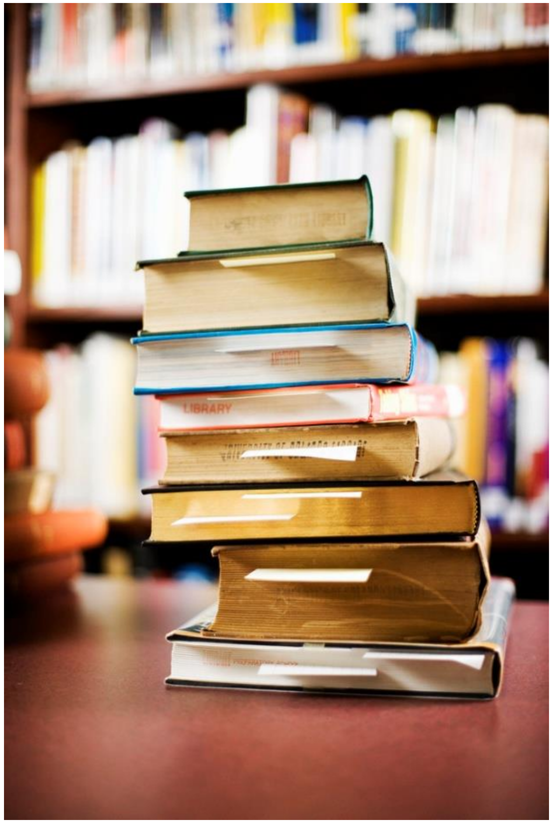
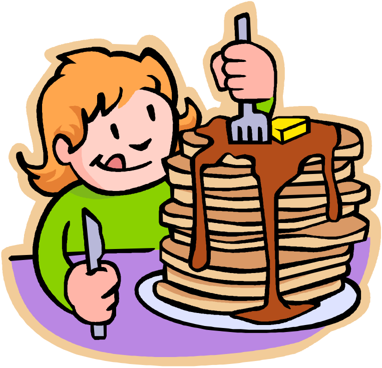
Stacks with linked nodes

Stack

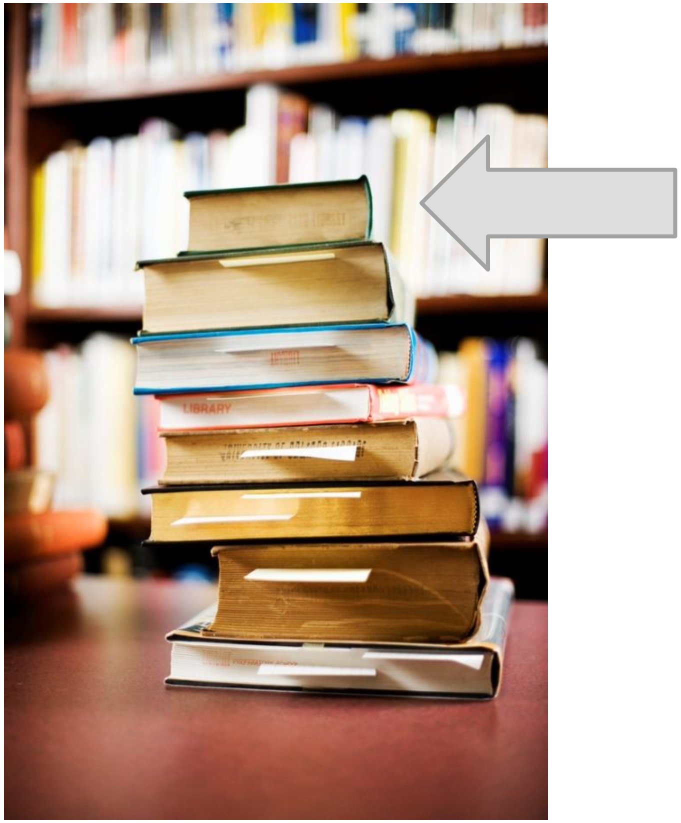
- examples of stacks





Top of Stack

- top of the stack

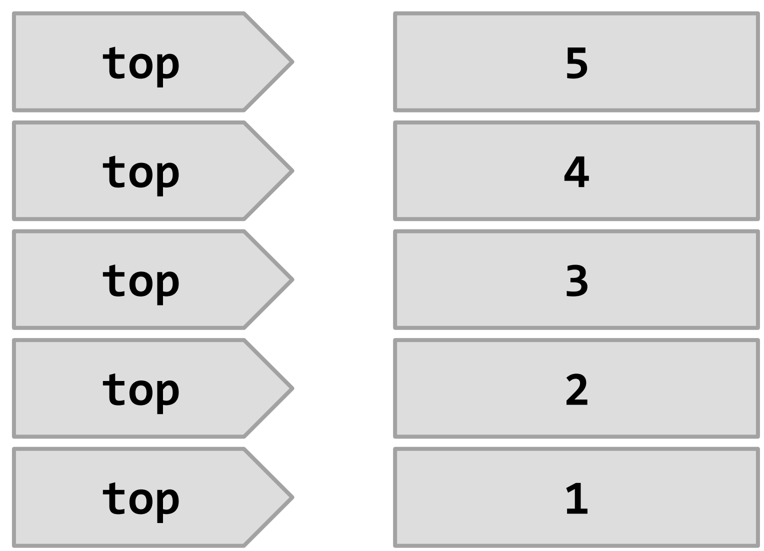


Stack Operations

- classically, stacks only support two operations
  1. push
     - add to the top of the stack
  2. pop
     - remove from the top of the stack
     - throws an exception if there is nothing on the stack

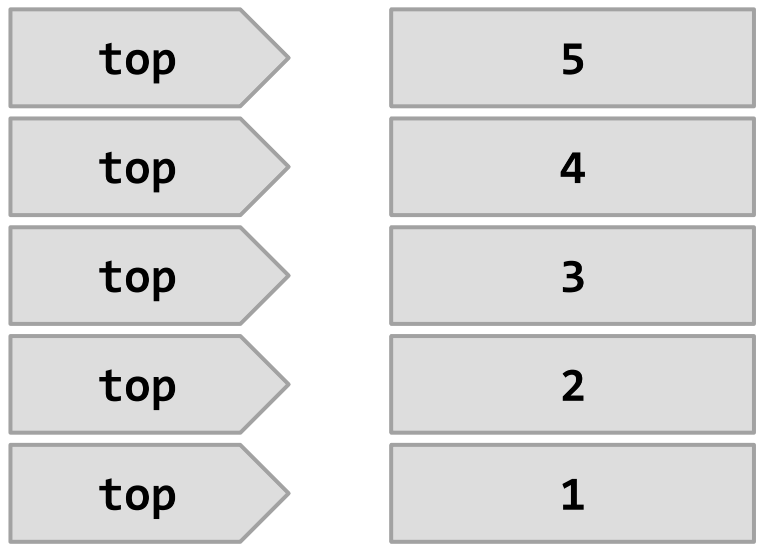
Push

1. `push(st, 1);`
2. `push(st, 2);`
3. `push(st, 3);`
4. `push(st, 4);`
5. `push(st, 5);`



Pop

1. `int x = pop(st);`
2. `x = pop(st);`
3. `x = pop(st);`
4. `x = pop(st);`
5. `x = pop(st);`



Applications

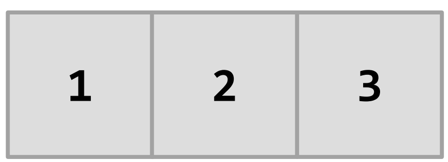
- stacks are used widely in computer science and computer engineering
  - undo/redo
  - widely used in parsing
  - a call stack is used to store information about the active functions in a C program
  - convert a recursive function into a non-recursive one

Example: Reversing an array

- a silly and inefficient way to reverse an array is to use a stack

Don't do this

```
void reverse(size_t n, int arr[n]) {
    lstack *st = lstack_init();
    for (int i = 0; i < n; i++) {
        lstack_push(st, arr[i]);
    }
    for (int i = 0; i < n; i++) {
        arr[i] = lstack_pop(st);
    }
    lstack_free(st);
}
```



`arr`

stack **`st`**


1
2
3
arr
1
stack st


1
2
3
arr
2
1
stack st

`arr` stack `st`


3
2
3
arr
3
2
1
stack st

`arr`

stack **`st`**


3
2
1
arr
1
stack st

Complexity of push and pop

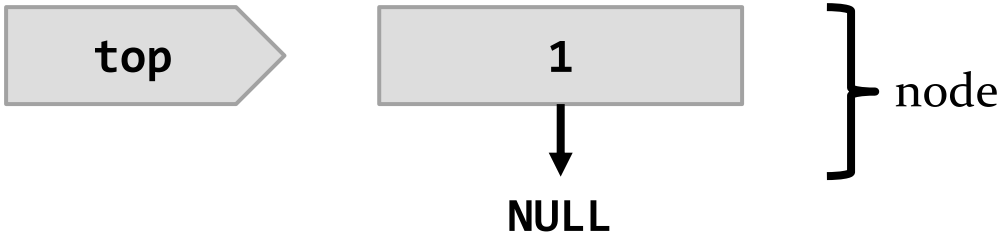
- for an array-based stack or an array list-based stack, the complexity of the:
  - **pop** operation is always in $O(1)$
  - **push** operation is in $O(N)$ when the stack size equals the array capacity
- we can guarantee a **push** operation having $O(1)$ complexity by changing how we store elements

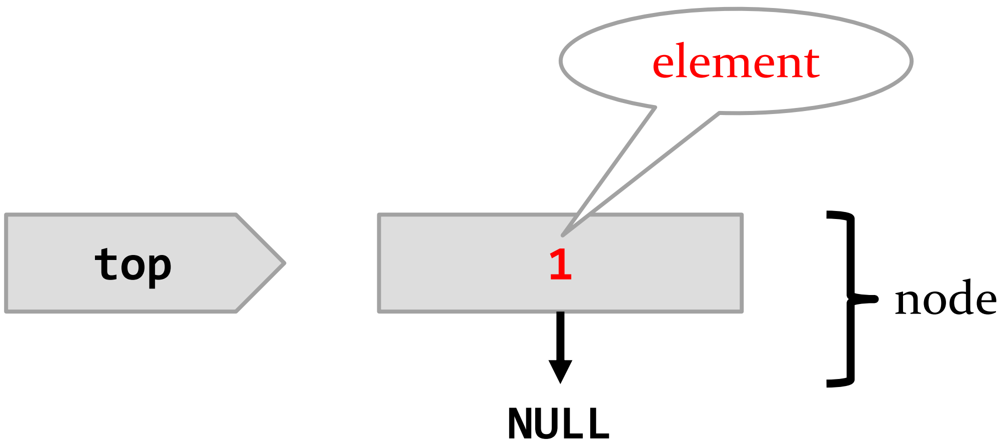
Nodes

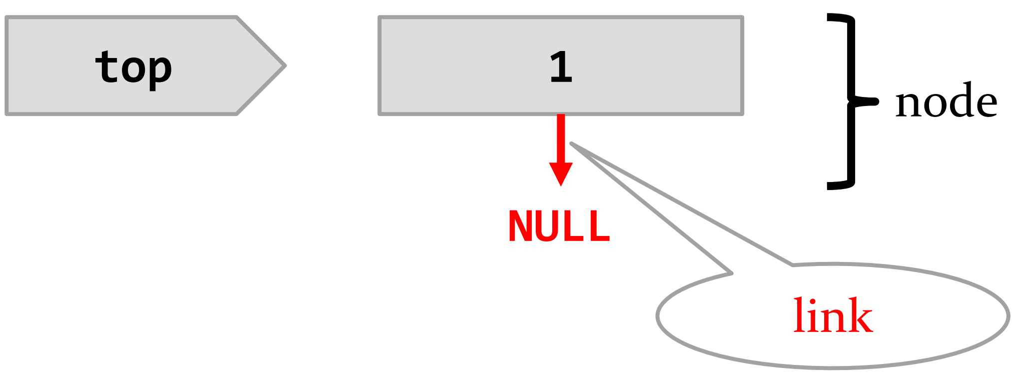
- many data structures that represent a collection of elements can be implemented using *nodes*
  - linked lists (CISC124/CISC235)
  - trees (CISC235)
  - graphs (CISC235 (sometimes), CISC365, MATH401)
- in general, a node stores:
  - an element, or a pointer to an element
  - pointers to zero or more other nodes
    - these pointers are often called *links*

Nodes for a stack

- a stack is a linear collection of elements
  - elements are arranged in a sequence starting from the top element
  - each element is connected to the next element deeper in the stack
- a node in a stack stores:
  - an element, or a pointer to an element
  - a pointer to the next node deeper in the stack

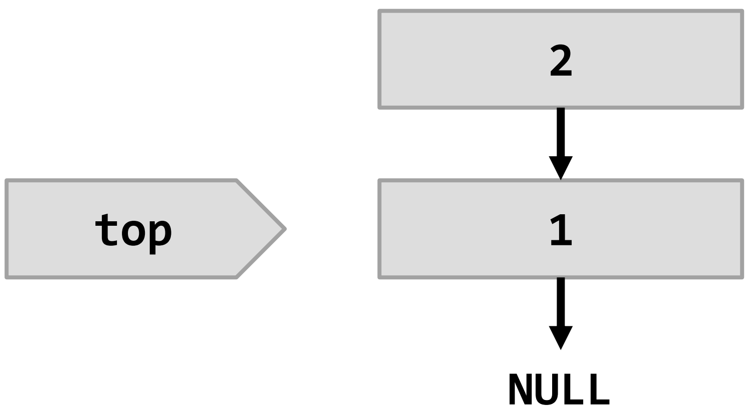

top
1
NULL
node

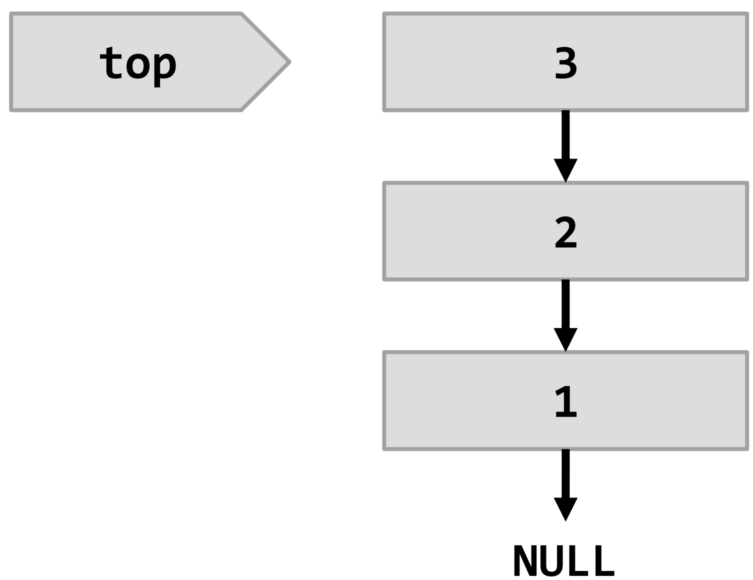

element
top
1
node
NULL


top
1
node
NULL
link

pushing an element onto the stack:

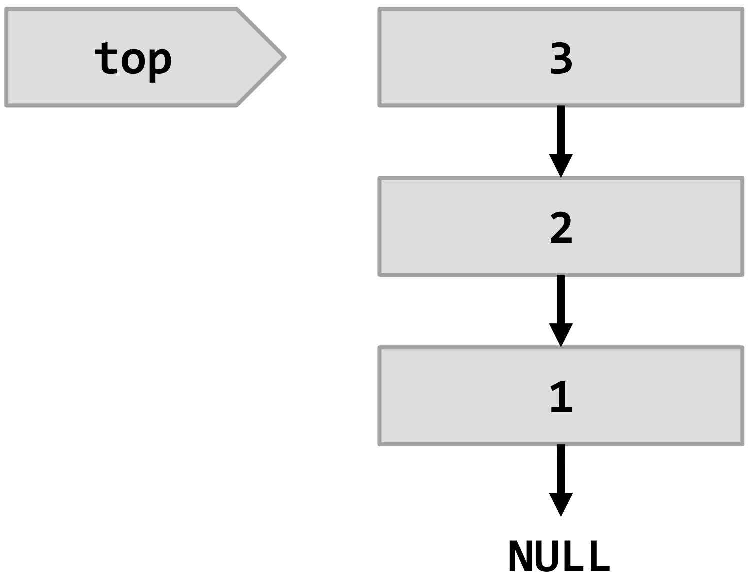
1. make a new node to hold the element pushed onto the stack
2. set the link of the new node to point to the current top node
3. set top to refer to the new node
4. add one to the stack size




top
3
2
1
NULL

popping an element off the stack:

1. copy the element stored in the current top node
2. set top to point to the next node deeper in the stack
3. subtract one from the stack size
4. return the copy obtained in Step 1 to return the popped element



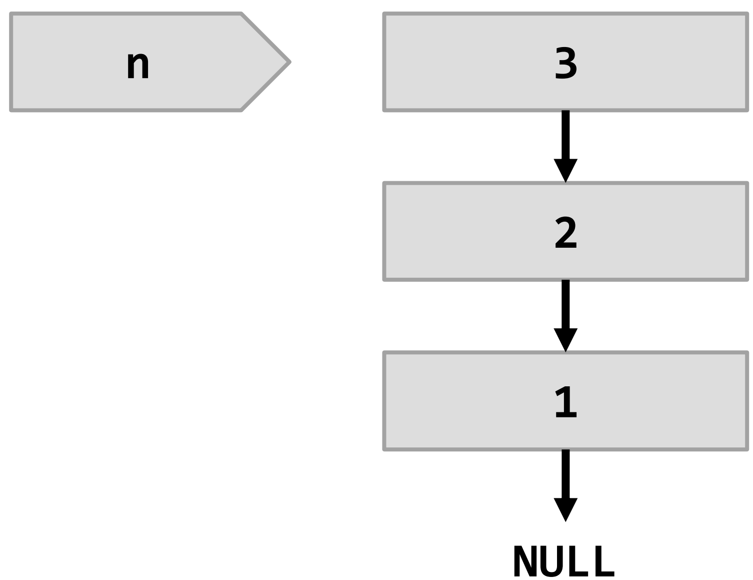
See **`lstack.h`**, **`lstack.c`**, and **`stack_demo.c`**

Iterating over a linked sequence

- printing a stack requires iterating over the sequence of nodes
  - iterating over a sequence of linked nodes is a very common operation when working with linked structures
- observe that the end of the sequence is indicated by a null next link
  - starting at the first node of the sequence, we can simply follow each next link to the next node until we reach a null node

General iteration pattern

```
node *n = startingNode;
while (n) {
    // do something with n here
    //
    // then advance to next node
    n = n->next;
}
// DO NOT dereference n here!!!
```


n
3
2
1
NULL

```c
void lstack_print(const lstack *s) {
    printf("TOP, ");
    node *n = s->top;
    while (n) {
        printf("%d, ", n->elem);
        n = n->next;
    }
    printf("BOTTOM\n");
}
```

Structural recursion

- the **node** struct is an example of recursion of structure
  - every **node** has a pointer to a **node** which has a pointer to a **node** which has a pointer to a **node**, and so on
  - every **node** can be viewed as being the top node of a stack
- the recursive structure makes it easy to write recursive algorithms
  - e.g., consider `lstack_print`

```c
void lstack_print(const lstack *s) {
    printf("TOP, ");
    print_rec(s->top);
    printf("BOTTOM\n");
}

void print_rec(const node *n) {
    // base case
    if (!n) {
        return;
    }
    // n not NULL, recursive case
    // print the element in n
    printf("%d, ", n->elem);
    // then print the remaining elements
    print_rec(n->next);
}
```