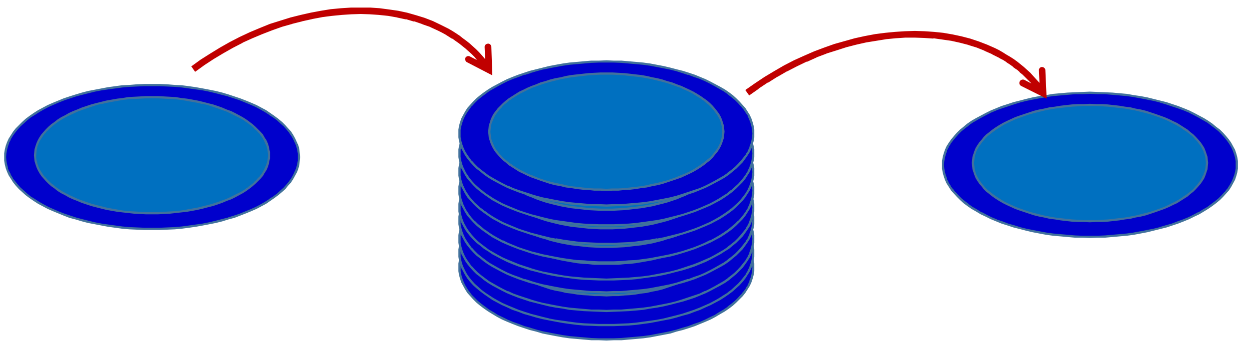
CSE 105:
Data Structures and Algorithms-I
(Part 2)

Instructor

Dr Md Monirul Islam

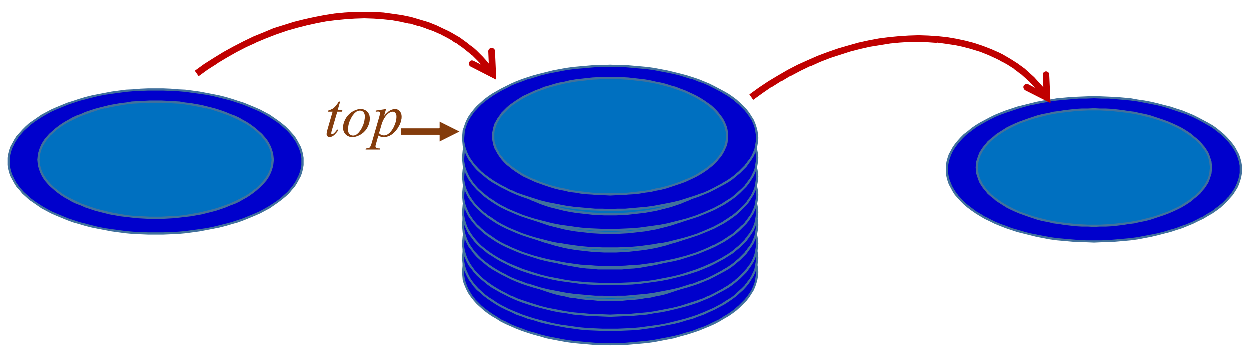
Stack Data Structure



A stack of plates

Stack Data Structure

last-in–first-out (LIFO)

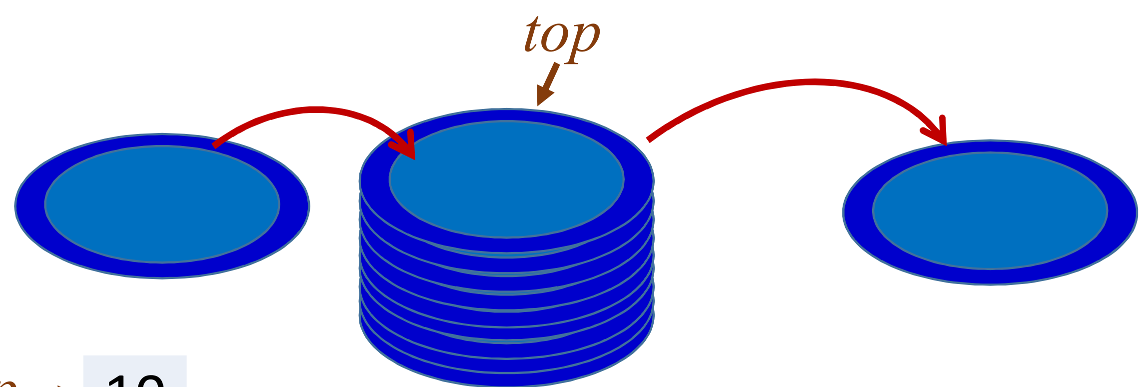


Stack Data Structure

last-in–first-out (LIFO)

Restricted form of list:

Insert and remove only at front (top) of list

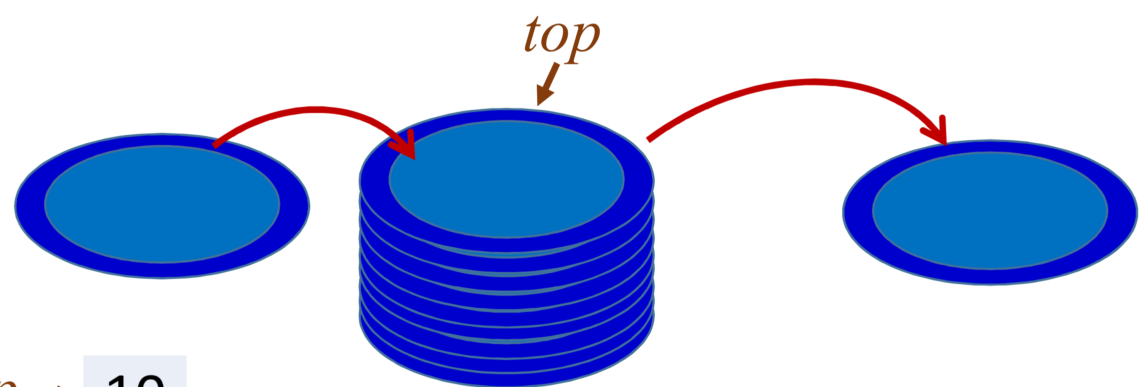


Stack Data Structure

last-in–first-out (LIFO)

Restricted form of list:

Insert and remove only
at front (top) of list



Notation:
Insert: PUSH
Remove: POP
Only accessible element
is called TOP

Stack ADT

Operations
push (*element*);
element pop();
element top();

Stack ADT

Operations
push (*element*);
element pop();
element top();

Implementations:
Array based
Link linked based

Stack ADT

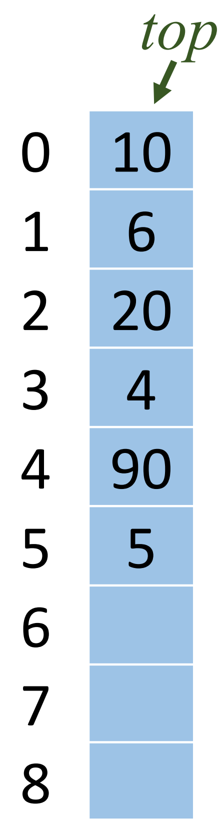
Operations
push (*element*);
element pop();
element top();

```
// Array-based stack implementation
int maxSize;  // Max size of stack
int top;      // Index for top
E [] listArray;
```

Issues:
Which end is the top?
Where does "top" point to?
What are the costs of the operations?

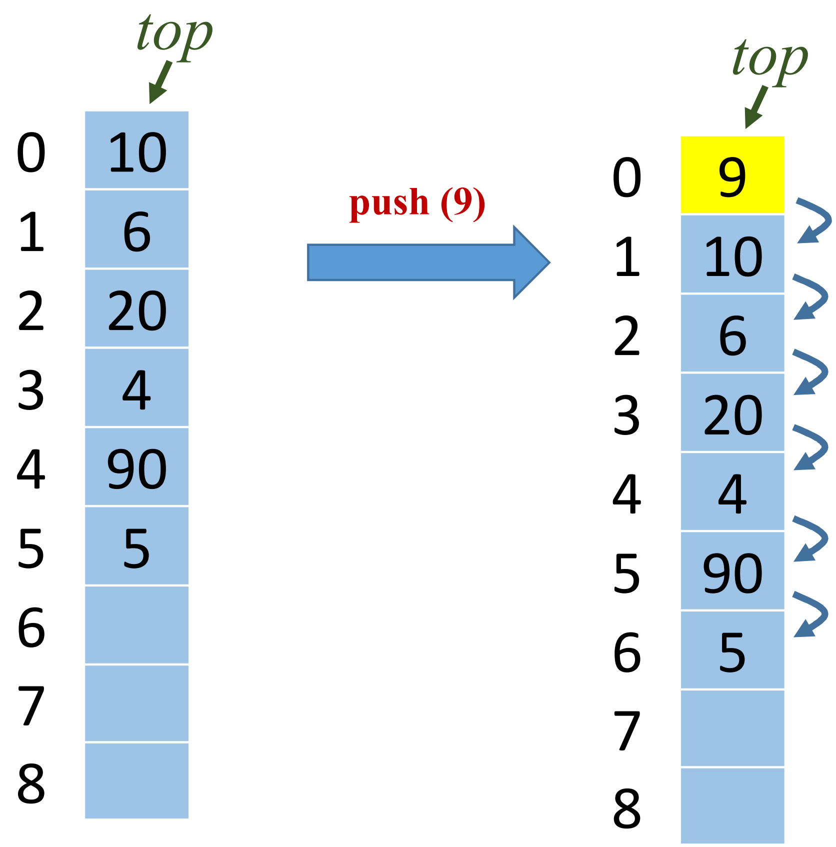
Array based Stack

Issues:
Which end is the top?
Where does “top” point to?
What are the costs of the operations?



Array based Stack

Issues:
Which end is the top?
Where does "top" point to?
What are the costs of the operations?

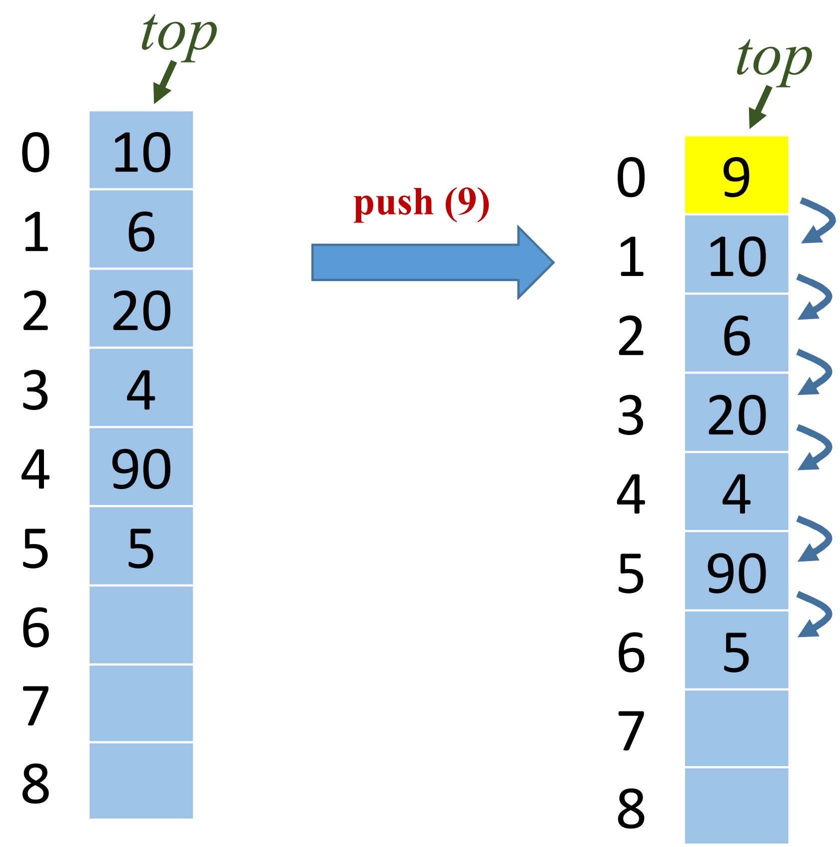


Array based Stack

Issues:
Which end is the top?
Where does "top" point to?
What are the costs of the operations?

top at index 0 : Inefficient
Requires $O(n)$ data movement
Every push is costly.

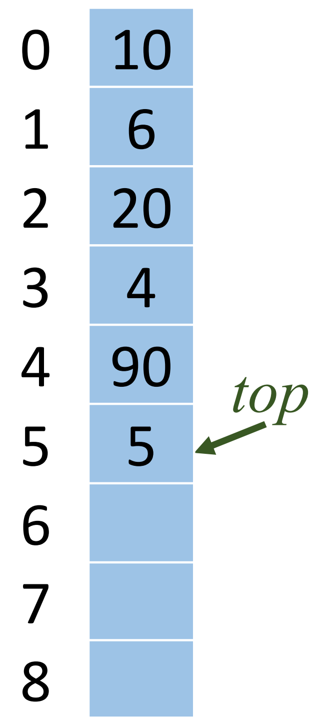
Complexity: O(n)



Array based Stack

Issues:
Which end is the top?
Where does “top” point to?
What are the costs of the operations?

top at tail?



Array based Stack

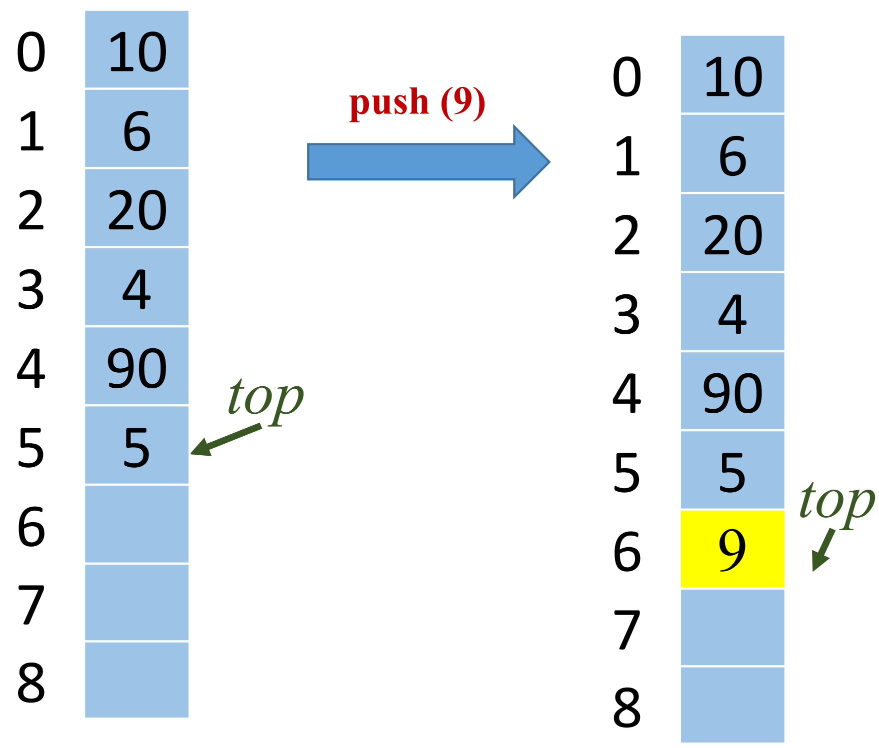
Issues:

Which end is the top?

Where does “top” point to?

What are the costs of the operations?

top at tail?

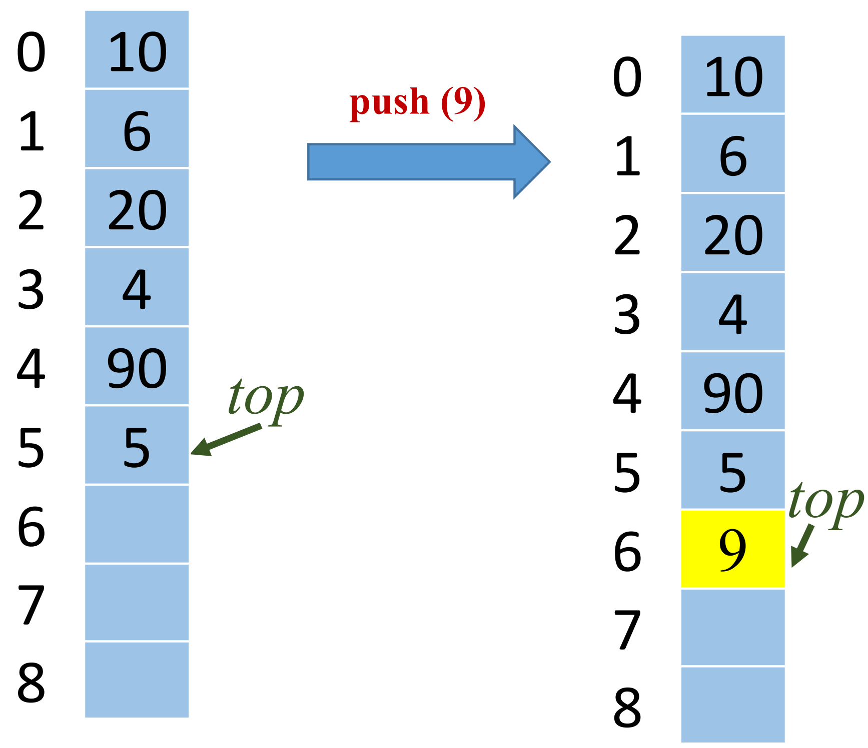


Array based Stack

Issues:
Which end is the top?
Where does "top" point to?
What are the costs of the operations?

top at tail : Efficient
Requires **NO** data movement

Similar to appending at tail.
Complexity: O(1)



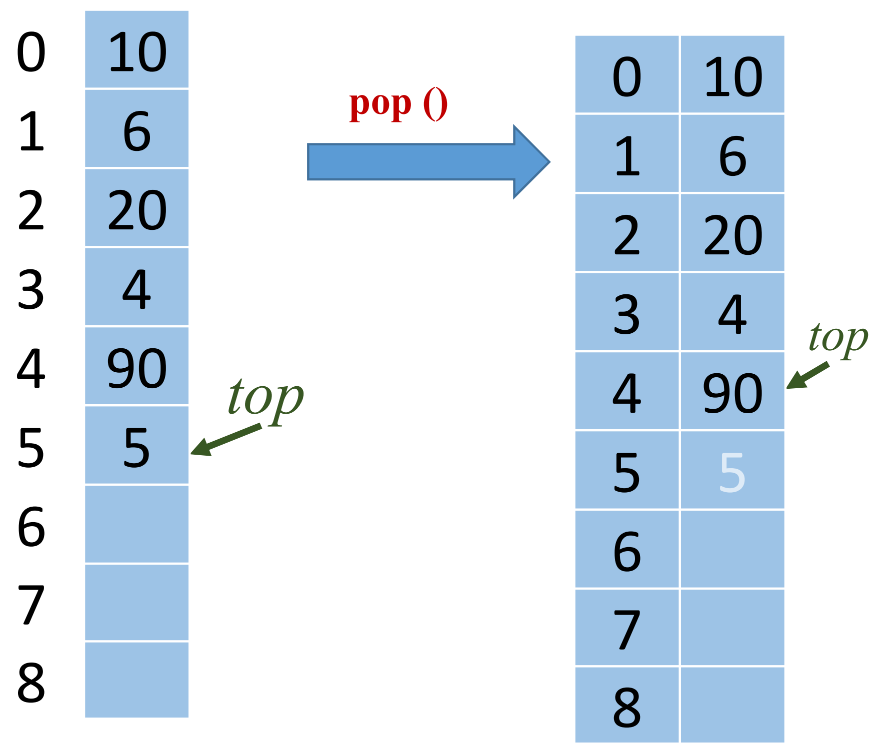
Array based Stack

Issues:

pop()

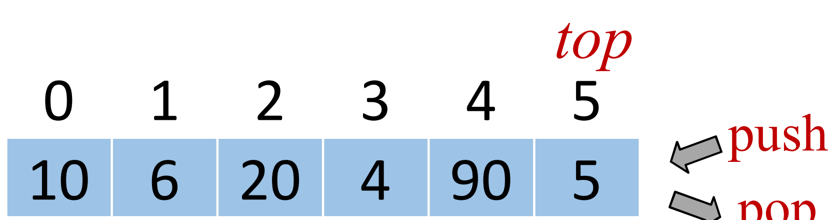
Removes element from tail

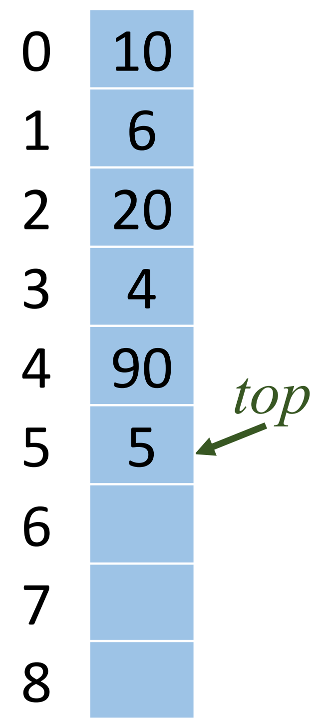
Complexity: O(1)



Common Features

- Uses an explicit linear ordering
- Insertions and removals are performed independently
- Inserted elements are *pushed onto* the stack
- The *top* of the stack is the most recently element pushed onto the stack
- When an element is *popped* from the stack, the current *top* is erased





Applications

Numerous applications:

- Parsing code:
  - Matching parenthesis
  - XML (e.g., XHTML)
- Tracking function calls
- Dealing with undo/redo operations
- …..

A simple data structure

- Given any problem, if it is possible to use a stack, this significantly simplifies the solution

Array based Implementation of Stack

0	1	2	3	4	5	*top*
10	6	20	4	90	5	

int maxSize; // Max size of stack
int top; // Index for top
int *array;

initialize ()
top = -1;
array = //make necessary allocation of size maxSize;

push(int data)
if (isFull()) //error
else array[++top] =data;

isEmpty()
return top == -1;

isFull()
return top == maxSize -1;

pop()
if (isempty()) //error
else return array[top--];

Stack with resizing array

How to grow array when capacity reached?
How to shrink array (else it stays big even when stack is small)?

First try:

- **push()**: increase size of **array[]** by 1
- **pop()** : decrease size of **array[]** by 1

Stack with resizing array

How to grow array when capacity reached?
How to shrink array (else it stays big even when stack is small)?

First try:

- **push()**: increase size of **array[]** by 1
- **pop()** : decrease size of **array[]** by 1

Too expensive

- Need to copy all of the elements to a new array.

Stack with resizing array

at n = 1, to push a new element

1) create a new array of size 2 and
2) copy all the old array elements to the new array
3) at the end add the new element.

Total 1x copy

Stack with resizing array

at $n = 1$, to push a new element

1) create a new array of size 2 and
2) copy all the old array elements to the new array
3) at the end add the new element.

Total 1x copy

at $n = 2$, to push a new element

1) create a new array of size 3 and
2) copy all the old array elements to the new array
3) at the end add the new element.

Total 2x copies

Stack with resizing array

at $n = n$-1, to push a new element

Requires Total $(n - 1)$x copeis

Total ops= $1+2+3+\ldots.+n = O(n^2)$ for n push

On average each push requires = $O(n^2)/n$ or $O(n)$ operations

Stack with resizing array

Alternate solutions: Repeated Doubling when array is full

Resize twice only when array is full

Stack with resizing array

Alternate solutions: Repeated Doubling when array is full

Resize twice only when array is full

Let we need $n = 32$ push

We resize only at $n = 1, 2,4,8, 16$

Stack with resizing array

Let we need $n = 32$ push
We resize only at $n = 1, 2,4,8, 16$
at $n = 1$, to push a new element
As before Total 1x copy

at $n = 2$, to push a new element
1) create a new array of size 4 and
2) copy all the old array elements to the new array
3) at the end add the new element.

Total 2x copies

Stack with resizing array

Let we need $n = 32$ push
We resize only at $n = 1, 2,4,8, 16$
at $n = 1$, to push a new element
As before Total 1x copy

at $n = 2$, to push a new element
1) create a new array of size 4 and
2) copy all the old array elements to the new array
3) at the end add the new element.

Total 2x copies
And so on….
Total ops= 1+2+4+8+. . .+16 = 31

Stack with resizing array

For n push, we resize $\log(n)$ times

and

Total ops $= 1+2+4+\ldots.+ n/4 + n/2 + n$

$$= n + n/2 + n/4 + \ldots + 4 + 2 + 1$$

$$= n\,(1 + ½ + 1/4 + \ldots + 4/n + 2/n + 1/n)$$

$$\approx n \times 2$$

$$= O(n)$$

Stack with resizing array

For n push, we resize $\log(n)$ times

and

Total ops $= 1+2+4+\ldots.+ n/4 + n/2 + n$

$= n + n/2 + n/4 + \ldots +4+2+1$

$= n\,(1+ \frac{1}{2} +1/4+ \ldots + 4/n + 2/n + 1/n)$

$\approx n \times 2$

$= O(n)$

For each push op, amortized time is $O(1)$

Stack with resizing array

push(int data)

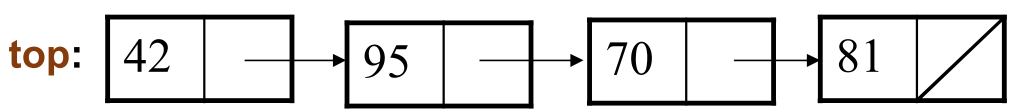
```
if (isFull() )
  doubleStackArray();

array[++top] =data;
```

```
 doubleStackArray()
1. int *temp = //allocate memory for 2*maxSize elements
2. Copy all elements from array to temp;
3. free (array);
4. array = tamp;
5. maxSize=2*maxSize;
```

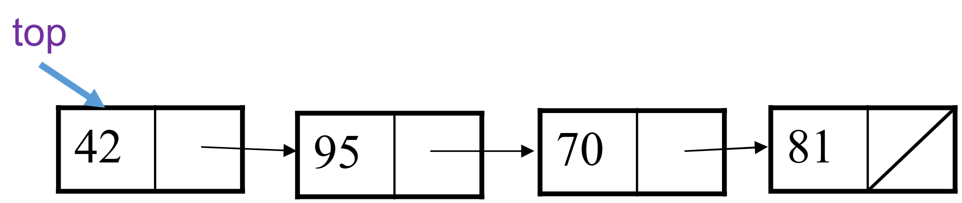
Linked List Based Stack

1. Elements are inserted and removed only from the head of the list.
2. A header node is not used because no special-case code is required for lists of zero or one elements.
3. The only data member is **top,** a pointer to the first (top) link node of the stack.
4. **top is Null** for empty stack.



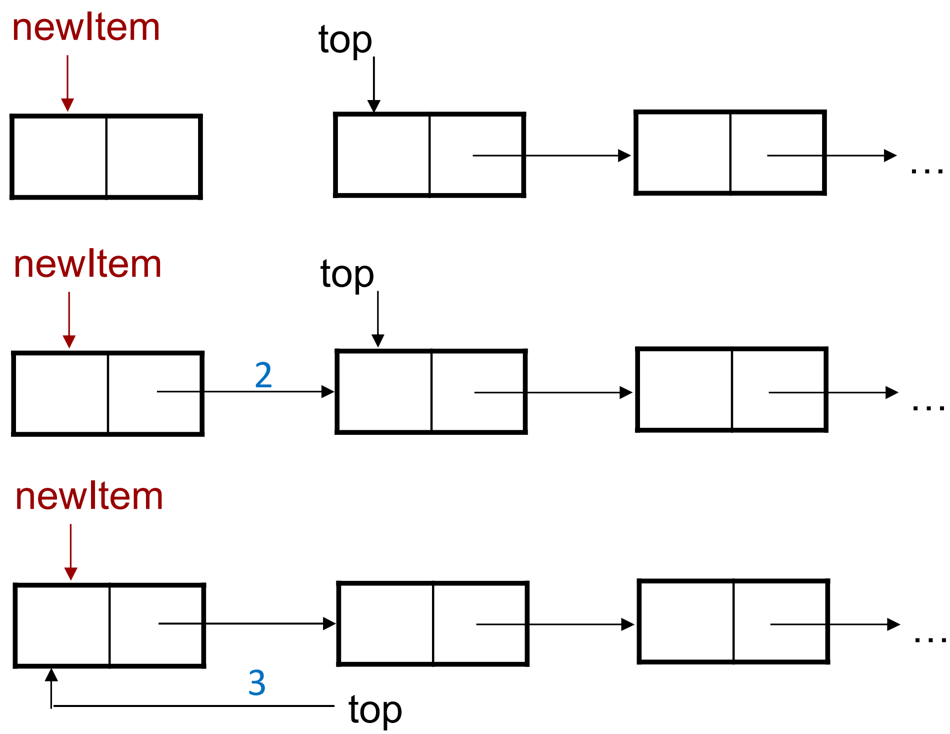
Linked List Based Stack

```
struct node {
    int element;
    struct node *next_node;
}
 struct node  *top;
```



Linked List Based Stack: push

- Step 1. Create a new node that is pointed by pointer *newItem*.
- Step 2. Link the new node to the first node of the linked list.
- Step 3. Set the pointer *top* to the new node.



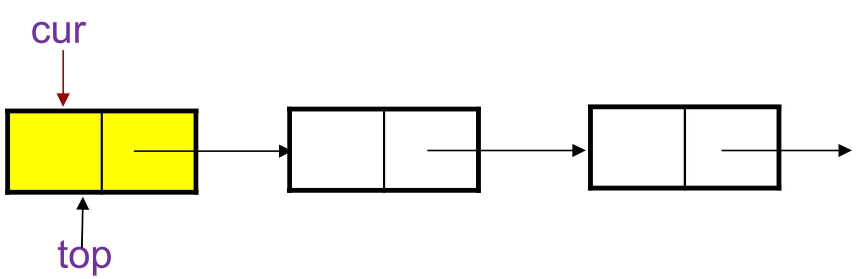

```
newItem = //create it as before
if (newItem ==NULL)  //error handling
newItem ->data= //assign it ;
newItem ->next=top;
top= newItem;
```

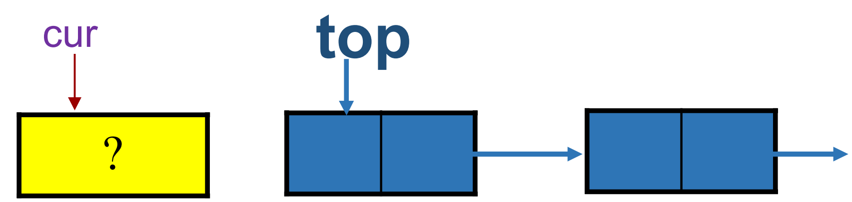
Complexity?

Linked List Based Stack: pop

- Step1. Initialize the pointer *cur* point to the top
- Step2. Move the pointer *top* to the second node of the list.
- Step3. Release the memory of the node that is pointed by the pointer *cur*.

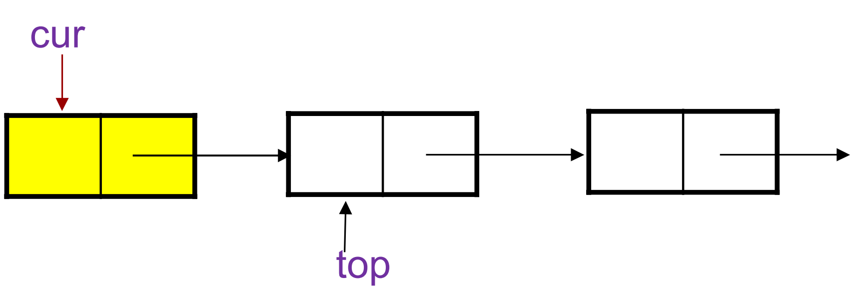
Step 1





Step 3

Step 2



Complexity?

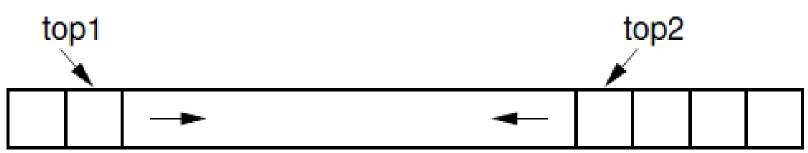
```
struct node* curr;
if (top ==NULL)  //error handling
curr =top;
data=curr->data;
top=top->next;
free (curr);
return data
```

Stack:Complexity

Operation	Array	Dynamic Array	Linked List
Space Complexity (for n push operations)	O(n)	O(n)	O(n)
Time Complexity of CreateStack()	O(1)	O(1)	O(1)
Time Complexity of Push()	O(1)	O(1)	O(1)
Time Complexity of Pop()	O(1)	O(1)	O(1)
Time Complexity of Top()	O(1)	O(1)	O(1)
Time Complexity of IsEmptyStack()	O(1)	O(1)	O(1)
Time Complexity of DeleteStack()	O(1)	O(1)	O(n)

One Array Two Stacks…

- Array based stack has a one-way growth. So, we can implement two stacks in one array
  - Each grows inward from each end
- This may lead to less wasted space.



One Array Two Stacks…

- only works well
  - when the space requirements of the two stacks are inversely correlated: ideally when one stack grows, the other will shrink.
- Very effective when elements are taken from one stack and given to the other.
- If both stacks grow at the same time, then the free space in the middle of the array will be exhausted quickly.