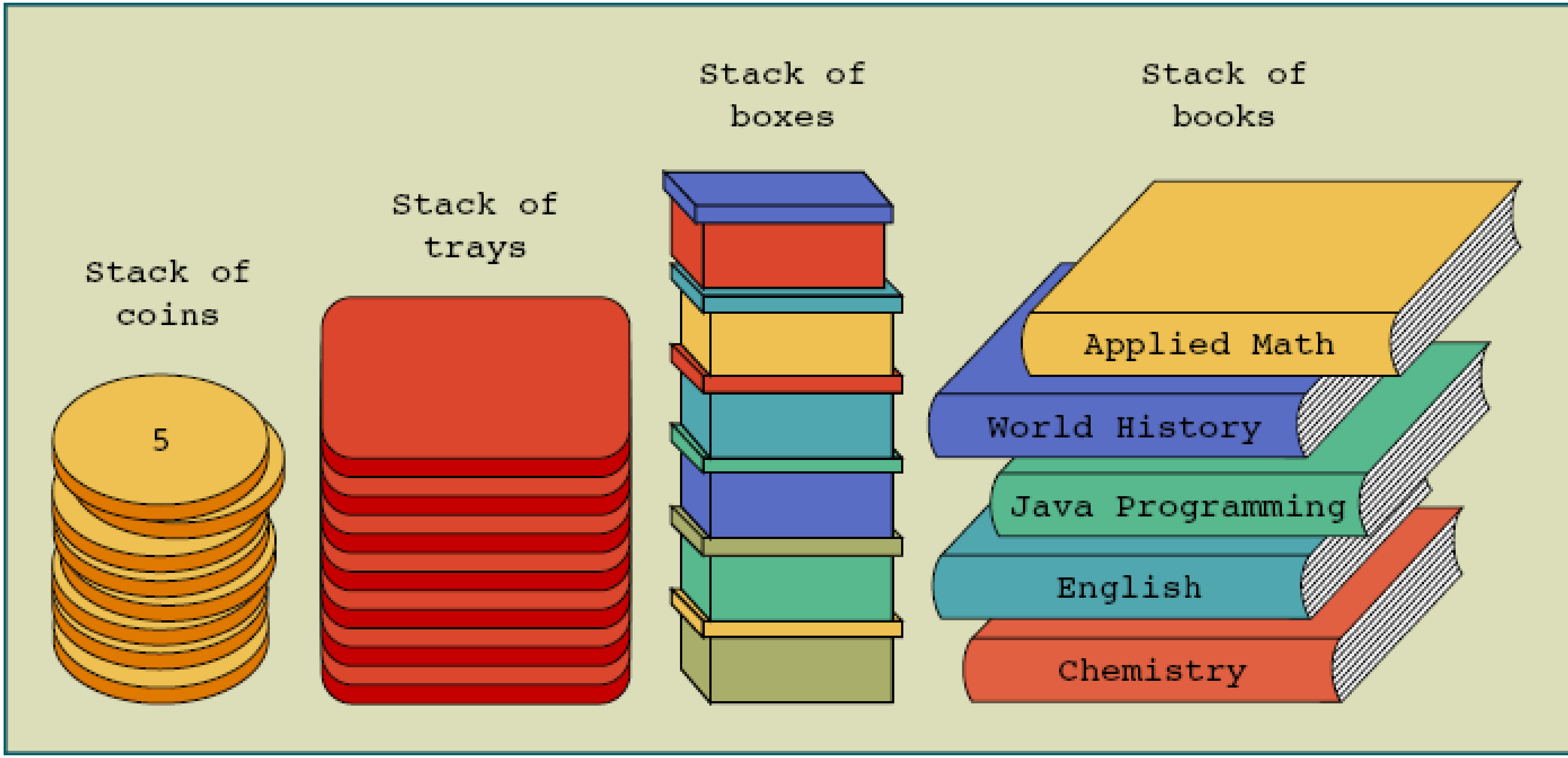
CSE 247
Data Structures

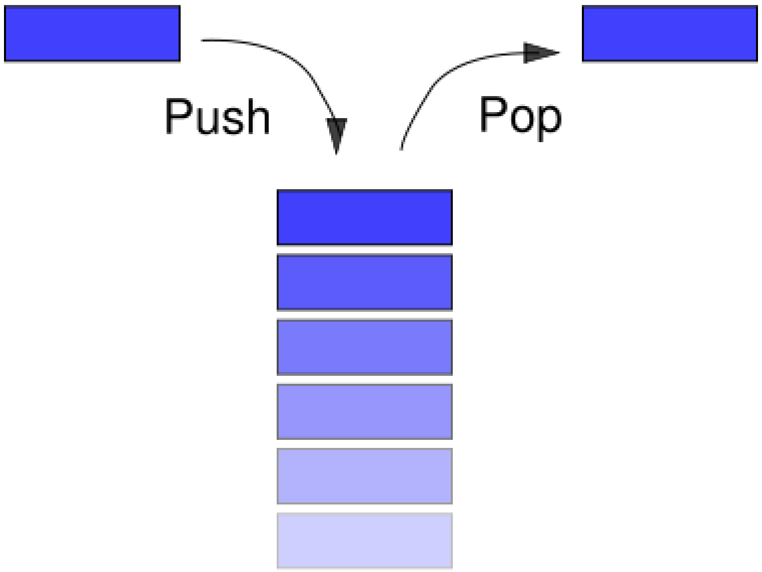
Stack

Conceptual Stack



Stack

- Lists of homogeneous elements in which
  - Addition and deletion of elements occur only at one end, called the top of the stack
  - The middle elements of the stack are inaccessible

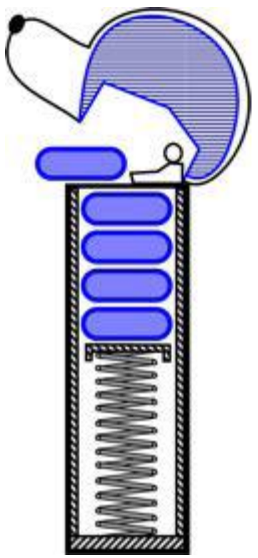


Two New ADTs

- Define two new abstract data types
- Both are restricted lists
- Can be implemented using arrays or linked lists
- Stacks is "Last In First Out" (LIFO)
- Queues is "First In First Out" (FIFO)

Stack

- Last-in, First-out (LIFO) structure
- Given a stack $S = (a_0, a_1, \ldots a_{n-1}, a_n)$, we say that a_0 is the bottom element, a_n is the top element if they are added in the order of $a_0, a_1, \ldots$ and a_n

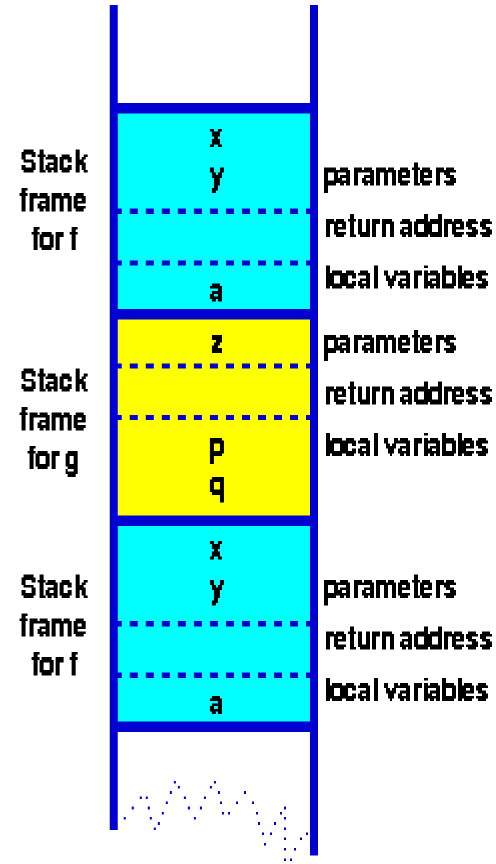


Applications

- Function/method calls implemented using stack
- Stacks are also used to convert recursive algorithms into non-recursive algorithms
- Arithmetic Expression handling (validation, postfix)
- Reversal of data
- Undo in editors and browser
- Other real life examples
  - in games,
  - back tracking,
  - palindrome (civic, level, refer, madam, radar, noon) etc.

Function calls implement using stack

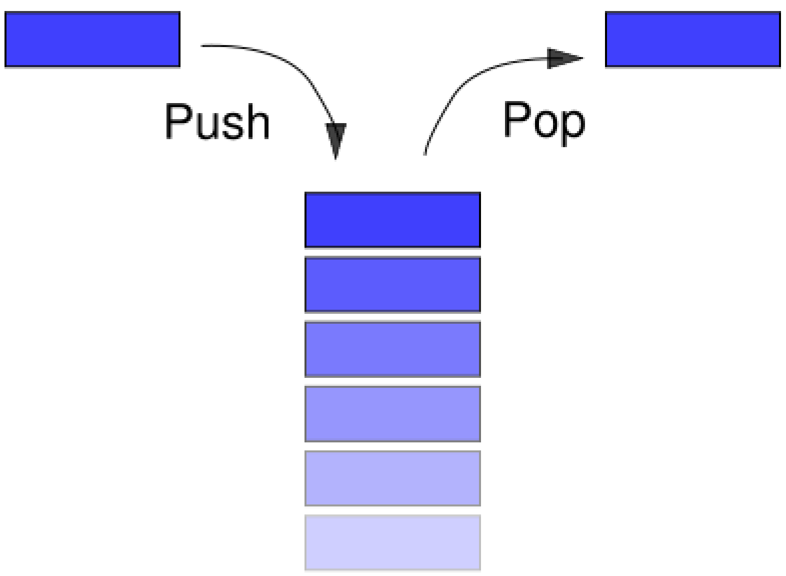
- Almost invariably, programs compiled from modern high level languages make use of a stack frame for the working memory of each procedure or function invocation.
- When any procedure or function is called, the stack frame is pushed onto a program stack.



Stack Operations

- Elements are added to and removed from one designated end called the top.
- Basic Operations
  - Push(), add element into the stack
  - pop(), remove & return topmost element

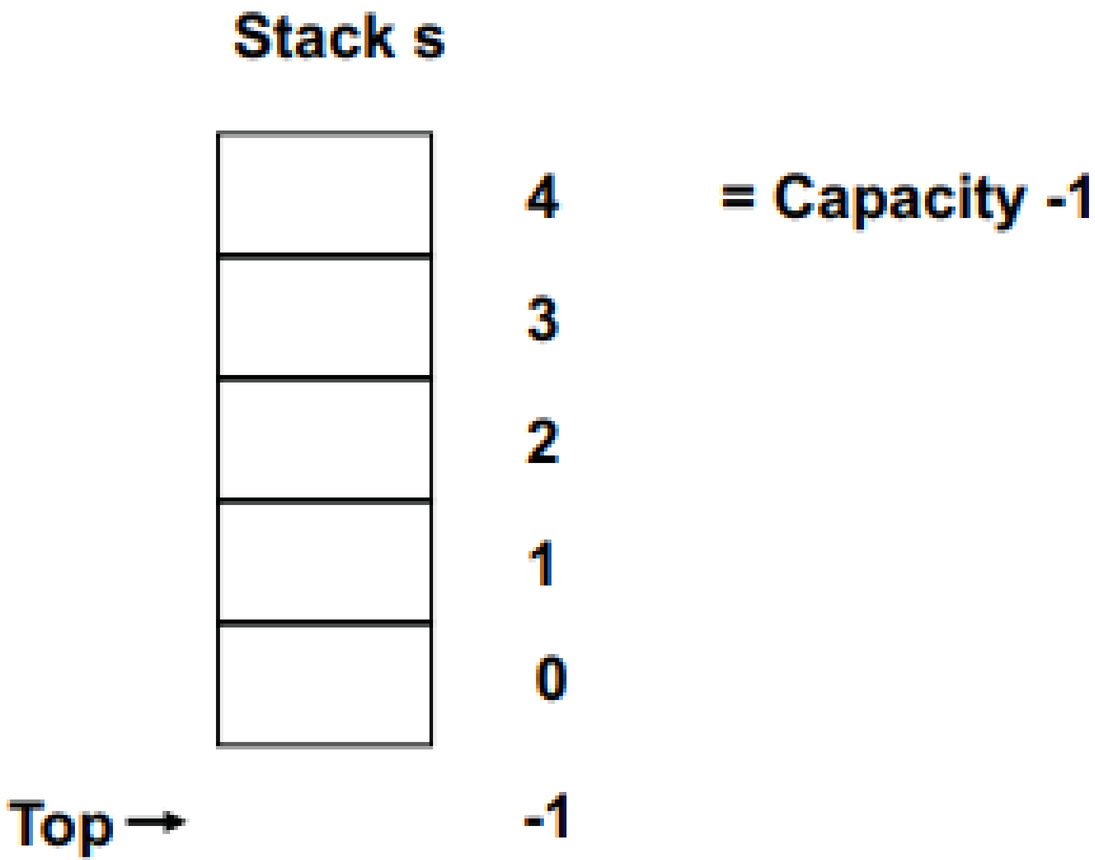
- Other Operation
  - isEmpty(): underflow
  - isFull() : overflow
  - Peek(): top element of stack
  - Size()
  - Display()



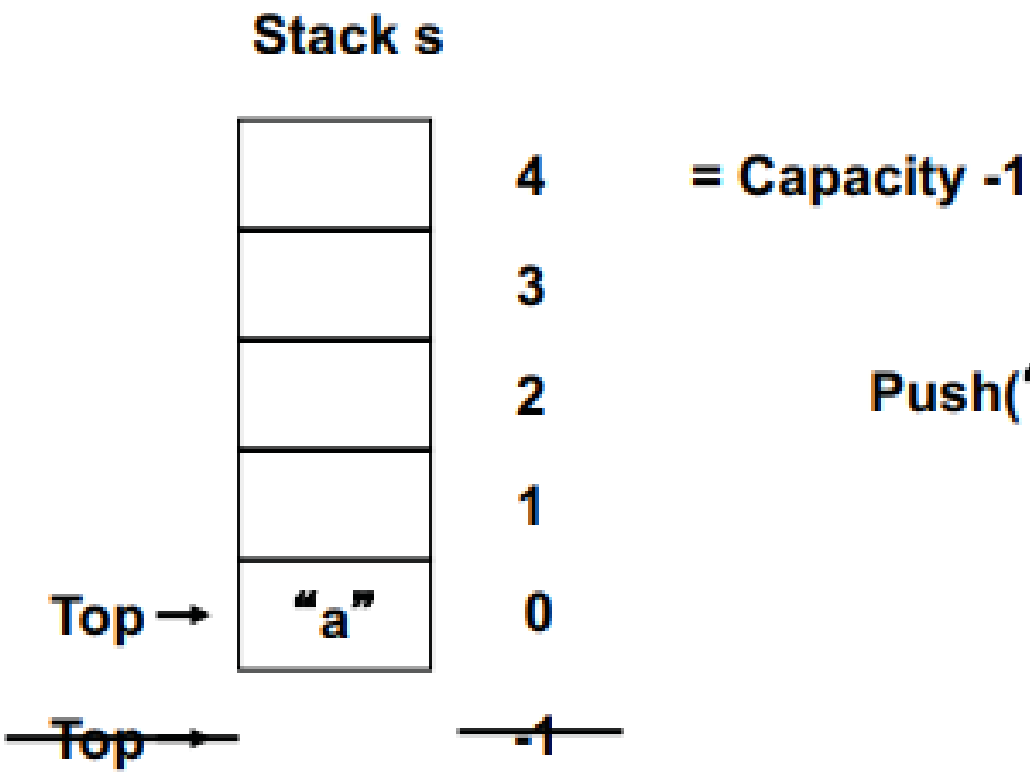
Implementation of stack

- Two possible implementations of stack
  1. Array based stack
  2. Linked list based stack

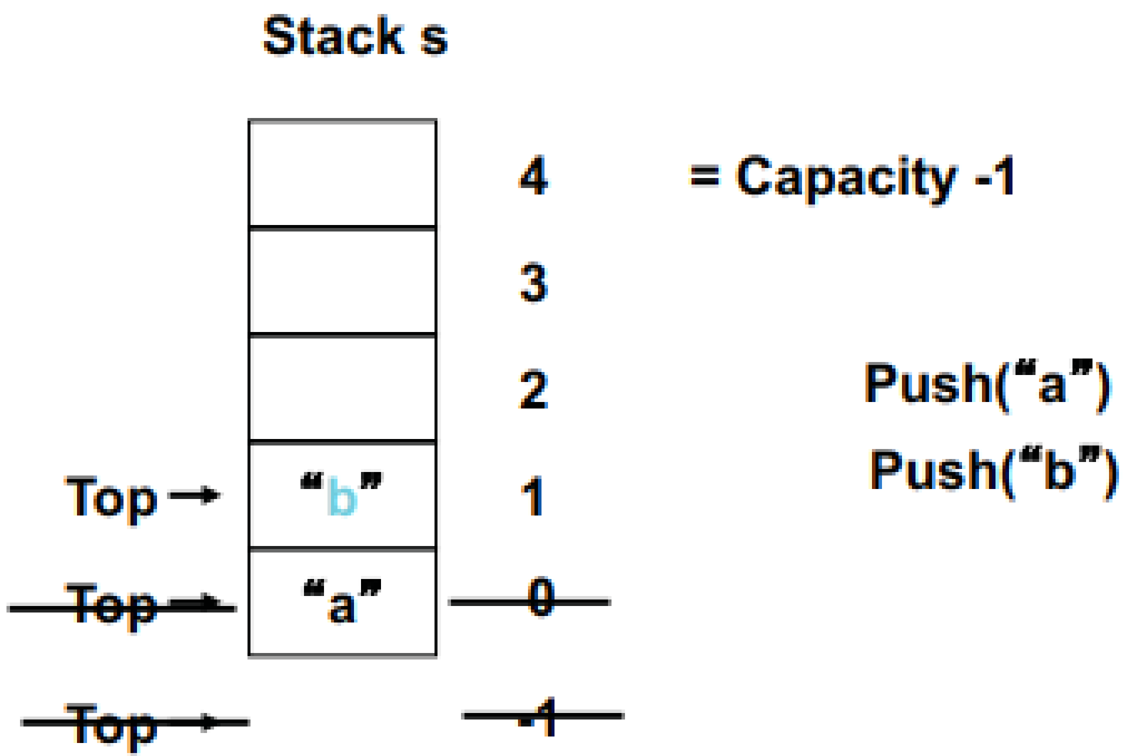
Stacks



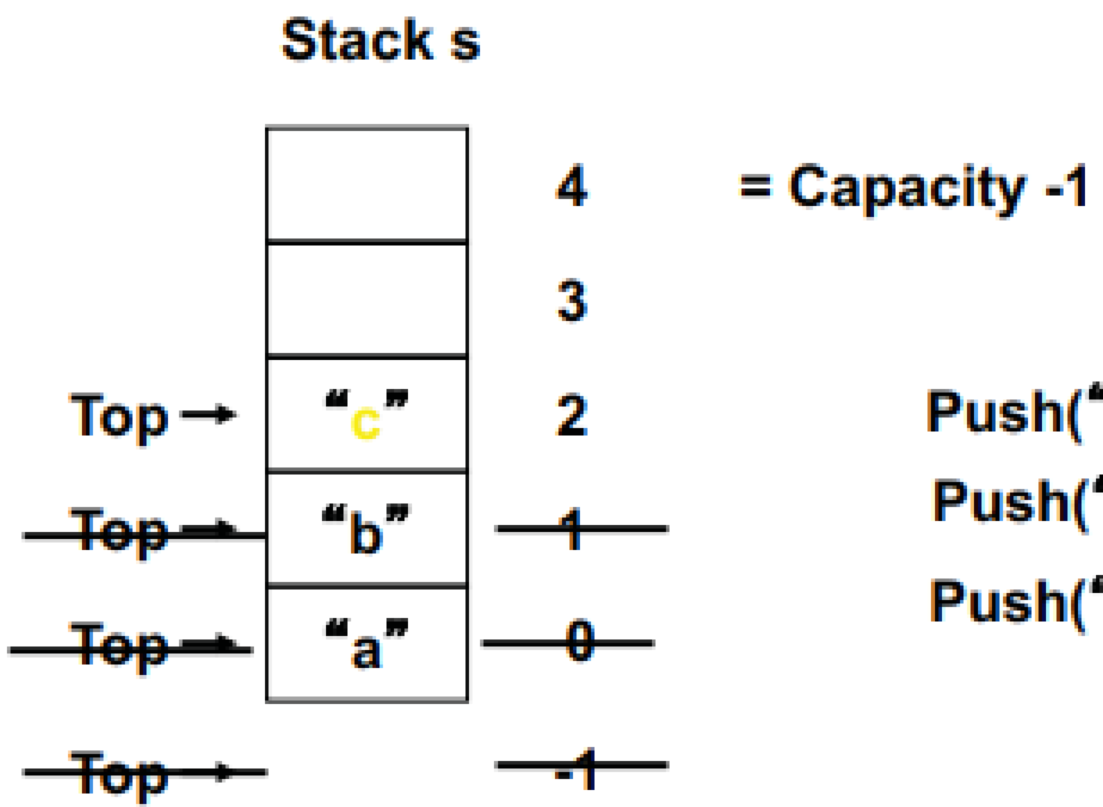
Stack



Push("a")

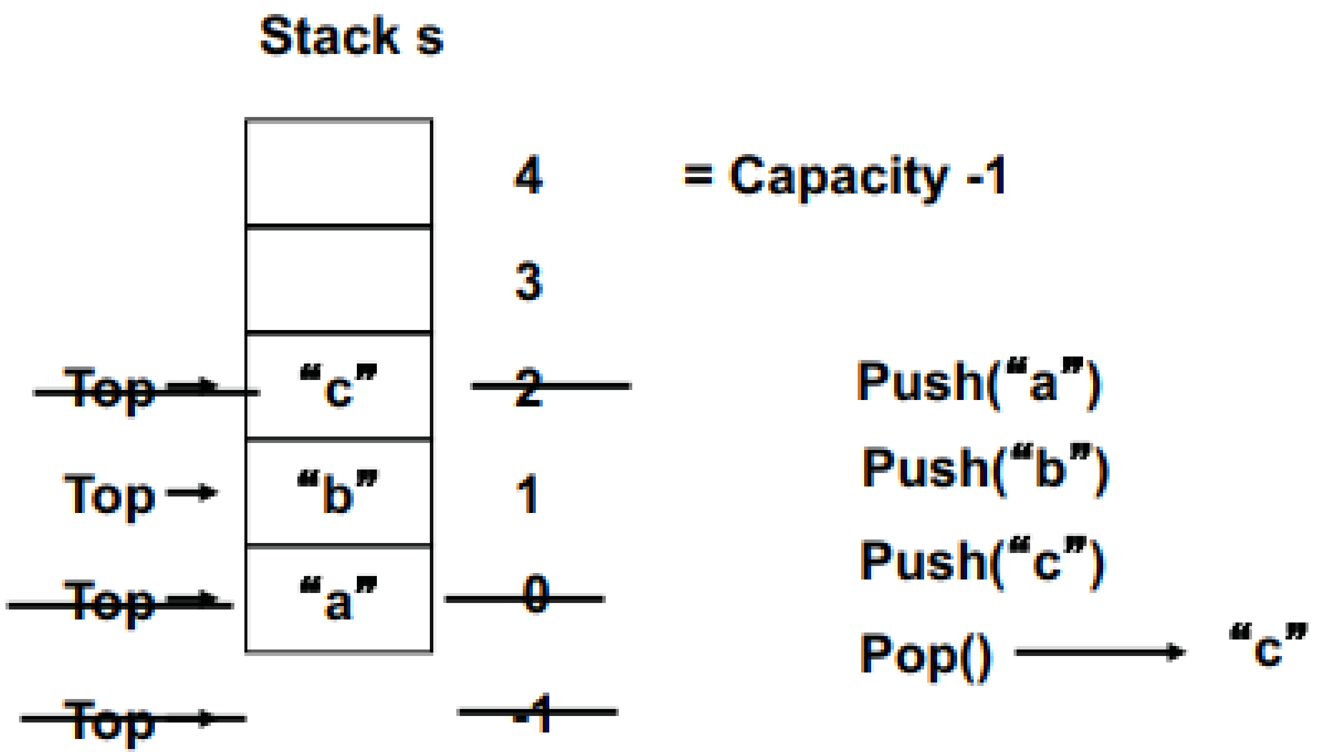

Stack s
4
= Capacity -1
3
2
Push("a")
Push("b")
Top
"b"
1
~~Top~~
"a"
~~0~~
~~Top~~
~~-1~~

Stack

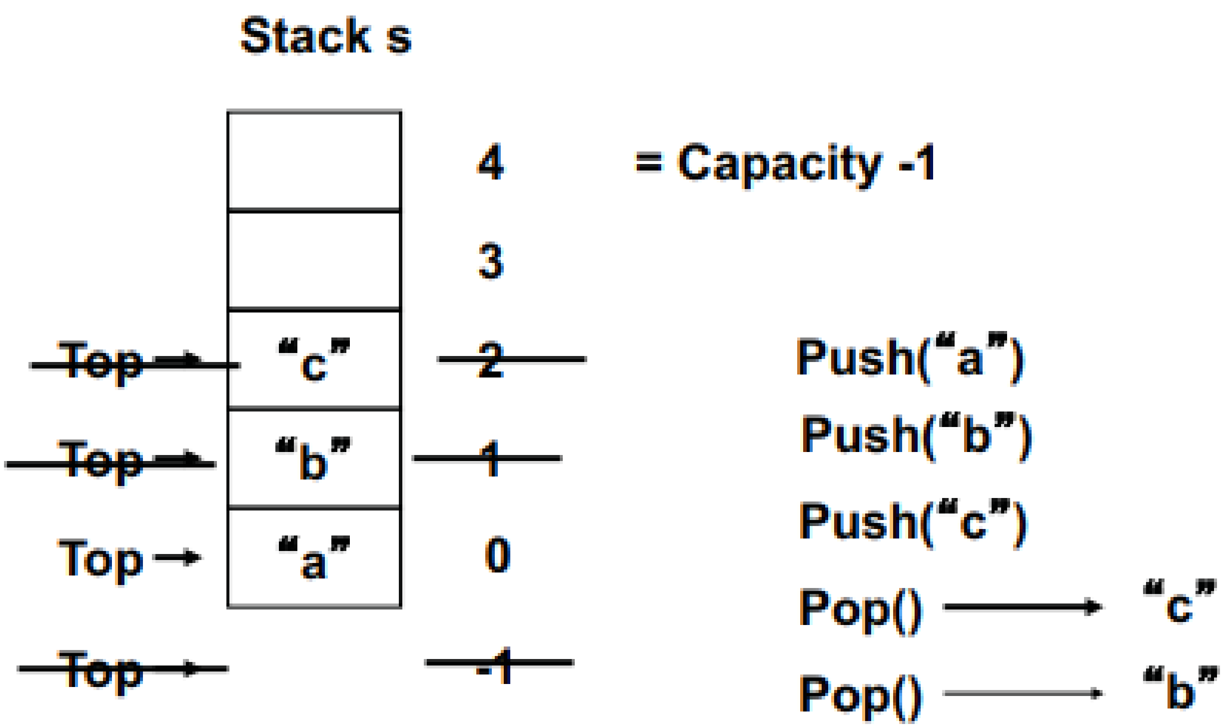


Push("a")
Push("b")
Push("c")

Stack


Stack s
4
= Capacity -1
3
~~Top~~
"c"
~~2~~
Top
"b"
1
~~Top~~
"a"
~~0~~
~~Top~~
~~-1~~
Push("a")
Push("b")
Push("c")
Pop()
"c"

Stack



Stack s
4
= Capacity -1
3
~~Top~~
"c"
~~2~~
~~Top~~
"b"
~~1~~
Top
"a"
0
~~Top~~
~~-1~~
Push("a")
Push("b")
Push("c")
Pop()
"c"
Pop()
"b"

Array based implementation of stack

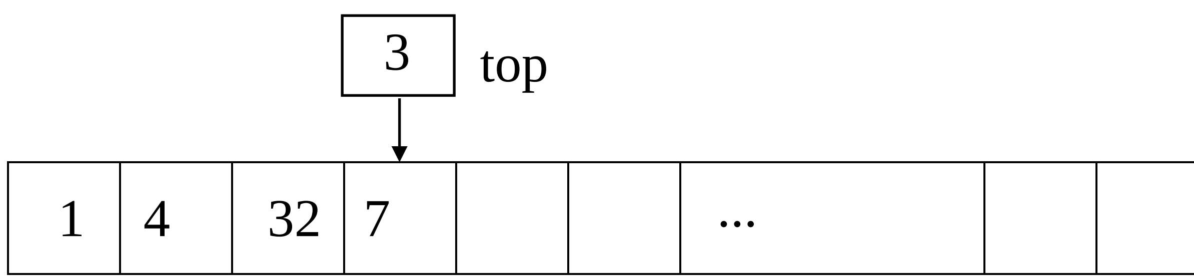
- The array implementing a stack is an array of reference variables
- Each element of the stack can be assigned to an array slot
- The top of the stack is the index of the last element added to the stack
- To keep track of the top position, declare a variable called `stackTop`.

Linked list based implementation of stack

- Arrays have fixed sizes
  - Only a fixed number of elements can be pushed onto the stack
- Dynamically allocate memory using reference variables
  - Implement a stack dynamically
- Similar to the array representation, `stackTop` is used to locate the top element
  - `stackTop` is now a reference variable

Array Implementation of a Stack

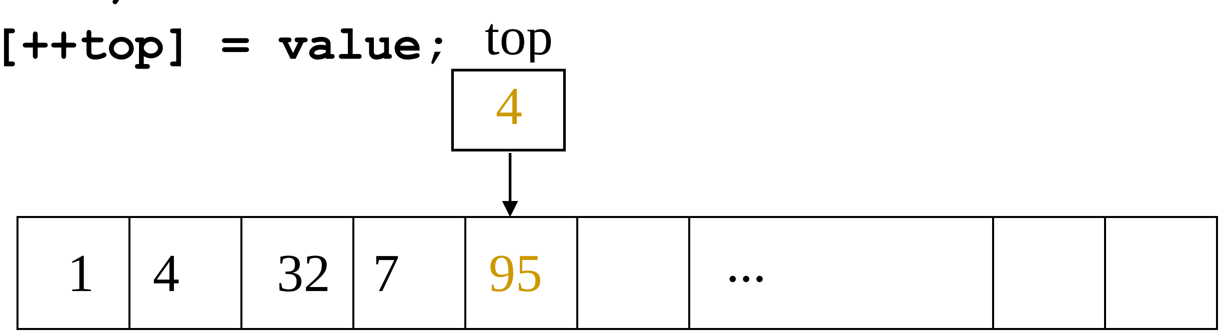
```
public class MyStack
{
    int stackArray[];
    int top;
    int MAX = 100;

    public MyStack()
    {
        stackArray = new int[MAX];
        top = -1;
    }
    // ...
}
```



ArrayStack class, continued

```
public class MyStack
{  // ...
 public boolean empty()
 {
      return (top == -1);
 }
 public void push(int value)
 {
   if (top < MAX-1)
    stackArray[++top] = value;
 }
   // ...
}
```

```
// in the code of main
// function that uses
// the stack …
MyStack S1=new MyStack();
…
S1.push( 95 );
```

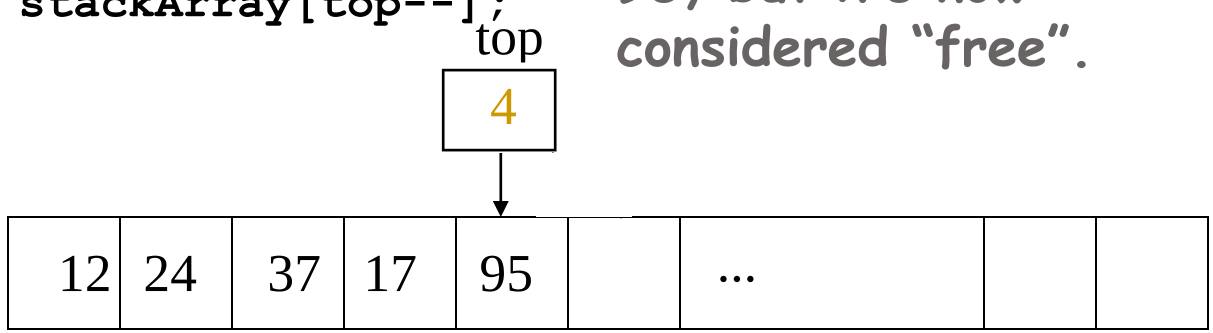


ArrayStack class, continued

```
public class MyStack
{
   // ...
   public int pop()
      throws Exception
   {
      if ( empty() )
         throw new Exception();
      else
         return stackArray[top--];
   }
}
```

```
// in the code that uses
// the stack …

int x = S1.pop();
// x gets 95,
// slot 4 is now free
```

Note:
Store[4] still contains 95, but it's now considered "free".



Array based Implementation of stack

- MAX (size of array) needs to be specified
- Consequences
  - stack may fill up (when top == MAX)
  - memory is wasted if actual stack consumption is below maximum
- The array implementation of a stack is simple and efficient for known size of list.
- Time complexity of all stack operations is O(1).

Linked List based implementation of stack

- Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**.
- Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.
- in a linked list, new elements can be stored anywhere in the memory.
- Address of the memory location allocated to the new element is stored in the previous node of linked list, hence forming a link between the two nodes/elements.

Stack implementation

- Pushing onto the array based stack can be implemented by appending a new element to the array, which takes O(1), Popping from the stack can be implemented by just removing the top element, which also runs in worst-case O(1).

- Because a singly-linked list supports O(1) time to append and delete-first, the cost to push or pop into a linked-list-based stack is also O(1) in worst-case. However, each new element added requires a new allocation, and allocations can be expensive compared to other operations.

Recognizing Strings in a Language

- Given a definition of a language, *L*
  - Special palindromes
  - Special middle character $
  - Example ABC$CBA $\in L$, but AB$AB $\notin L$
- A stack is useful in determining whether a given string is in a language
  - Traverse first half of string
  - Push each character onto stack
  - Reach $, undo, pop character, match or not

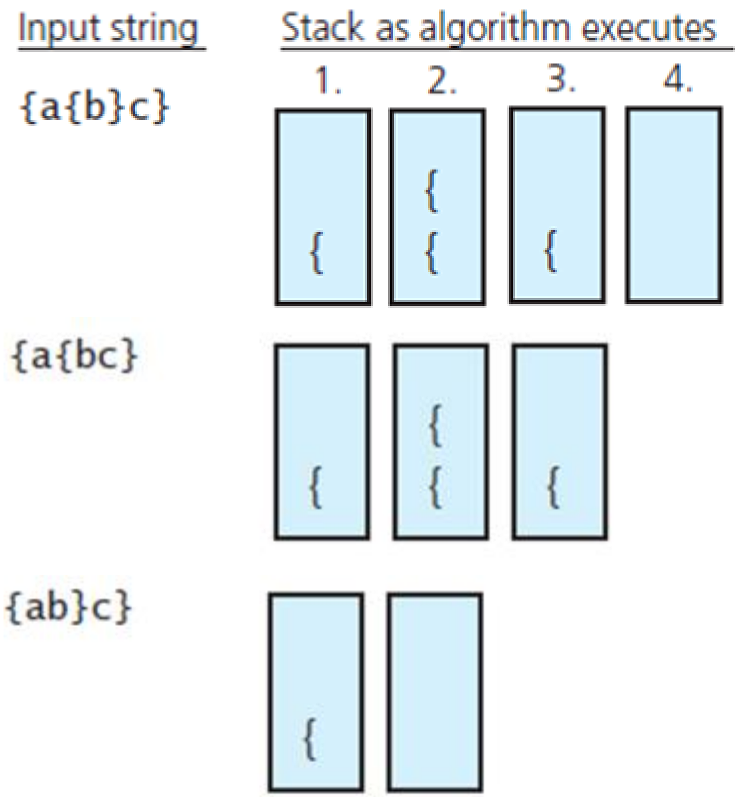
Using Stacks with Algebraic Expressions

- Strategy
  - Develop algorithm to evaluate postfix
  - Develop algorithm to transform infix to postfix

- These give us capability to evaluate infix expressions
  - This strategy easier than *directly* evaluating infix expression

Checking for Balanced Braces

- Traces of algorithm that checks for balanced braces



1. push {
2. push {
3. pop
4. pop

stack empty -> balanced

1. push {
2. push {
3. pop

Stack not empty ⟹ not balanced

1. push {
2. pop

Stack empty when "}" encountered ⟹ not balanced

Infix to Postfix

- Important facts
  - Operands always stay in same order with respect to one another.
  - Operator will move only “to the right” with respect to the operands;
    - If in the infix expression the operand *x* precedes the operator *op,*
    - Also true that in the postfix expression the operand *x* precedes the operator *op* .
  - All parentheses are removed.

Infix to Postfix

- Determining where to place operators in postfix expression
  - Parentheses
  - Operator precedence
  - Left-to-right association
- Note difficulty
  - Infix expression not always fully parenthesized
  - Precedence and left-to-right association also affect results

Infix to Postfix

ch	operatorStack (top to bottom)	postfixExp	
a		a	
–	–	a	
(	(–	a	
b	(–	a b	
+	+ (–	a b	
c	+ (–	a b c	
*	* + (–	a b c	
d	* + (–	a b c d	
)	+ (–	a b c d *	Move operators from stack to
	(–	a b c d * +	postfixExp until "("
	–	a b c d * +	
/	/ –	a b c d * +	
e	/ –	a b c d * + e	
	–	a b c d * + e /	Copy operators from
		a b c d * + e / –	stack to postfixExp

- A trace of the algorithm that converts the infix expression $a - (b + c * d) / e$ to postfix form

Steps to Convert Postfix to Infix

1. Read the symbol from the input. based on the input symbol go to steps 2 or 3.
2. If symbol is operand then push it into stack.
3. If symbol is operator then pop top values from the stack.
4. this popped value is our operand.
5. create a new string and put the operator between this operand in string.
6. push this string into stack.
7. At the end only one value remain in stack which is our infix expression.

Evaluating Postfix Expressions

- Infix expression   2 * (3 + 4)
- Equivalent postfix 2 3 4 + *
  - Operator in postfix applies to two operands immediately preceding

- Assumptions for our algorithm
  - Given string is correct postfix
  - No unary, no exponentiation operators
  - Operands are single lowercase letters, integers

Evaluating Postfix Expressions

- The effect of a postfix calculator on a stack when evaluating the expression 2 * (3 + 4)

Key entered	Calculator action		Stack (bottom to top):
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = peek	(4)	2 3 4
	pop		2 3
	operand1 = peek	(3)	2 3
	pop		2
	result = operand1 + operand2	(7)	
	push result		2 7
*	operand2 = peek	(7)	2 7
	pop		2
	operand1 = peek	(2)	2
	pop		
	result = operand1 * operand2	(14)	
	push result		14

Evaluating Postfix Expressions

- A pseudocode algorithm that evaluates postfix expressions

```
for (each character ch in the string)
{
    if (ch is an operand)
        Push the value of the operand ch onto the stack
    else // ch is an operator named op
    {
        // Evaluate and push the result
        operand2 = top of stack
        Pop the stack

        operand1 = top of stack
        Pop the stack

        result = operand1 op operand2
        Push result onto the stack
    }
}
```

Using Stack to Search a Flight Map

- Stack will contain directed path from
  - Origin city at bottom to …
  - Current visited city at top

- When to backtrack
  - No flights out of current city
  - Top of stack city already somewhere in the stack

Using Stack to Search a Flight Map

- Recall recursive search strategy.

```
To fly from the origin to the destination
{
    Select a city C adjacent to the origin
    Fly from the origin to city C
    if (C is the destination city)
        Terminate— the destination is reached
    else
        Fly from city C to the destination
}
```

Using Stack to Search a Flight Map

- Possible outcomes of exhaustive search strategy
  1. Reach destination city, decide possible to fly from origin to destination
  2. Reach a city, *C* from which no departing flights
  3. You go around in circles
- Use backtracking to recover from a wrong choice (2 or 3)