

# Abstract Data Types and Stacks

CSE 2320 – Algorithms and Data Structures  
Vassilis Athitsos and Alexandra Stefan  
University of Texas at Arlington

Last updated: 2/25/2016

# Abstract Datatypes (ADT)

- ADT (Abstract Data Type)
  - is a *data type* (a set of values and a collection of operations on those values) that can only be accessed through an *interface*.
- Client is a program that uses an ADT.
  - E.g.: walmart.c
  - Will have: `#include "list.h"`
- Implementation is a program that specifies the data type and the operations for it.
  - E.g.: list.c
  - Function definitions and struct definitions (or class definitions)
- Interface a list of operations available for that datatype.
  - E.g.: list.h (notice, a header file, not a c file)
  - It will contain headers of functions and typedef for data types,
  - It is *opaque*: the client can not see the implementation through the interface.

# Generalized Queues

- A generalized queue is an abstract data type that stores a set of objects.
  - Let's use **Item** to denote the data type of each object.
- The fundamental operations that such a queue must support are:
  - **void insert(Queue q, Item x):** adds object **x** to set **q**.
  - **Item delete(Queue q):** choose an object **x**, remove that object from **q**, and return it to the calling function.
  - create – creates a queue
  - destroy – destroys a queue
  - join – joins two queues

# Generalized Queues

- Basic operations:
  - void insert(Queue q, Item x)
  - Item delete(Queue q)
- **delete** must choose what item to delete.
  - Last inserted -> Stack / Pushdown Stack / LIFO
  - First inserted -> FIFO Queue
  - Random item.
  - Item with the smallest key (if each item contains a **key**).
    - > Priority Queue (Heap)

# Push and Pop

- The pushdown stack supports **insert** and **delete** as follows:
  - **insert push**: This is what we call the insert operation when we talk about pushdown stacks. It puts an item "on top of the stack".
  - **delete pop**: This is what we call the delete operation when we talk about pushdown stacks. It removes the item that was on top of the stack (the last item to be pushed, among all items still on the stack).

# Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- pop()

15

# Examples of Push and Pop

- push(15)
- **push(20)**
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- pop()

20
15

# Examples of Push and Pop

- push(15)
- push(20)
- **pop()** – returns 20
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- pop()

15



# Examples of Push and Pop

- push(15)
- push(20)
- pop()
- **push(30)**
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- pop()

30
15

# Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- **push(7)**
- push(25)
- pop()
- push(12)
- pop()
- pop()

7
30
15

# Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- **push(25)**
- pop()
- push(12)
- pop()
- pop()

25
7
30
15

# Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- **pop()** – returns 25
- push(12)
- pop()
- pop()

7
30
15

# Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- **push(12)**
- pop()
- pop()

12
7
30
15

# Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- **pop()** – returns 12
- pop()

7
30
15

# Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- **pop()** – returns 7

30
15

# Stack Applications

- Function execution in computer programs:
  - when a function is called, it enters the **calling stack**. The function that leaves the calling stack is always the last one that entered (among functions still in the stack).
- Interpretation and evaluation of symbolic expressions:
  - evaluate expressions like  $(5+2)*(12-3)$ , or
  - parse C code (as a first step in the compilation process).
- Search methods.
  - traverse or search a graph



# Implementing Stacks

- A stack can be implemented using:
  - lists or
  - arrays
- Both implementations are fairly straightforward.

# List-Based Stacks

- **List-based** implementation:
  - What is a stack?
    - A stack is essentially a list.
  - **push(stack, item)**
    - How? :
    - $O(??)$
  - **pop(stack)**
    - How? :
    - $O(??)$

# List-Based Stacks

- List-based implementation:
  - What is a stack?
    - A stack is essentially a list.
  - **push(stack, item)**
    - How? :
    - $O(1)$  – ideally !!!! (frequent operation for this data structure)
  - **pop(stack)**
    - How? :
    - $O(1)$  – ideally !!!! (frequent operation for this data structure)
- What type of insert and remove are fast for lists?
  - How many 'ways' can we insert in a list?



# List-Based Stacks

- List-based implementation:
  - What is a stack?
    - A stack is essentially a list.
  - **push(stack, item)**
    - How? : inserts that item at the **beginning** of the list.
    - $O(1)$
  - **pop(stack)**
    - How? : removes (and returns) the item at the **beginning** of the list.
    - $O(1)$



# Implementation Code

- See files posted on course website:
  - **stack.h**: defines the public interface.
  - **stack\_list.c**: defines stacks using lists.
  - **stack\_array.c**: defines stacks using arrays.

# The Stack Interface

- See file **stack.h** posted on the course website.

```
stack newStack(int mx_sz);  
void destroyStack(stack s);  
void push(stack s, Item data);  
Item pop(stack s);      //NOTE: type Item, not link  
Item isEmpty(stack s);
```

# Defining Stacks

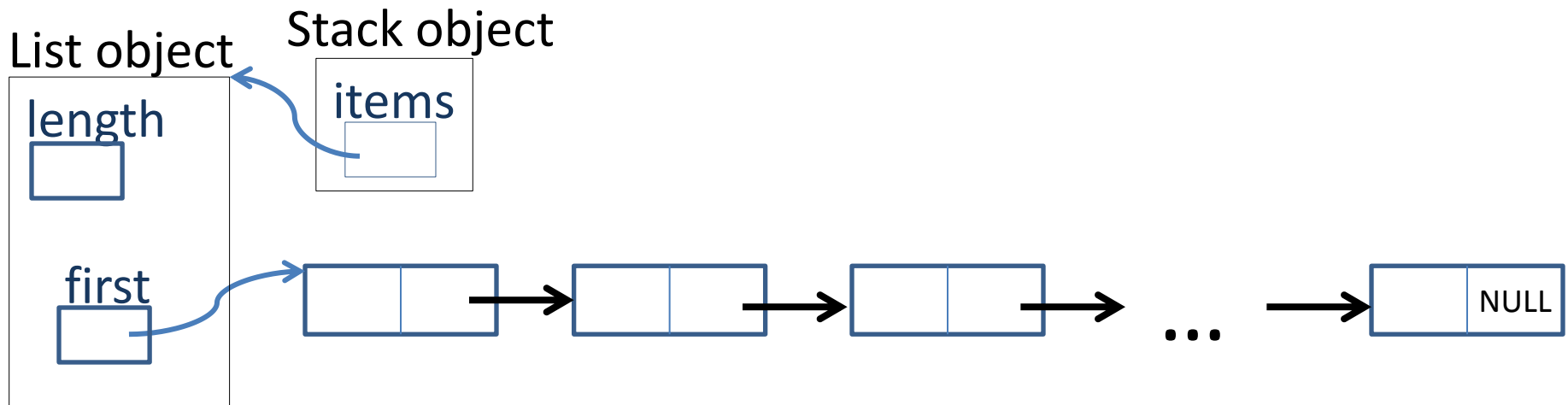
```
typedef struct stack_struct * stack;
```

```
struct stack_struct  
{  
    ???;  
};
```

# Defining Stacks

```
typedef struct stack_struct * stack;
```

```
struct stack_struct  
{  
    list items;  
};
```

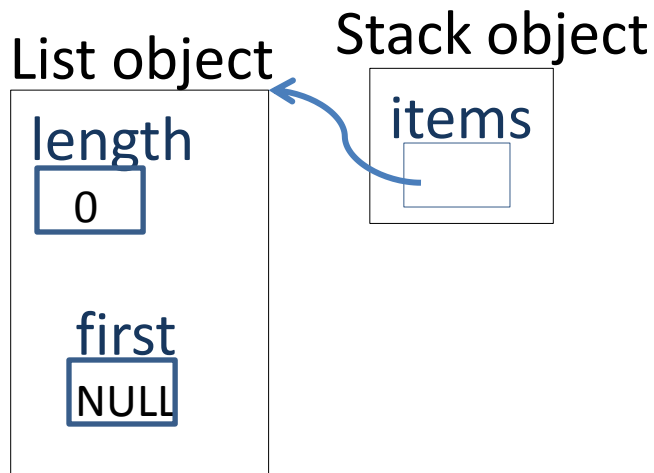




# Example

```
typedef struct stack_struct * stack;
```

```
struct stack_struct  
{  
    list items;  
};
```



## Empty stack.

We will insert values  
in order:

20

7

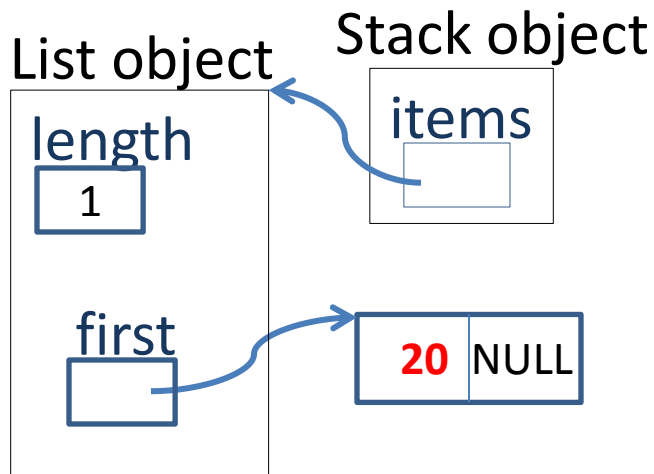
15

pop()

# Defining Stacks

```
typedef struct stack_struct * stack;
```

```
struct stack_struct  
{  
    list items;  
};
```



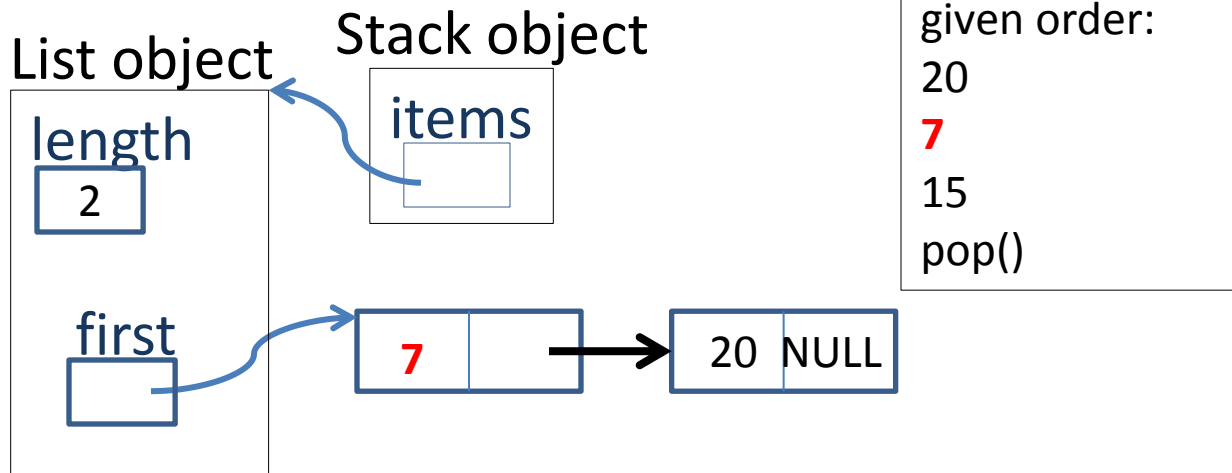
Insert values in  
given order:

**20**  
7  
15  
pop()

# Defining Stacks

```
typedef struct stack_struct * stack;
```

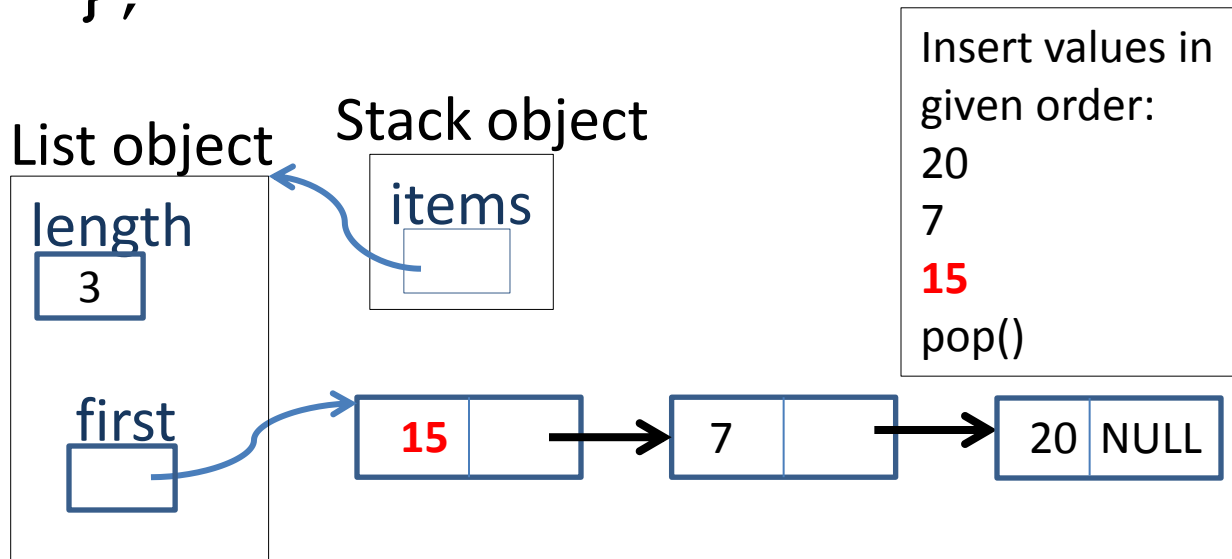
```
struct stack_struct  
{  
    list items;  
};
```



# Defining Stacks

```
typedef struct stack_struct * stack;
```

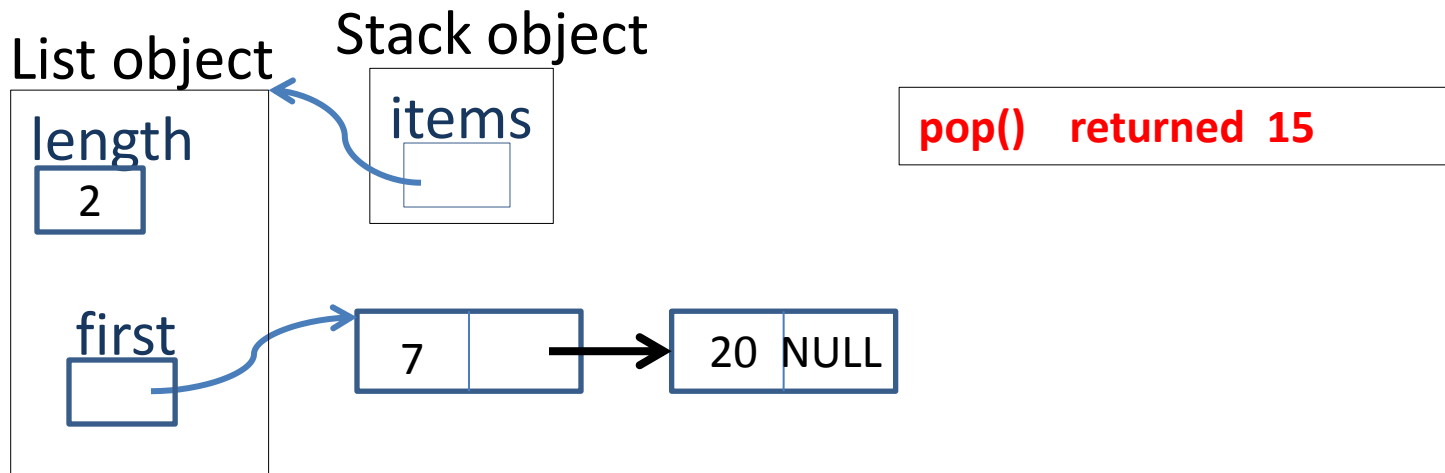
```
struct stack_struct  
{  
    list items;  
};
```



# Defining Stacks

```
typedef struct stack_struct * stack;
```

```
struct stack_struct  
{  
    list items;  
};
```



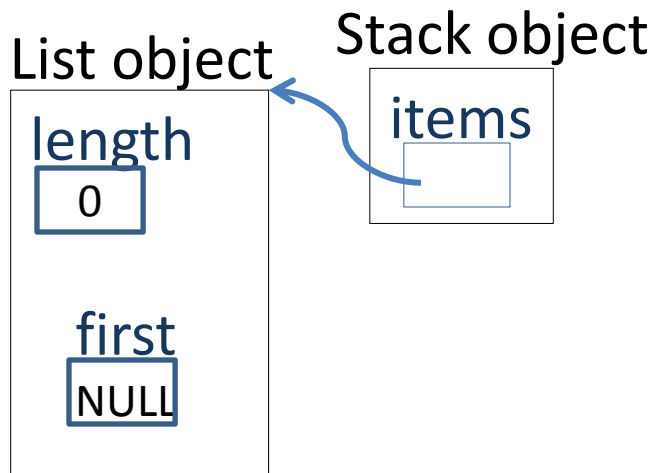
# Creating a New Stack

```
typedef struct stack_struct * stack;  
struct stack_struct  
{  
    list items;  
};
```

```
stack newStack(int mx_sz)  
{  
    ???  
}
```

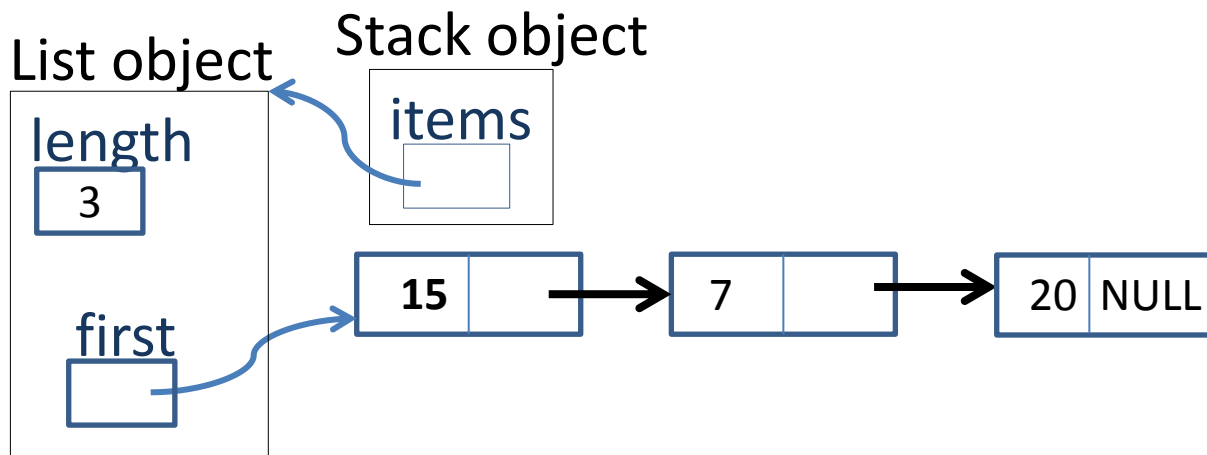
# Creating a New Stack

```
typedef struct stack_struct * stack;
struct stack_struct {
    list items;
};
// mx_sz-needed for compatibility with array implementation
stack newStack(int mx_sz) {
    stack result = (stack)malloc(sizeof(*result));
    result->items = newList(); //mem alloc
    return result;
}
```



# Destroying a Stack

```
typedef struct stack_struct * stack;  
struct stack_struct {  
    list items;  
};  
void destroyStack(stack s) {  
    destroyList(s->items);  
    free(s);  
}
```





# Pushing an Item

```
typedef struct stack_struct * stack;
struct stack_struct {
    list items;
};

void push(stack s, Item data) {
    link L = newLink(data, NULL);
    insertAtBeginning(s->items, L);
}
```

# Popping an Item

```
typedef struct stack_struct * stack;  
struct stack_struct {  
    list items;  
};
```

```
Item pop(stack s) {  
    link top = removeFirst(s->items);  
    return linkItem(top);  
}
```

What is wrong with this definition of **pop**?

# Popping an Item

```
typedef struct stack_struct * stack;  
struct stack_struct {  
    list items;  
};
```

```
Item pop(stack s) {  
    link top = removeFirst(s->items);  
    return linkItem(top);  
}
```

What is wrong with this definition of **pop**? **Memory leak!!!**

# Popping an Item

```
typedef struct stack_struct * stack;
struct stack_struct {
    list items;
};
```

```
Item pop(stack s) {
    if (isStackEmpty(s))
        ERROR. No item to remove!!!
    link top = removeFirst(s->items);
    Item item = linkItem(top);
    free(top);
    return item;
}
```

# WHY use a Stack ADT?

- Why should we have a Stack interface, when all we do is use list operations? Why not use a list directly?
  - **Protection:** from performing unwanted operations (e.g. an insert at a random position in the list)
  - **Flexibility:**
    - To modify the current implementation
    - To use another stack implementation
  - **Isolates the dependency** of the Stack implementation on the list interface: if the list INTERFACE (.h file) is modified we will only need to go and change the STACK implementation (.c file), not all the lines of code when a stack operation is done in all the client programs.
  - It makes the **stack behavior explicit**: what we can do and we can not do with a stack.

# “Explicitly, why is this better than just using a list where I only insert and delete from the beginning?”

- Keep in mind that your **goal here is to implement a stack datatype, for others to use**, not for your own one-time usage.
- Directly providing a list, gives access to operations not allowed for a stack like reversing the list, or removing any item in it.
- Any client code that includes the stack.h file, can only use the functions provided in that file so they can only call: pop, push, isEmpty, newStack, destroyStack. They can not reverse the list because they do not have access to the stack\_struct definition so they can not get the list object.
- A function in the stack\_list.cpp file can call any list function but that is the point where you focus on implementing a stack and so you should not do operations that are not allowed. Notice that even in this file, you cannot access fields of the list struct directly. It has to go through function calls (e.g. you cannot write *my\_list->length*, but you can write *getLength(my\_list)* ). The stack\_list.cpp is a client for list.cpp and so it does not have access to the list representation.

# Array-Based Implementation of Stacks

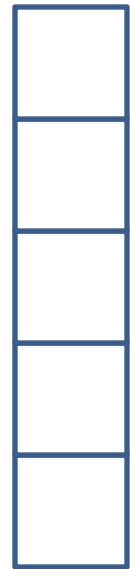
# Array-based Stacks

- **Array-based** implementation:
  - What is a stack? What will hold the data of the stack?
  - **push(stack, item)**
    - How? :
    - $O(1)$  - can we get this?
  - **pop(stack)**
    - How? :
    - $O(1)$  - can we get this?

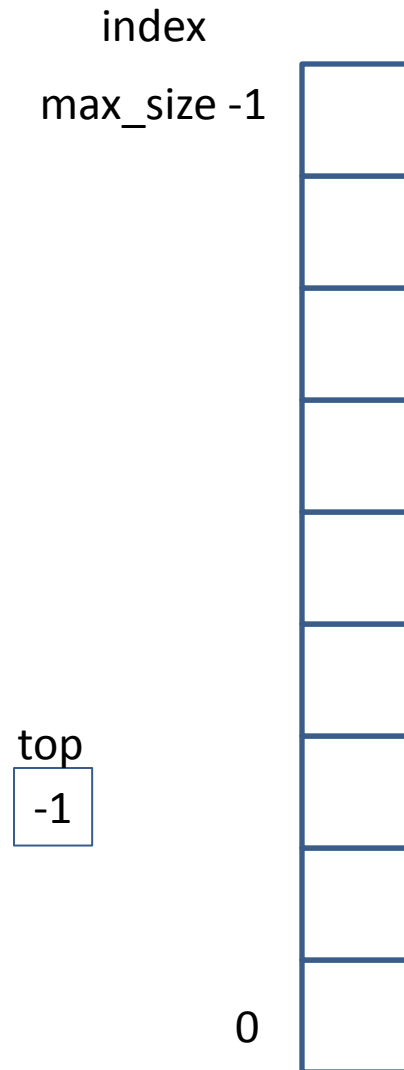


# Array-based Stacks

- **Array-based** implementation:
  - What is a stack? What will hold the data of the stack?
    - An array.
  - **push(stack, item)**
    - How? : 'insert' at the end of the array.
    - $O(1)$  - Yes
  - **pop(stack)**
    - How? : 'remove' from the end of the array.
    - $O(1)$  - Yes
- See `stack_array.c`

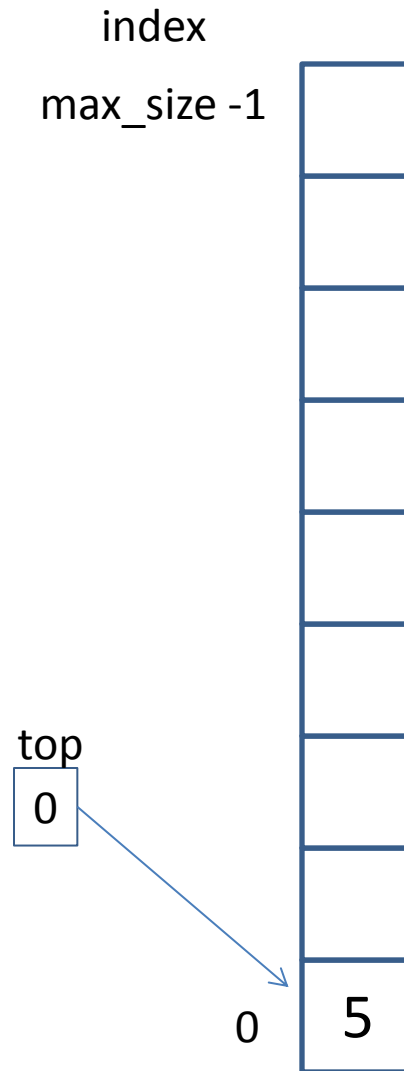


# Array-based Stacks



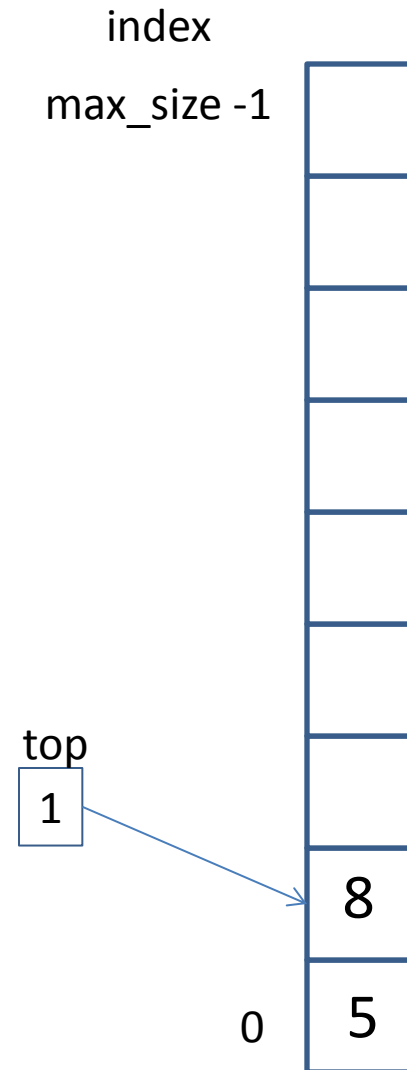
# Array-based Stacks

push(5)



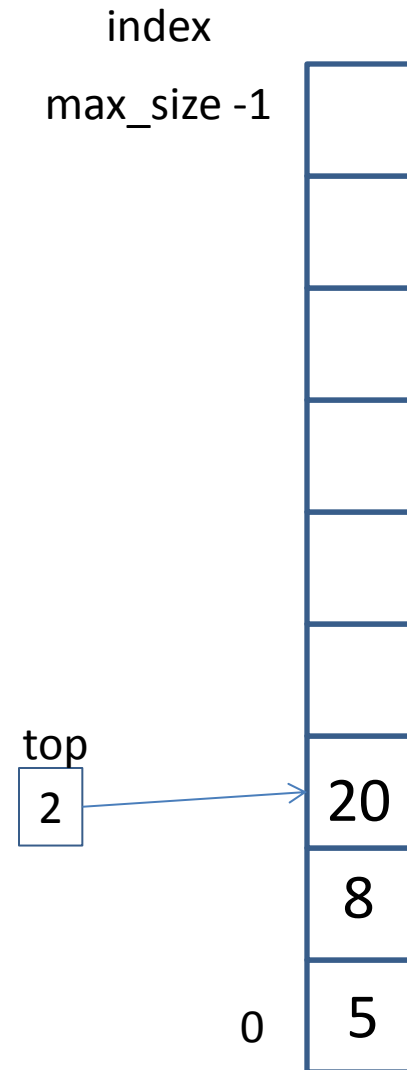
# Array-based Stacks

push(5)  
push(8)



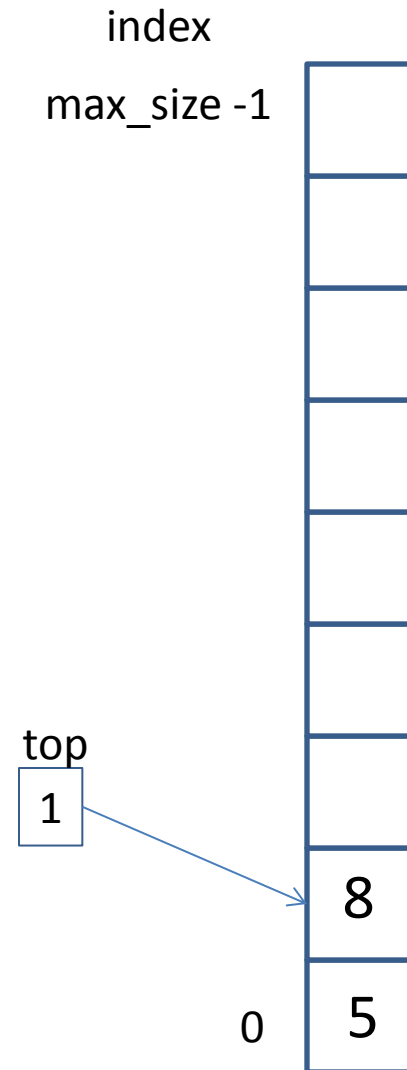
# Array-based Stacks

push(5)  
push(8)  
push(20)



# Array-based Stacks

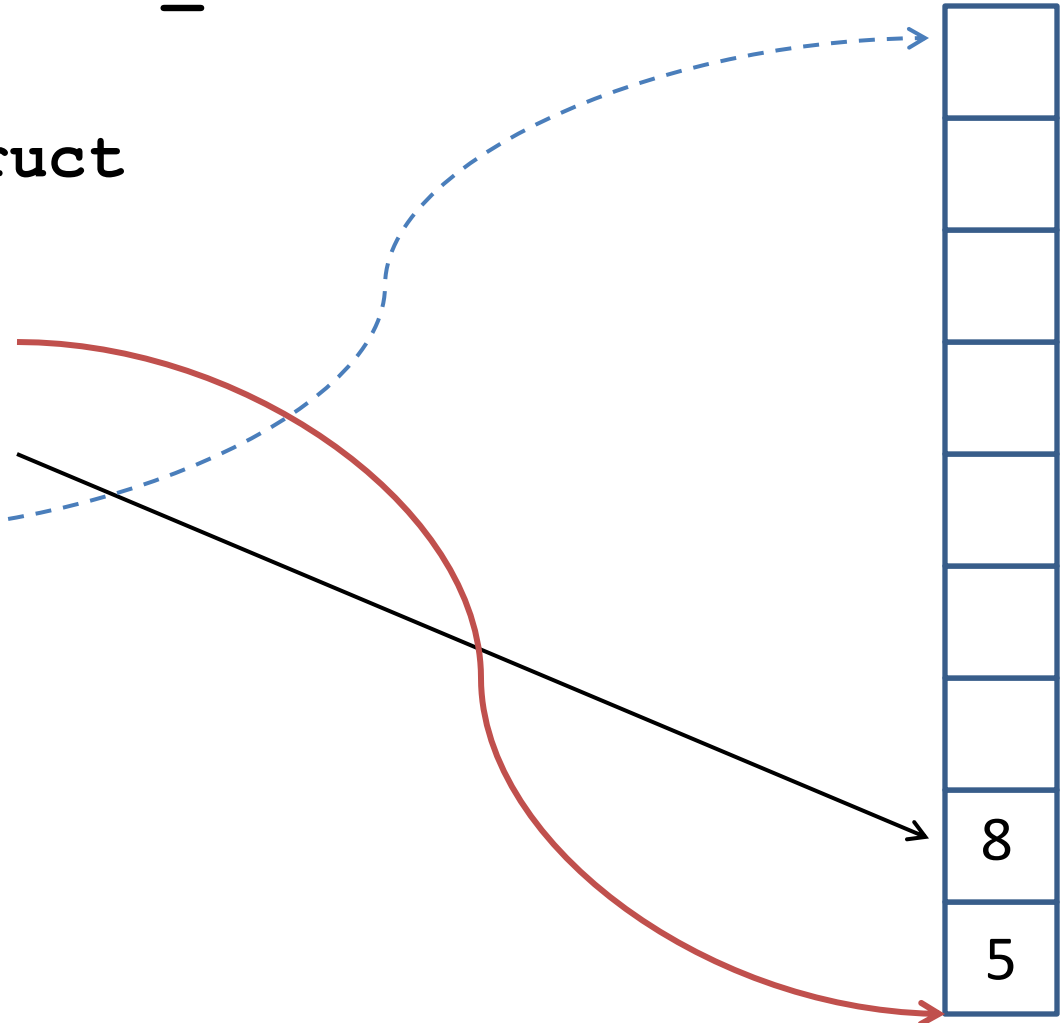
push(5)  
push(8)  
push(20)  
pop()



# Defining Stacks Using Arrays

```
typedef struct stack_struct * stack;
```

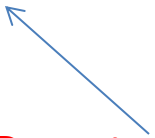
```
struct stack_struct  
{  
    Item * items;  
    int top;  
    int max_size;  
};
```



# Creating a New Stack

```
struct stack_struct
{
    Item * items;
    int top;
    int max_size;
};
```

```
stack newStack(int mx_sz)
{
    stack result = (stack)malloc(sizeof(*result));
    result->items = (Item*)malloc(mx_sz * sizeof(Item));
    result->max_size = mx_sz;
    result->top = -1;
    return result;
}
```



Do not use a array for items (e.g. items[100])!  
See the Victim-TAB example showing the  
difference between Stack and Heap  
(in memory allocation)



# Destroying a Stack

```
struct stack_struct
{
    Item * items;
    int top;
    int max_size;
};
```

```
void destroyStack(stack s)
{
    free(s->items); // s->items is an array
    free(s);
}
```

# Pushing an Item

```
struct stack_struct
{
    Item * items;
    int top;
    int max_size;
};
```

```
void push(stack s, Item data)
{
    if (s->top == s->max_size - 1)
        ERROR. No more room ( the array is full)!!!
        return;

    s->top = s->top + 1;
    s->items[s->top] = data;
}
```

# Popping an Item

```
struct stack_struct  
{  Item * items;  
    int top;  
    int max_size;  
};
```

```
??? pop(stack s)  
{  
    ???  
}
```

# Popping an Item

```
struct stack_struct
{
    Item * items;
    int top;
    int max_size;
};
```

```
Item pop(stack s)
{
    if (isStackEmpty(s))
        //ERROR. No data to pop!!!
        return 0;
    Item item = s->items[s->top];
    s->top = s->top - 1;
    return item;
}
```

# Exercises

- Letter/digit means push
  - \* means pop
1. Given sequence of operations, show the stack:
    1. THI\*\*S\*I\*\*S\*A\*NEXAM\*\*P\*\*L\*\*E\*\*
    2. THI\*\*S\*I\*\*S\***\***A\*NEXAM\*\*P\*\*L\*\*E\*\* (error)
  2. Given input and output sequence, show the push & pop operations
    1. Input: INSANE,
      1. Output: SANENI, Operations Sequence:
      2. Output: **ANS**INE, Operations Sequence: (error)