Linear Structures

Stacks

Abstract Definition

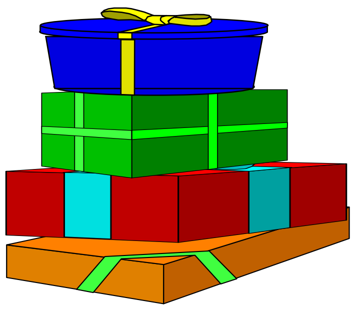
A Stack is an ordered collection of homogeneous elements (i.e. a list), in which all insertions and deletions are made at one end of the list called the top of the stack.

Specifications

- A stack is a LIFO “last in, first out” structure, which contains elements of some data type.
- The order of arrival of elements into the stack is determined by its LIFO structure.

Stacks of Boxes and Books

TOP OF THE STACK **TOP OF THE STACK**



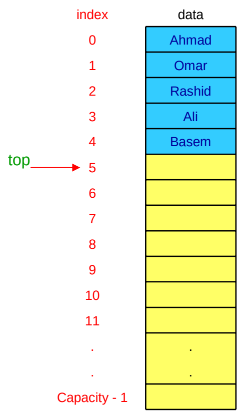
Operations

Remember that a stack is a specialized list.

Constructor()	Constructs an empty stack.
clear()	Sets the stack to an empty state.
isEmpty()	Checks if the stack is empty.
isFull()	Checks if the stack is full.
push(entry)	Adds entry at the top of the stack.
pop()	Removes and returns the top element of the stack.
peek()	Retrieves and returns the top element of the stack.

Stack Implementation Using Arrays

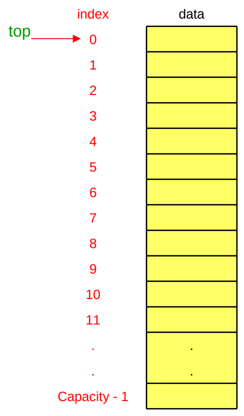
- Use a partially filled array of fixed capacity
- Use one integer variable called top, which points to the top element of the stack.
- An empty stack is initialized by setting top = 0.
- Variable top, always points to the next empty location in the array.



Mapping Operation: Constructor ... *O*(1)

- **Postcondition**:
  - The stack has been initialized as an empty stack.
- **Code**:

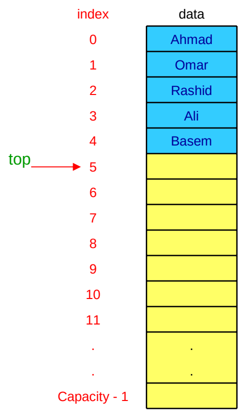
```
top = 0;
```



Mapping Operation: clear() ... *O*(1)

- **Postcondition**:
  - The stack has been emptied.
- **Code**:

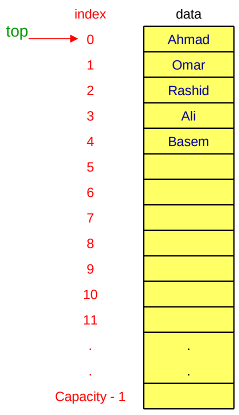
```
top = 0;
```



Mapping Operation: clear() … O(1)

- **Postcondition**:
  - The stack has been emptied.
- **Code**:

```
top = 0;
```



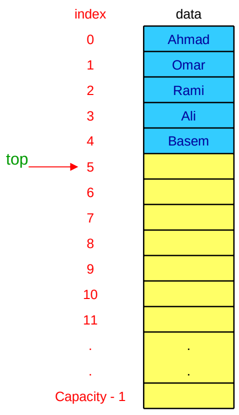
Mapping Operation: isEmpty() … O(1)

- **Postcondition**:
  - The return value is true if the stack has no items, otherwise, it is false.
- **Code**:

```
return (top == 0);
```

- **Example**:

  isEmpty(); ➔ returns false.



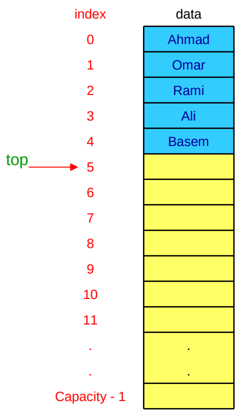
Mapping Operation: isFull() … O(1)

- **Postcondition**:
  - The return value is true if the stack has a number of items equal to its capacity, otherwise it is false.
- **Code**:

```
return (top == CAPCITY);
```

- **Example**:

  isFull(); ➔ returns false.



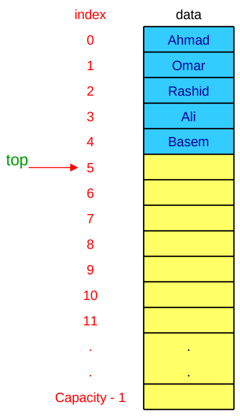
Mapping Operation: peek() … O(1)

- **Precondition:**
  - isEmpty() returns false.
- **Postcondition**:
  - A copy of the top element of the stack has been returned, and the stack remains unchanged.
- **Code**:

```
assert !isEmpty();
return data[top - 1];
```

- **Example**:

  peek(); ➔ returns 'Basem'.



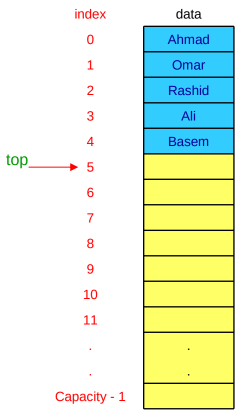
Mapping Operation: push(entry) ... *O*(1)

- **Precondition:**
  - isFull() returns false.
- **Postcondition**:
  - A new copy of entry has been added to the top of the stack.
- **Code**:

```
assert !isFull();
data[top] = entry;
top++;
```

- **Example**:

```
push('Fouad');
```



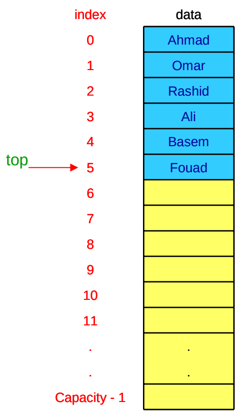
Mapping Operation: push(entry) ... *O*(1)

- **Precondition:**
  - isFull() returns false.
- **Postcondition**:
  - A new copy of entry has been added to the top of the stack.
- **Code**:

```
assert !isFull();
data[top] = entry;
top++;
```

- **Example**:

```
push('Fouad');
```



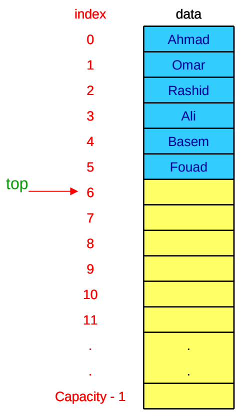
Mapping Operation: push(entry) … O(1)

- **Precondition:**
  - isFull() returns false.
- **Postcondition**:
  - A new copy of entry has been added to the top of the stack.
- **Code**:

```
assert !isFull();
data[top] = entry;
top++;
```

- **Example**:

  push('Fouad');



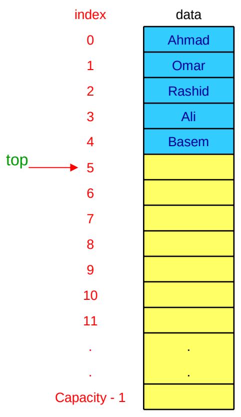
Mapping Operation: pop() … O(1)

- **Precondition:**
  - isEmpty() returns false.
- **Postcondition**:
  - The top item of the stack has been removed and returned.
- **Code**:

```
assert !isEmpty();
top--;
return data[top];
```

- **Example**:

  pop(); ➔ returns 'Basem'.



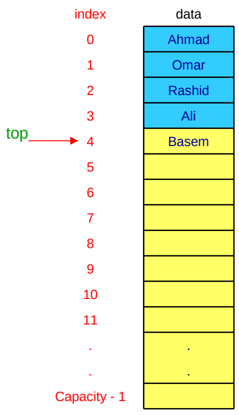
Mapping Operation: pop() … *O*(1)

- **Precondition:**
  - isEmpty() returns false.
- **Postcondition**:
  - The top item of the stack has been removed and returned.
- **Code**:

```
assert !isEmpty();
top--;
return data[top];
```

- **Example**:

  pop(); ➔ returns 'Basem'.



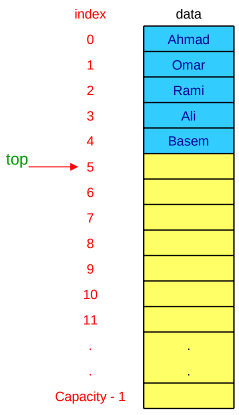
Mapping Operation: size() … *O*(1)

- **Postcondition**:
  - The return value is the number of items in the stack.
- **Code**:

```
return top;
```

- **Example**:

  size(); ➔ returns 5.



Stack Implementation in Java Using Arrays and Generic Classes

```
public class ArrayStack<E> {
   private static final int CAPACITY = 30;          // default array capacity
   private int top;                      // Index for the top of the stack
   private E data[ ];                    // Generic array used for the stack


   // DEFAULT CONSTRUCTOR
   public ArrayStack() {
      data = (E[ ]) new Object[CAPACITY];         // Array of default capacity
      top = 0;                                    // Indicates empty stack
   }
```

Stack Implementation in Java Using Arrays and Generic Classes

```
   // MUTATOR METHODS
   public void push(E entry)        { assert !isFull(); data[top] = entry; top++; }
   public E pop();                  { assert !isEmpty(); top--; return data[top];}
   public void clear()              { top = 0; }


   // OBSERVER METHODS
   public boolean isEmpty()         { return top == 0; }
   public boolean isFull()          { return top == CAPACITY; }
   public int size()                { return top; }
   public E peek()                  { assert !isEmpty(); return data[top - 1]; }
}
```

Stack Applications

- Parsing
- Function calls
- Evaluating postfix expressions.
- Converting infix to postfix.
- Bracket matching.
- Undo operations
- Mazes

Stack Applications
The Run-time Stack

Whenever a function begins execution (i.e., is activated), an *activation record* (or *stack frame*) is created to store the *current environment* for that function. Its contents include:

Value Parameters
Local Variables
Caller's Return Address

What kind of data structure should be used to store these so that they can be recovered and the system reset when the function resumes execution?

The Run-time Stack (cont.)

Problem: *FunctionA* can call *FunctionB*
FunctionB can call *FunctionC*.

When a function calls another function, it interrupts its own execution and needs to be able to resume its execution in the same state it was in when it was interrupted.

When *FunctionC* finishes, control should return to *FunctionB*.
When *FunctionB* finishes, control should return to *FunctionA*.
So, the order of returns from a function is the *reverse* of function invocations; that is, **LIFO** behavior.

∴ Use a **stack** to store the activation records.
Since it is manipulated at *run-time*, it is called the **run-time stack**.

The Run-time Stack (cont.)

What happens when a function is called?

1. **Push** a copy of its activation record **onto the run-time stack**
2. Copy its arguments into the parameter spaces
3. Transfer control to the address of the function's body

The **top activation record** in the run-time stack is ***always*** that of the function **currently executing**.

What happens when a function terminates?

1. **Pop** activation record of terminated function from the run-time stack
2. Use new top activation record to **restore the environment of the interrupted function and resume** execution of the interrupted function.

The Reverse Polish Notation (RPN)

1. What is RPN (Reverse Polish Notation)?
   A notation for arithmetic expressions in which operators are written **after** the operands. Expressions can be written without using **parentheses**.

   Developed by Polish logician, Jan Lukasiewics, in 1950's

Infix notation:	operators written **between** operands
Postfix " (**RPN**):	operators written **after** operands
Prefix " :	operators written **before** operands

RPN (cont.)

Examples:

INFIX	RPN (POSTFIX)	PREFIX
`A + B`	`A B +`	`+ A B`
`A * B + C`	`A B * C +`	`+ * A B C`
`A * (B + C)`	`A B C + *`	`* A + B C`
`A - (B - (C - D))`	`A B C D - - -`	`- A - B - C D`
`A - B - C - D`	`A B - C - D -`	`- - - A B C D`

Evaluating RPN Expressions 1

"By hand" (Underlining technique):

1. Scan the expression from left to right to find an operator.
2. Locate ("underline") the last two preceding operands and combine them using this operator.
3. Repeat until the end of the expression is reached.

Example:

$2\ 3\ 4\ +\ 5\ 6\ -\ -\ *$

$\rightarrow 2\ \underline{3\ 4\ +}\ 5\ 6\ -\ -\ *$

$\rightarrow 2\ 7\ 5\ 6\ -\ -\ *$

$\rightarrow 2\ 7\ \underline{5\ 6\ -}\ -\ *$

$\rightarrow 2\ 7\ -1\ -\ *$

$\rightarrow 2\ \underline{7\ -1\ -}\ * \rightarrow 2\ 8\ * \rightarrow \underline{2\ 8\ *} \rightarrow 16$

Stack Applications
Evaluating RPN Expressions 2

Algorithm using a Stack:

Receive: An RPN expression.
Return: A stack whose top element is the value of RPN expression (unless an error occurred).

Evaluate:

T = Second OP First

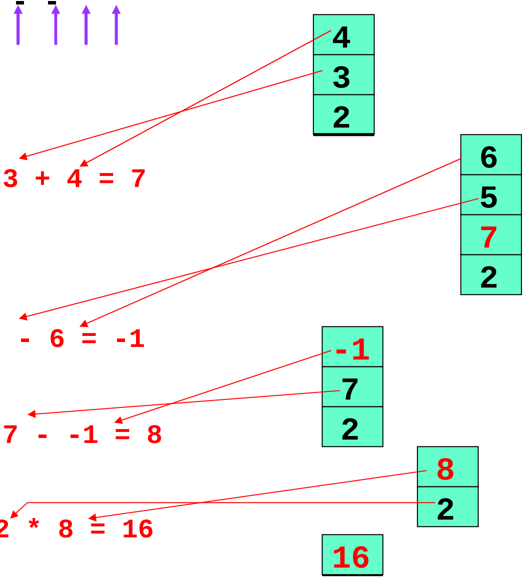
```
1. Initialize an empty stack.
2. Repeat the following until the end of the expression is encountered:
   a. Get next token (constant, variable, arithmetic operator) in the RPN expression.
   b. If the token is an operand, push it onto the stack.
      If it is an operator, then:
        (i) Pop top two values from the stack.  If the stack does not contain two items,
            signal error due to a malformed RPN and terminate evaluation.
        (ii) Apply the operator to these two values.
        (iii) Push the resulting value back onto the stack.
3. When the end of the expression is encountered, its value is on top of the stack (and, in
   fact, must be the only value in the stack).
```

Push T

Sample RPN Evaluation

Example: `2 3 4 + 5 6 - - *`

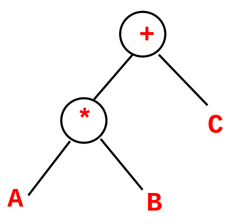
- Push `2`
- Push `3`
- Push `4`
- Read `+`
- Pop `4`, Pop `3`, `3 + 4 = 7`
- Push `7`
- Push `5`
- Push `6`
- Read `-`
- Pop `6`, Pop `5`, `5 - 6 = -1`
- Push `-1`
- Read `-`
- Pop `-1`, Pop `7`, `7 - -1 = 8`
- Push `8`
- Read `*`
- Pop `8`, Pop `2`, `2 * 8 = 16`
- Push `16`



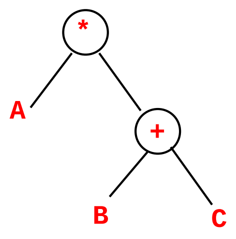
Converting Infix to RPN 1

By hand: Represent infix expression as an *expression tree*:

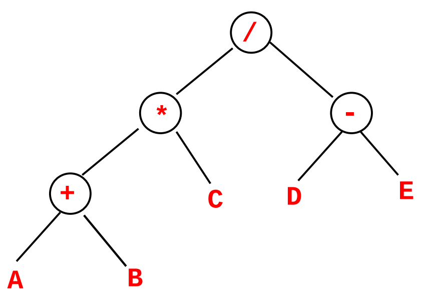
`A * B + C`

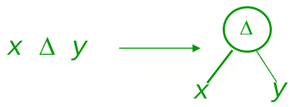


`A * (B + C)`



`((A + B) * C) / (D - E)`





Traverse the tree in *Left*-*Right*-*Parent* order (*postorder*) to get RPN**:**

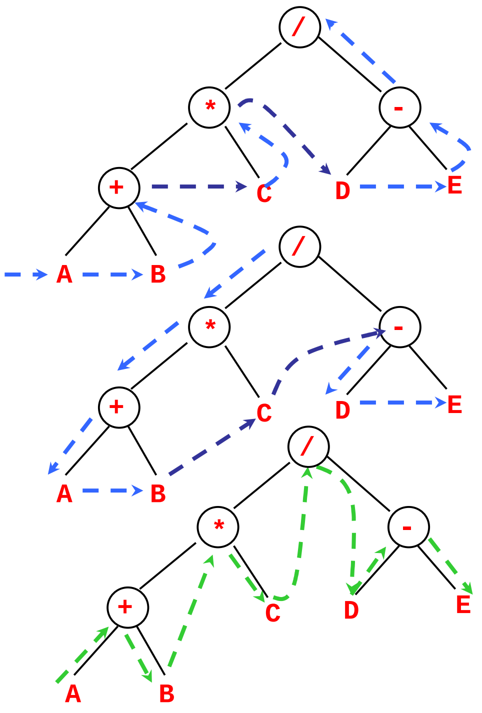
A B + C * D E - /

Traverse tree in *Parent*-*Left*-*Right* order (*preorder*) to get **prefix:**

/ * + A B C - D E

Traverse tree in *Left*-*Parent*-*Right* order (*inorder*) to get **infix:**
— must insert ()'s

(((A + B)* C) /(D - E))



Converting Infix to RPN 2

By hand: "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.
2. Replace each right parenthesis by the corresponding operator.
3. Erase all left parentheses.

Examples:

```
A * B + C   →  ((A * B) + C)        A * (B + C)  →  (A * (B + C) )
               → ((A B * C +                         → (A (B C + *
               → A B * C +                           → A B C + *
```

Exercise:

```
((A + B) * C) / (D - E)
```

Stack Applications
Converting Infix to RPN 3

Algorithm using a Stack :

1. Initialize an empty stack of operators.

2. While no error has occurred and the end of the infix expression is not reached

   a. Get the next *token* in the infix expression.

   b. If the *token* is:

(i) a *left parenthesis*:	Push it onto the stack.
(ii) a *right parenthesis*:	Pop and display stack elements until a left parenthesis is encountered, pop but do not display it. (Error if the stack is empty with no left parenthesis found.)

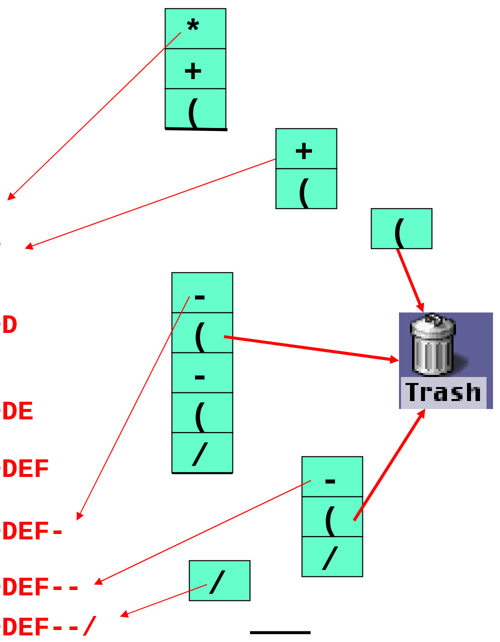
Converting Infix to RPN 3 (cont.)

(iii) an *operator:*	If the stack is empty or *token* has higher priority than top stack element, push the *token* onto the stack. (A left parenthesis in the stack has lower priority than operators) Otherwise, pop and display the top stack element; then repeat the comparison of the *token* with the new top stack item.
(iv) an *operand*:	Display it.

3. When the end of the infix expression is reached, pop and display stack items until the stack is empty.

Example: **(A+B*C)/(D-(E-F))**

	Output
Push **(**	
Display **A**	**A**
Push **+**	
Display **B**	**AB**
Push *****	
Display **C**	**ABC**
Read **)**	
Pop *****, Display *****,	**ABC***
Pop **+**, Display **+**, Pop **(**	**ABC*+**
Push **/**	
Push **(**	
Display **D**	**ABC*+D**
Push **-**	
Push **(**	
Display **E**	**ABC*+DE**
Push **-**	
Display **F**	**ABC*+DEF**
Read **)**	
Pop **-**, Display **-**, Pop **(**	**ABC*+DEF-**
Read **)**	
Pop **-**, Display **-**, Pop **(**	**ABC*+DEF--**
Pop **/**, Display **/**	**ABC*+DEF--/**



Stack Applications
Undo Operations (Backtracking)

- When several steps are needed to reach a solution, and at each step only one of several available options of action must be chosen, then at some point it may be necessary to go back to a previous step (i.e. undo all steps that were done after that step), so that other options can be explored.
- Example:

  The N-Queens Problem

Advantages and Disadvantages

- Time complexity of all stack operations is O(1).
  - A very efficient data structure.

- Fixed Storage space must be reserved in advance
  - Stack depth is limited to the array size that was reserved.
  - May overflow or may waste unused space.

- What is a Double Stack?