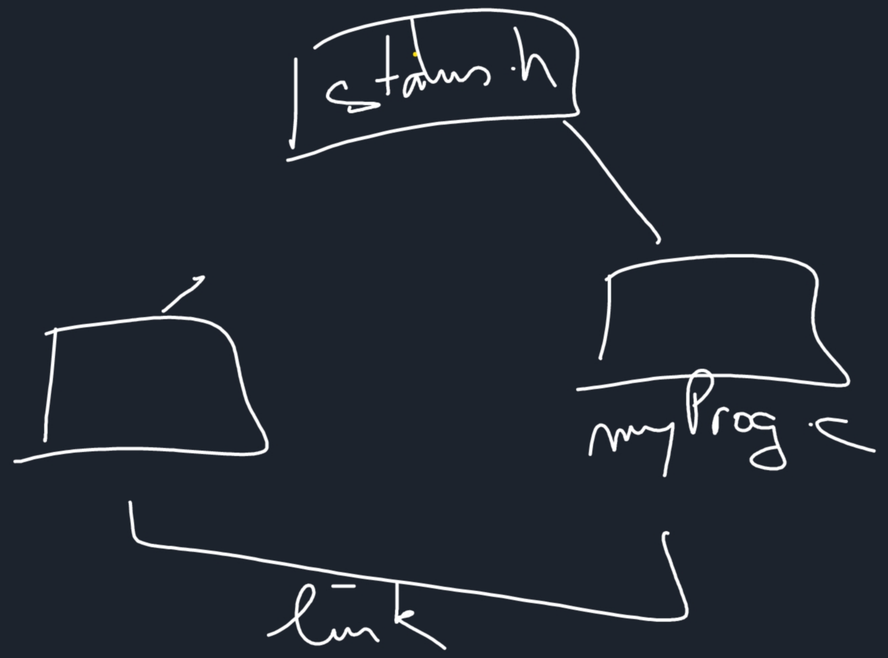
$\mathcal{O}(n^2)$ $\mathcal{O}(n \log n)$

time $/tIntArray -s 1000

Simple technique for handling errors in C

```c
/* status.c */
#include "status.h"
static char * messages[] = {
    "",
    "memory allocation error",
    "value not found",
    "empty collection",
    "can't open file",
    ...
    "unknown error"};
char * errorMessage(status s) {
    return s<ERRUNKNOWN ? msgs[s] : msgs[ERRUNKNOWN];
}
```

```c
/* status.h */
typedef enum {
    OK,
    ERRALLOC,
    ERRNFOUND,
    ERREMPTY,
    ERROPEN,
    ...
    ERRUNKNOWN } status;
char * errorMessage(status s);
```



int messages = 0;

tools.a
tools.so
tools.dll
tools.lib

convention about using status type:

void f(...); ⟶ status f(...);

type* f(...); ⟹ 0 ⇔ error, otherwise ok

type f(...); ⟶ status f(..., type* result);

ex: Stack.h

Stack * createStack(); → same

int isEmpty(Stack*s); → status push(Stack*s, int e);

void push(Stack*s, int e); → status pop(Stack*s);

void pop(Stack*s); → status top(Stack*s, int*res);

void destroyStack(Stack*s); →

int top(Stack*s);

```
e = top(s);                    echo $?
↳ st = top(s, &e);
status pushd(Stack* s, int e){
    Box* tmp=(Box*)malloc(sizeof(Box));
    if(!tmp) return ERRALLOC;
    tmp->value=e;
    tmp->next=*s;
    *s=tmp;
    return OK;
}
```

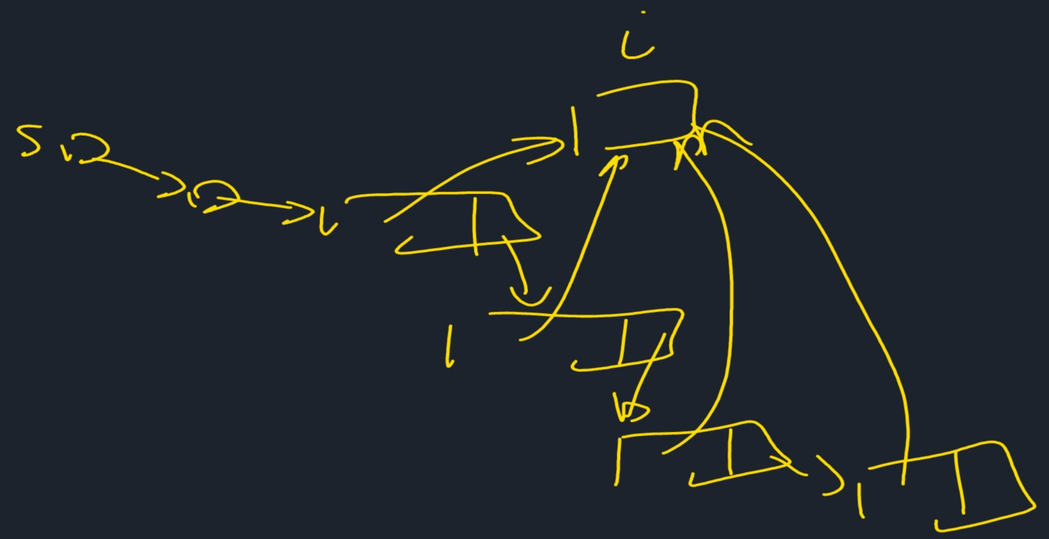
```
int top(Stack* s) {
    return (*s)->value;
}
```

```
status top(Stack* s, int* res) {
    if(!*s) return ERREMPTY;
    *res = (*s)->value;
    return OK;
}
```

#makefile

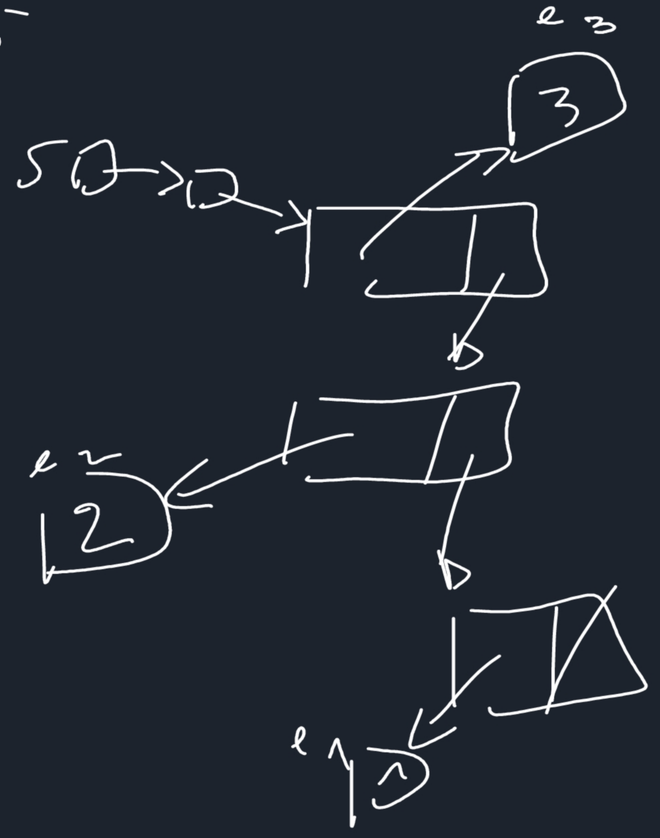
```
OBJ = status.o Stack.o tStack.o
tStack : $(OBJ)
tStack.o Stack.o : Stack.h status.h
status.o : status.h
```

```
int main() {
    int i; status st;
    Stack & s = createStack();
    if (!s) return 1;
    for (i=0; i<10; i++) {
        st = push(s,&i);
        if (st) { printf("%s\n", errorMessage(st)); return st; }
    }
    while (!isEmpty(s)) {
        st = top(s, &i);
        if (st) { printf("%s\n", errorMessage(st)); return st; }
        printf("%d\n", i);
        st = pop(s);
        if (st) { printf("%s\n", errorMessage(st)); return st; }
    }
    return 0;
}
```

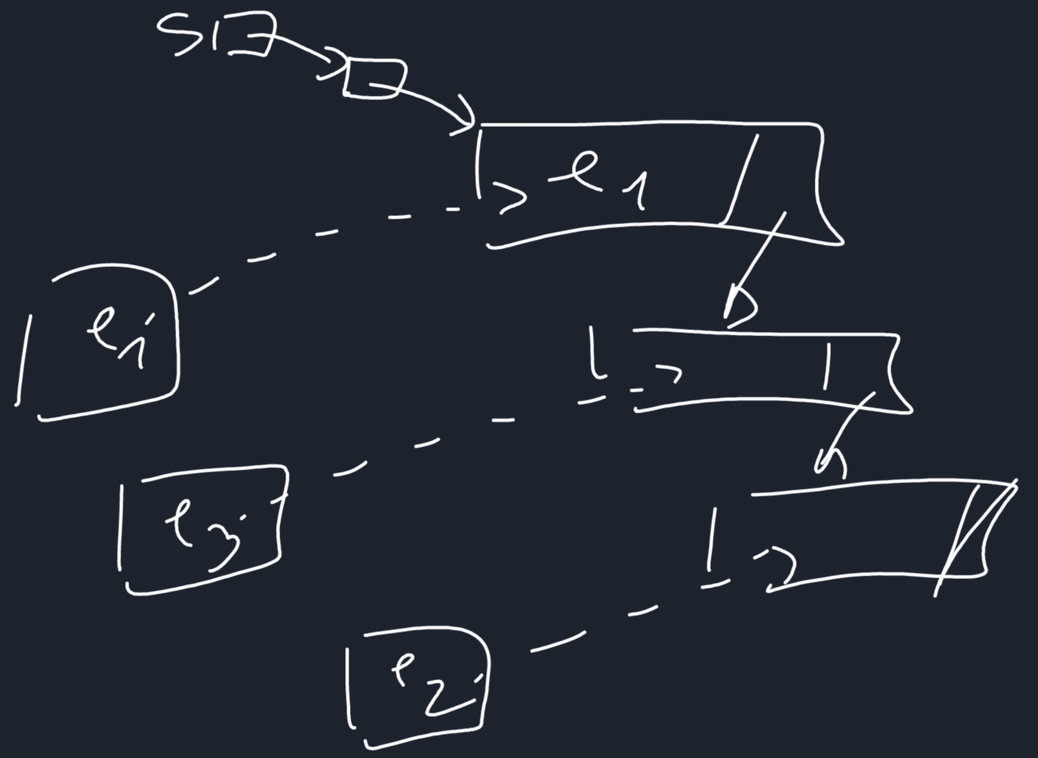


```
/* Stack.h */

typedef struct Box {
    void * value;
    struct Box* next;
} Box, *Stack;

Stack* createStack();
status push(Stack *s, void *e);
status pop(Stack *s);
int isEmpty(Stack *s);
void destroyStack(Stack *s);
status top(Stack *s, void *no);
```

the stack does not "own" its element



if stack "owns" its elt:

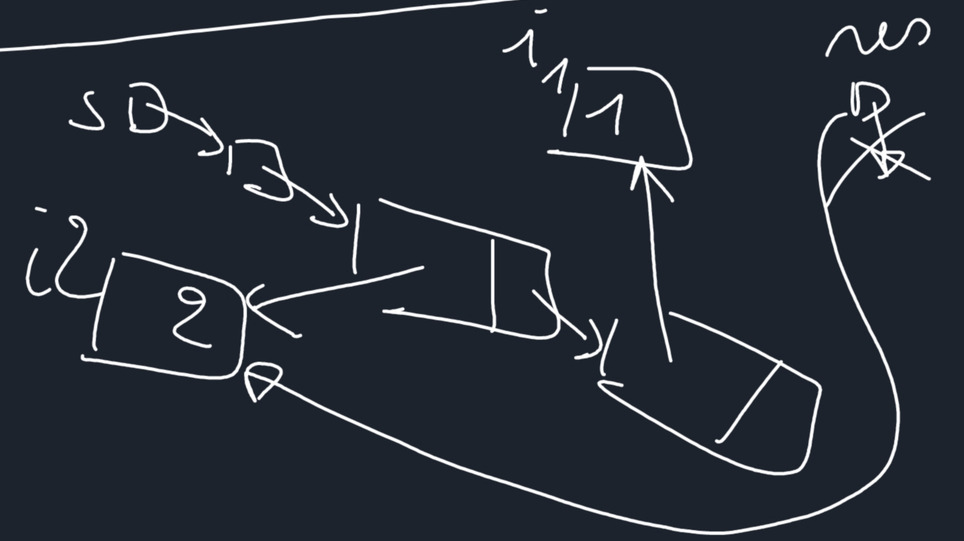


```
/* in Stack.c */
status push(Stack *s, void* e){
    Box* tmp=(Box*)malloc(sizeof(Box));
    if(!tmp) return ERRALLOC;
    tmp->value=e;
    tmp->next=*s;
    *s=tmp;
    return ok;
}
```

test: define | a stack
2 integers,
push them to stack,
and while stack not empty
- print top
- pop stack

```
status top(Stack* s, void* res){
    if(!*s) return ERREMPTY;
    *(void**)res = (*s)->value;
    return Ok;
}
```

```
int main(){
    int i1 = 1;
    int i2 = 2;
    Stack *s = createStack();
    push(s, &i1);
    push(s, &i2);
    while(!isEmpty(s)){
        int *res;
        top(s, &res);
        printf("%d\n", *res);
        pop(s);
    }destroyStack(s);
    return 0;
}
```



expected:

2

1

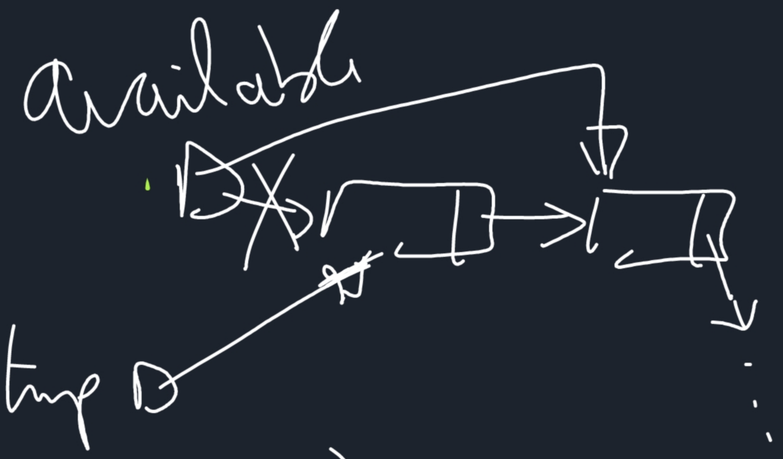
deamp();

/*stack.c*/

static Box * available = 0;

/* in Stack.c */

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

    } /* same */
```



```
Status pop(Stack *s){
    Box *tmp;
    if(!*s) return ERREMPTY;
    tmp = (*s)->next;
    (*s)->next = available;
    available = *s;
    *s = tmp;
    return OK;
}
```