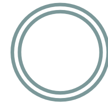


Presentation on Stacks

Stacks

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT = LIFO
- It means: the last element inserted is the first one to be removed

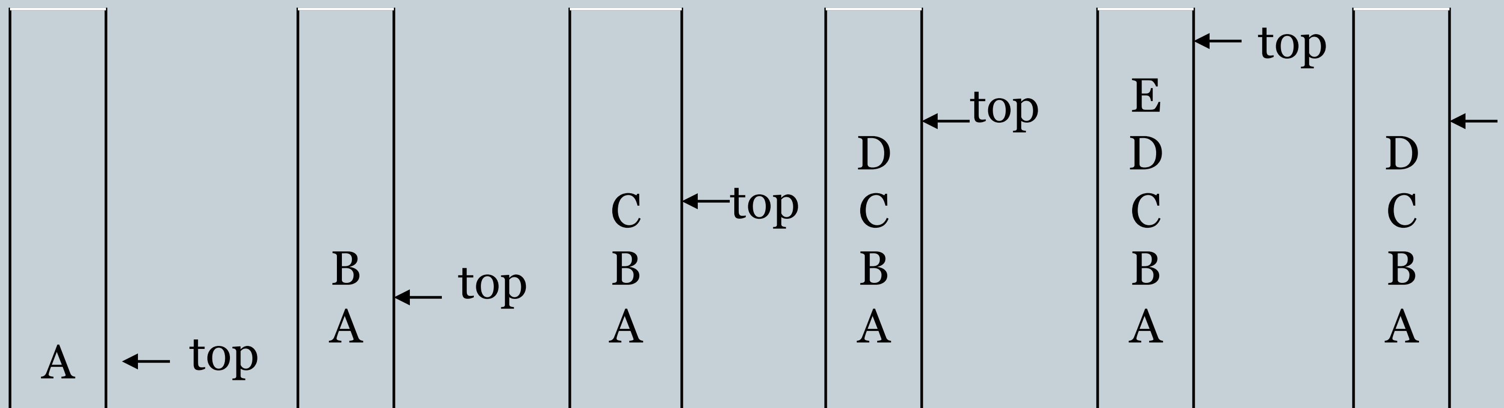
- Example :**Pile of coins**
- Elements can be accessed
- At only one end, called

'top'.

Stacks

- A stack is a sequence of items that are accessible at only

Principle of Stacks: Last In First Out



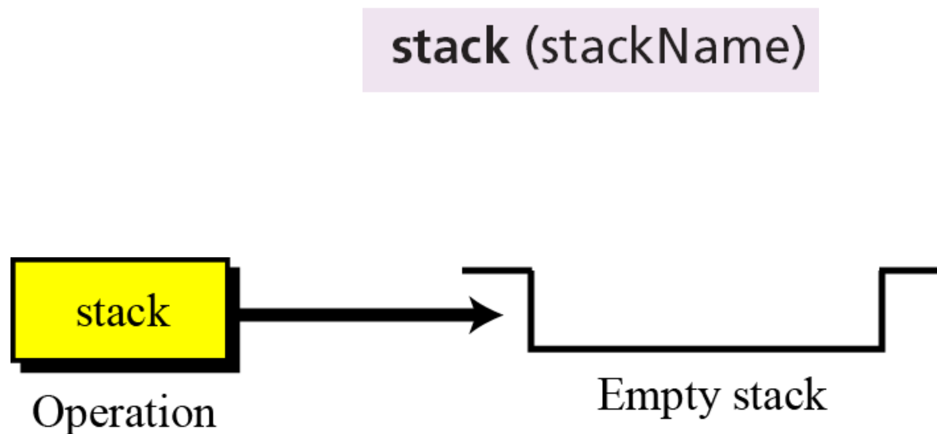
Operations on stacks

There are four basic operations, *stack*, *push*, *pop* and *empty*.

The *stack* operation

The *stack* operation creates an empty stack. The following shows the format.

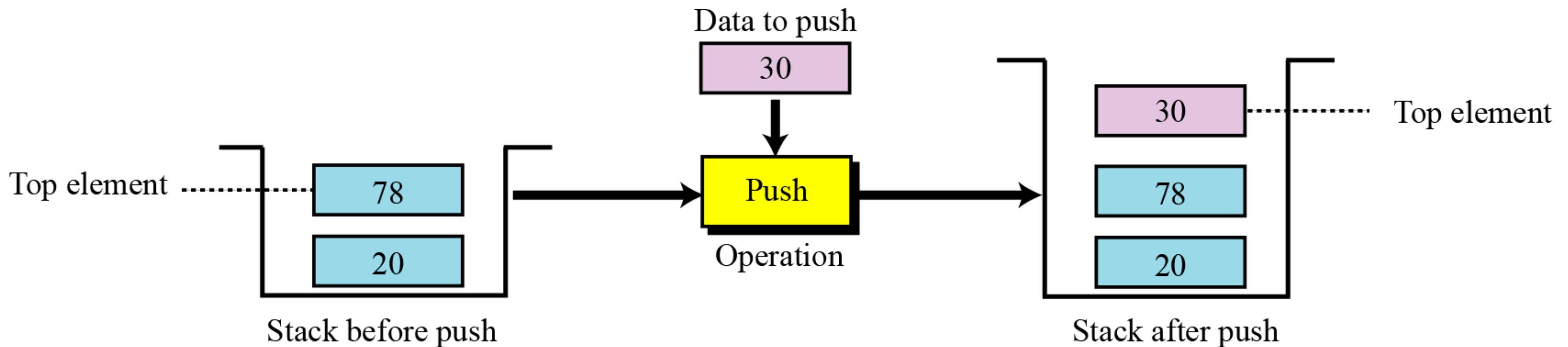
stack (stackName)



The *push* operation

The *push* operation inserts an item at the top of the stack. The following shows the format.

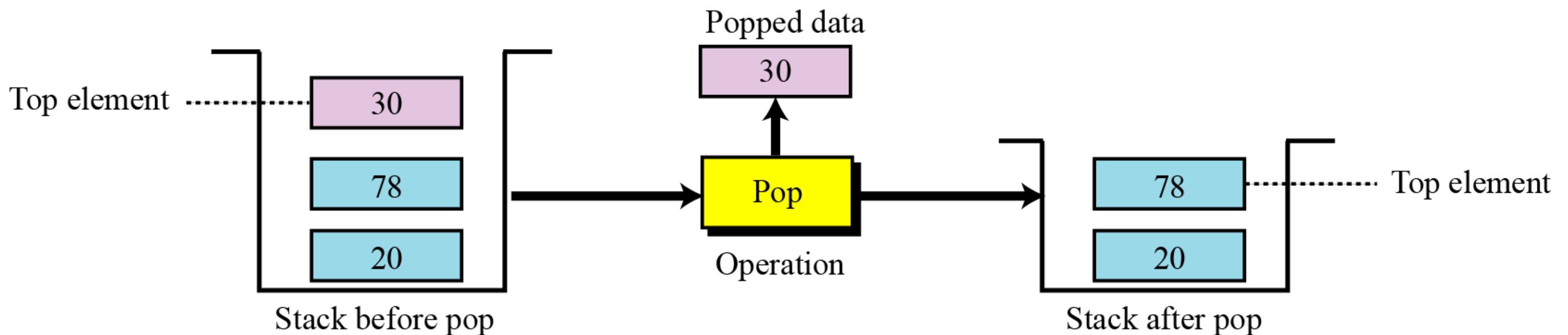
```
push (stackName, dataItem)
```



The *pop* operation

The *pop* operation deletes the item at the top of the stack. The following shows the format.

pop (stackName, dataItem)



The *empty* operation

The *empty* operation checks the status of the stack. The following shows the format.

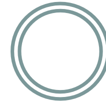
empty (stackName)

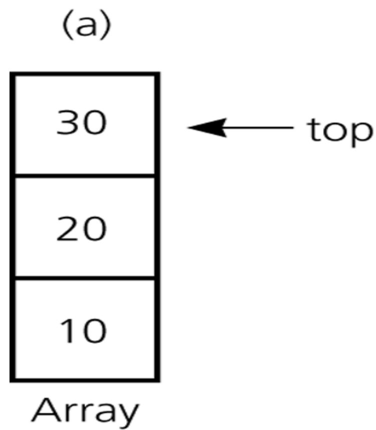
This operation returns true if the stack is empty and false if the stack is not empty.

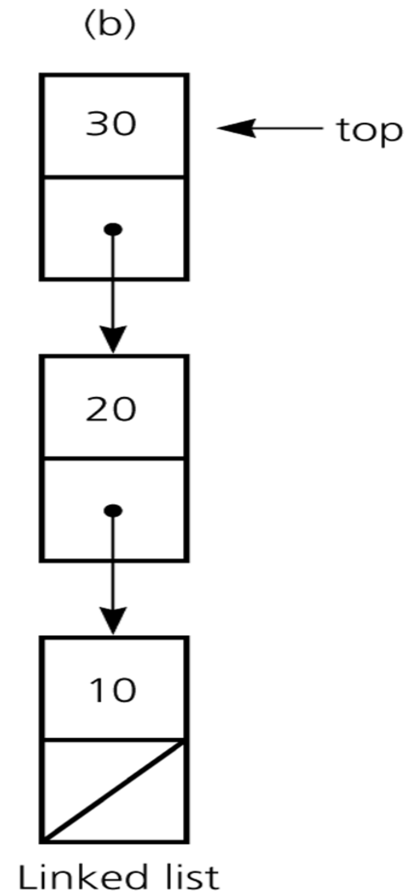
Implementations of the Stack

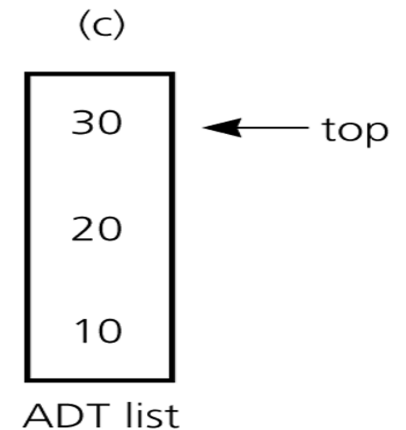
- The stack can be implemented using
  - An array
  - A linked list
  - The ADT list
- All three implementations use a `StackException` class to handle possible exceptions

Implementations of the Stack









Array-based Stack Implementation

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Bottom stack element stored at element 0
- last index in the array is the *top*
- Increment *top* when one element is pushed, decrement after pop

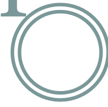
Implementing Stacks: Array

- Advantages
  - best performance
- Disadvantage
  - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack

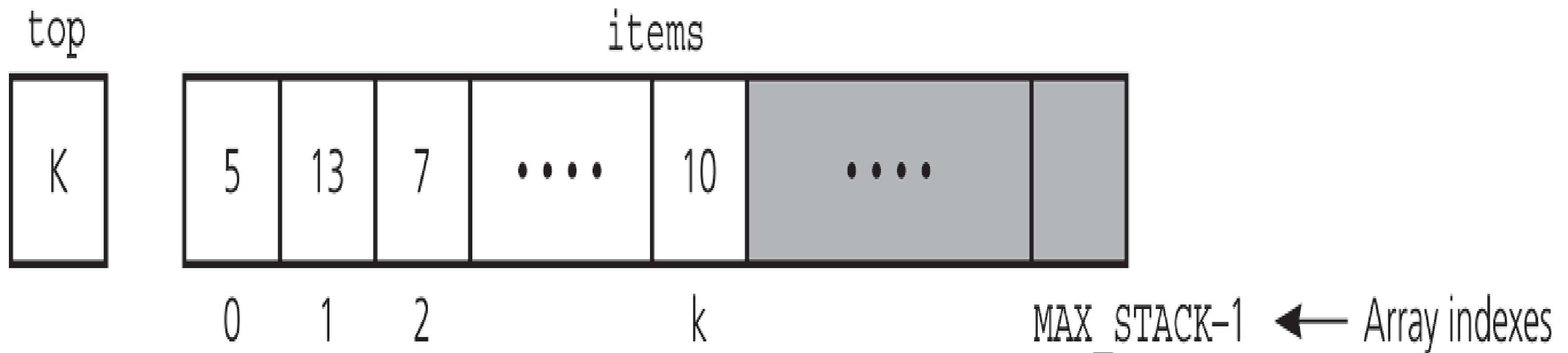
Implementing Operations

- Constructor
  - Compiler will handle allocation of memory
- Empty
  - Check if value of **`myTop == -1`**
- Push (if **`myArray`** not full)
  - Increment **`myTop`** by 1
  - Store value in **`myArray [myTop]`**
- Top
  - If stack not empty, return **`myArray[myTop]`**
- Pop
  - If array not empty, decrement **`myTop`**
- Output routine added for testing

An Array-Based Implementation of Stack



- Private data fields
  - An array of `items` of type `StackItemType`
  - The index `top`
- Compiler-generated destructor and copy constructor



Array Based Stack Implementation

12
15
5
2
3
4

stack

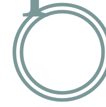
[0] array

What are array values?

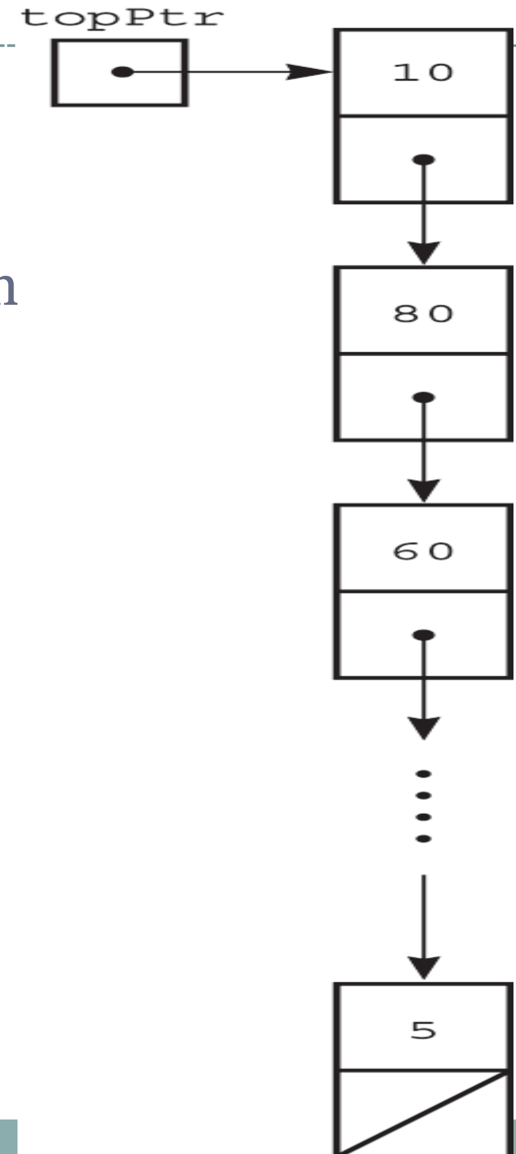
s.push (20)

s.pop( )

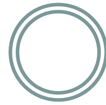
A Pointer-Based Implementation of Stack

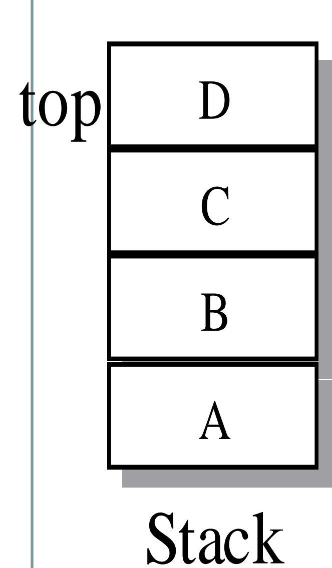


- A pointer-based implementation
  - Required when the stack needs to grow an shrink dynamically
- `top` is a reference to the head of a linked list of items
- A copy constructor and destructor must be supplied

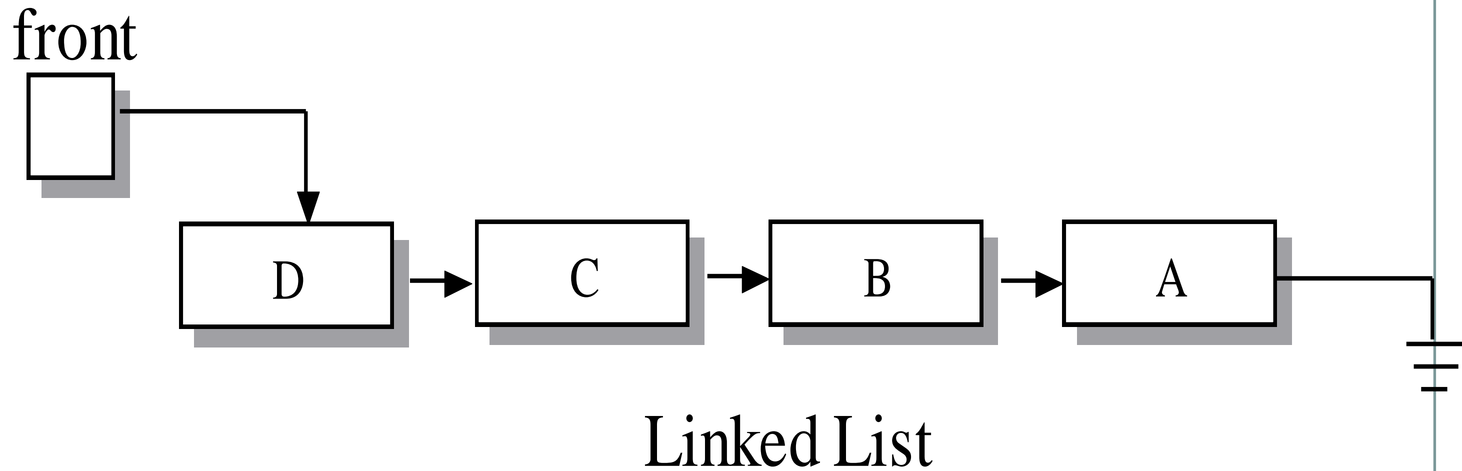


Linked List Implementation





Stack



Linked List

s.push (‘F’)

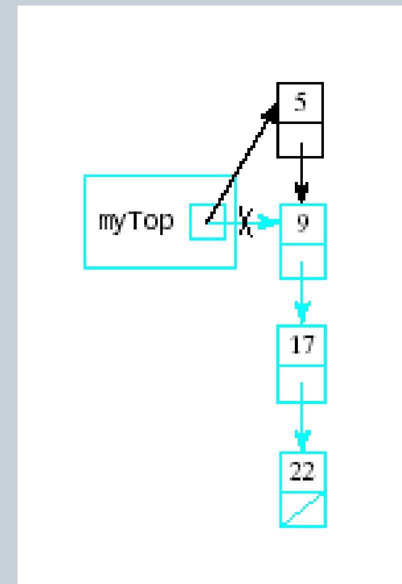
s.pop( )

Implementing a Stack: Linked List

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is the slowest of all the implementations
  - can grow to an infinite size
- Basic implementation
  - list is initially empty
  - *push()* method adds a new item to the head of the list
  - *pop()* method removes the head of the list

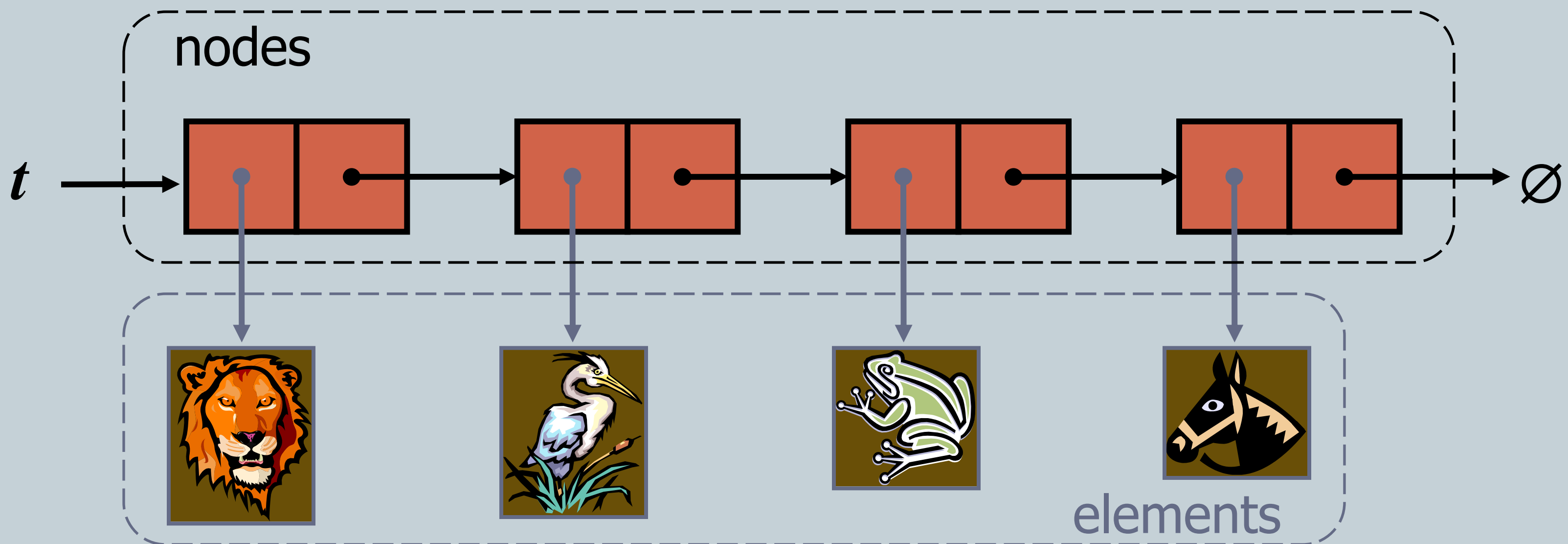
Implementing Linked Stack Operations

- Constructor
  - Simply assign null pointer to `myTop`
- Empty
  - Check for `myTop == null`
- Push
  - Insertion at beginning of list
- Top
  - Return data to which `myTop` points



Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $\boldsymbol{O}(\boldsymbol{n})$ and each operation of the Stack ADT takes $\boldsymbol{O}(1)$ time



An Implementation That Uses the ADT List

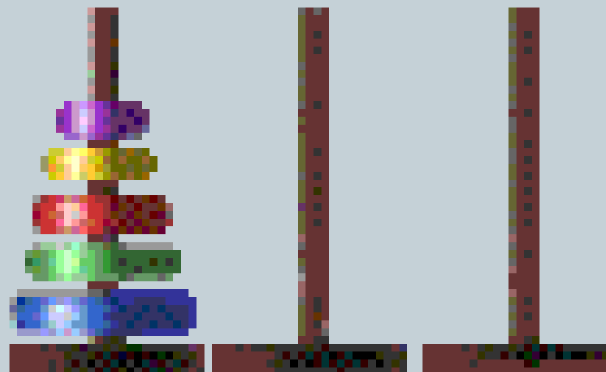
- The ADT list can be used to represent the items in a stack
- If the item in position 1 is the top
  - `push(newItem)`
    - `insert(1, newItem)`
  - `pop()`
    - `remove(1)`
  - `getTop(stackTop)`
    - `retrieve(1, stackTop)`

Stack Applications

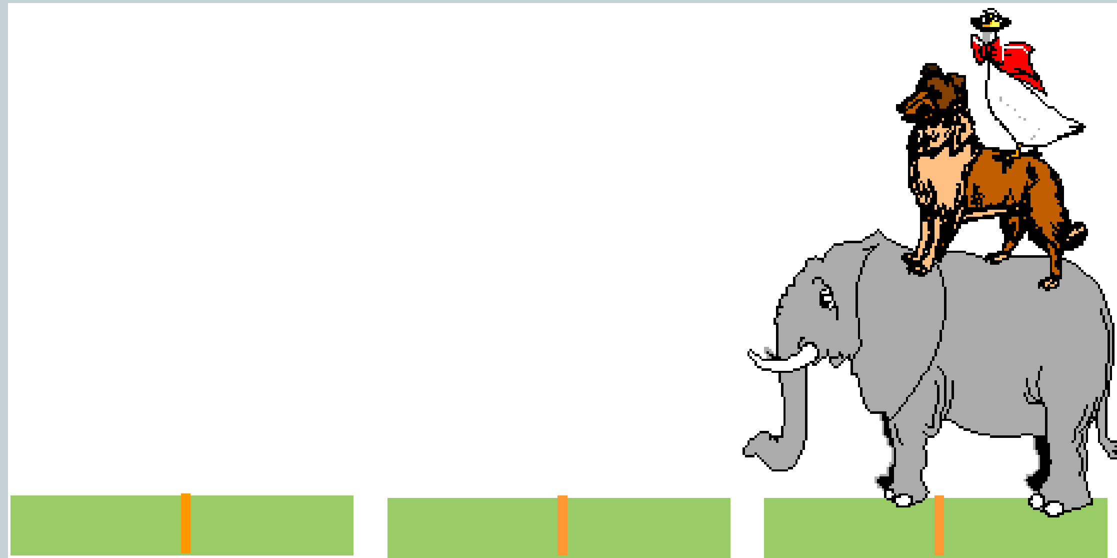
- Real life
  - Pile of books
  - Plate trays
- More applications related to computer science
  - Program execution stack (read more from your text)
  - Evaluating expressions

The Towers of Hanoi
A Stack-based Application

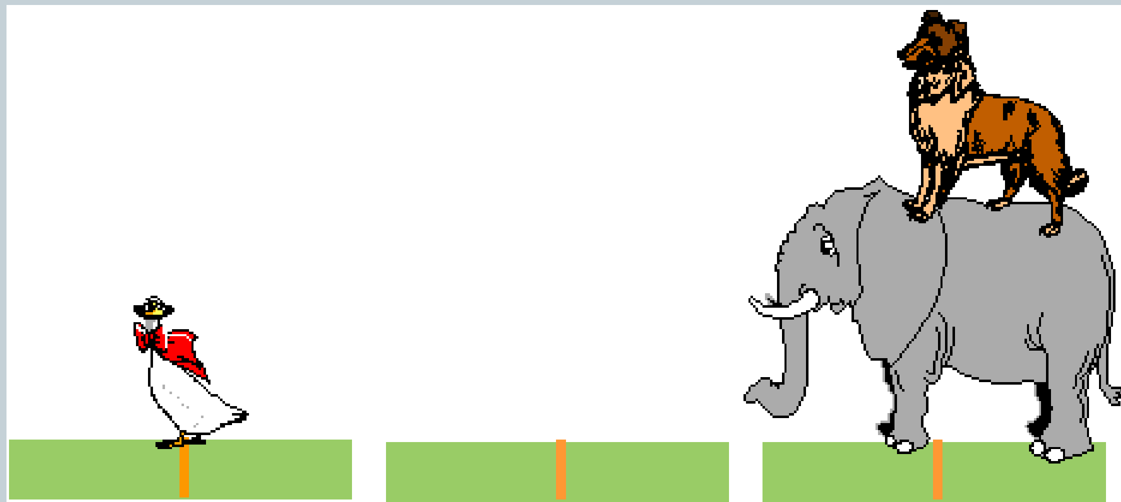
- GIVEN: three poles
- a set of discs on the first pole, discs of different sizes, the smallest discs at the top
- GOAL: move all the discs from the left pole to the right one.
- CONDITIONS: only one disc may be moved at a time.
- A disc can be placed either on an empty pole or on top of a larger disc.



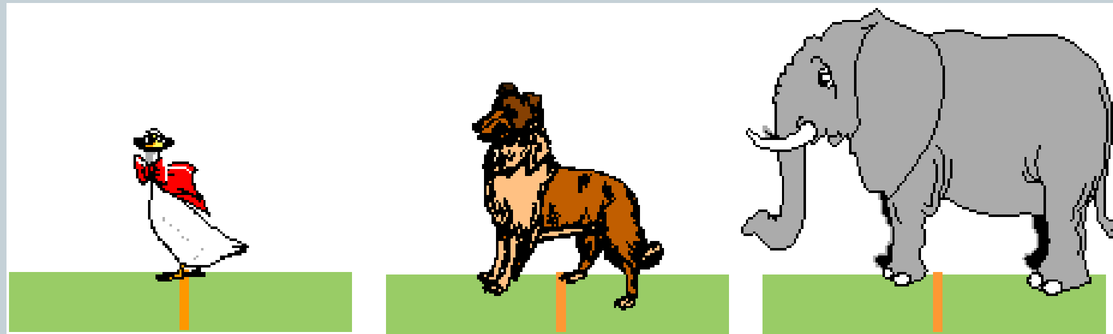
Towers of Hanoi



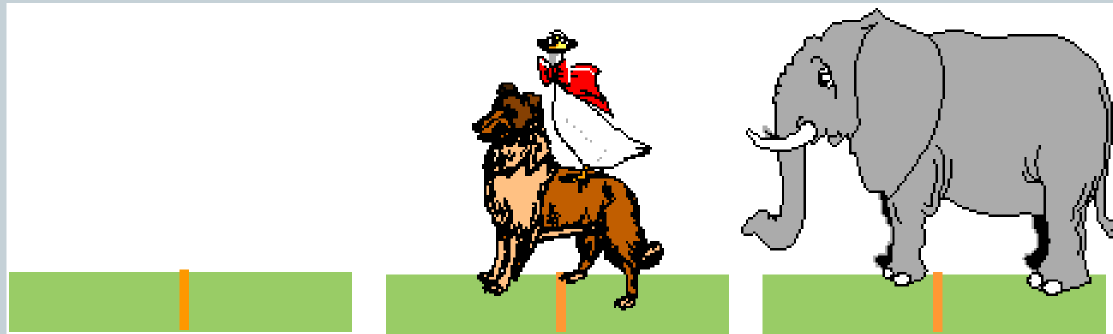
Towers of Hanoi



Towers of Hanoi

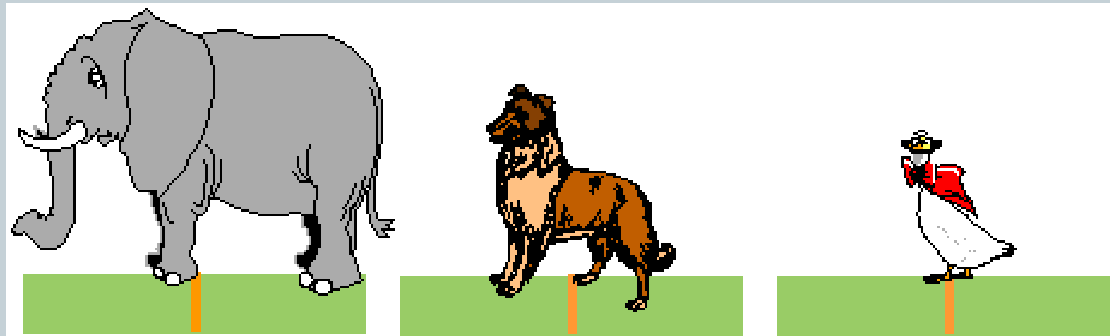


Towers of Hanoi

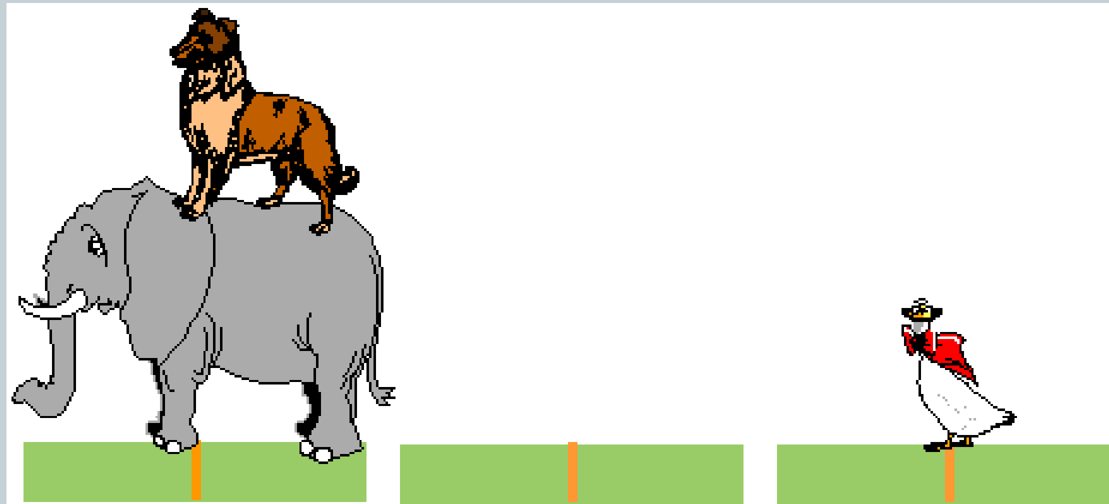


Towers of Hanoi

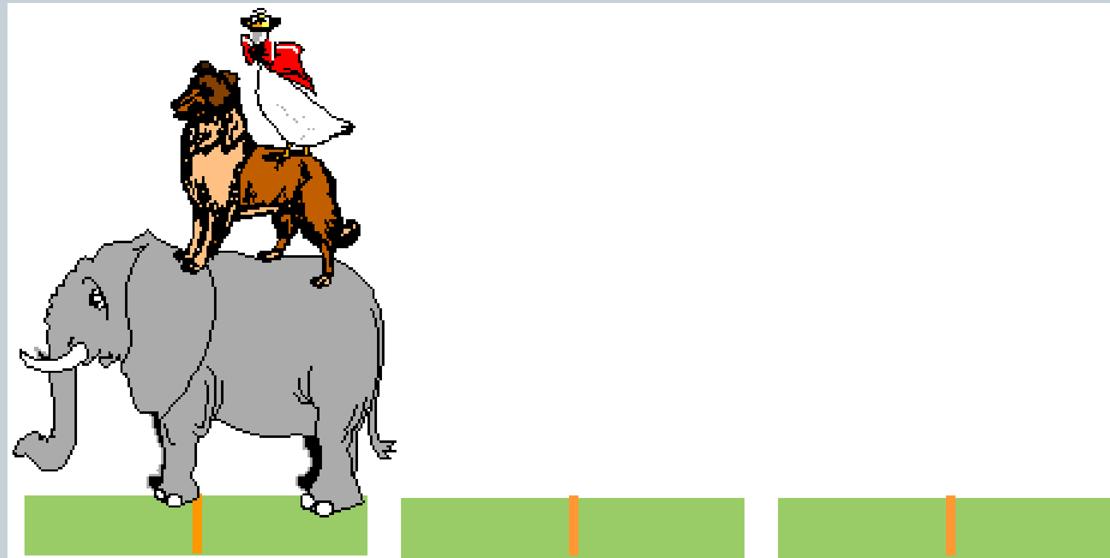
Towers of Hanoi



Towers of Hanoi



Towers of Hanoi



Some more applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Real life examples where stacks are used:-

a) Processing of **procedure calls** and their termination.

b) In a **recursive call** of a function.

c) When a person wear **bangles** the last bangle worn is the first one to be removed and the first bangle would be the last to be removed. This follows last in first out (LIFO) principle of stack.

(d**) CD's in the case**

e) In a **stack of plates,** once can take out the plate from top or can keep plate at the top. The plate that was placed first would be the last to take out. This follows the LIFO principle of stack.

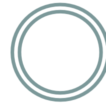
f)**Batteries in the flashlight** :- You cant remove the second battery unless you remove the last in. So the battery that was put in first would be the last one to take out. This follows the LIFO principle of stack.

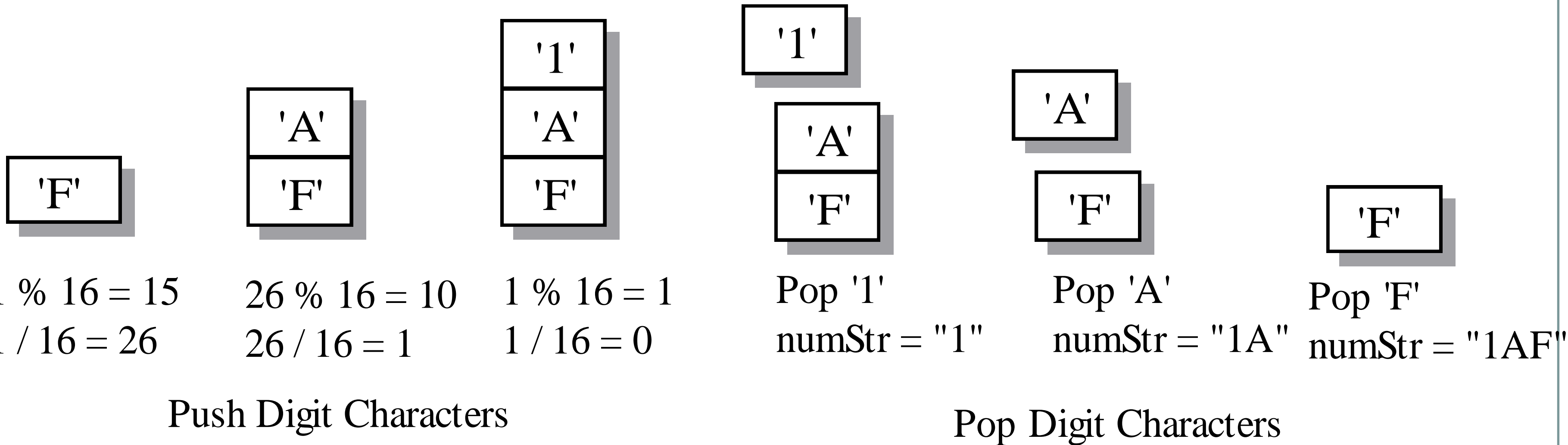
g) **Cars in a garage** :- In order to take out the car that was parked first you need to take out the car that was parked last. So the car that was parked first would be the last to take out. This follows the LIFO principle of stack.

h) **Clothes in the trunk**

i) **Tennis balls in their container.**

Using a Stack to Create a Hex Number





Evaluating Postfix Expressions

- A postfix calculator
  - When an operand is entered, the calculator
    - Pushes it onto a stack
  - When an operator is entered, the calculator
    - Applies it to the top two operands of the stack
    - Pops the operands from the stack
    - Pushes the result of the operation on the stack

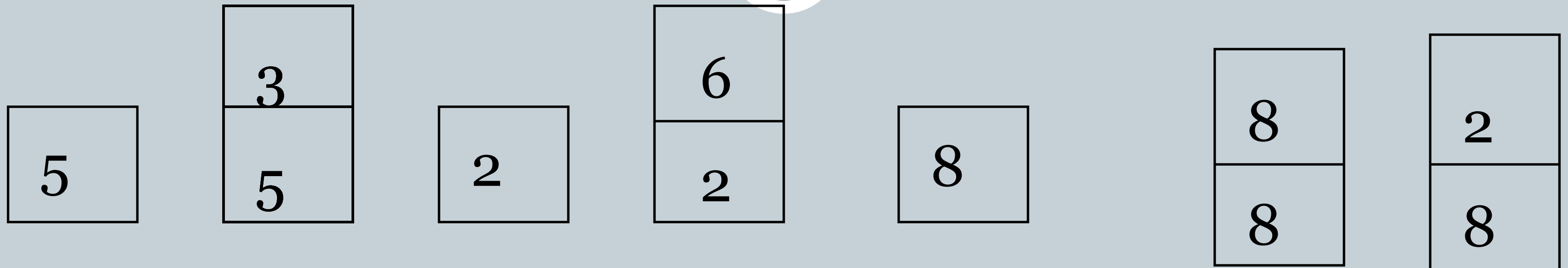
Evaluating Postfix Expressions

- To evaluate a postfix expression which is entered as a string of characters
  - Simplifying assumptions
    - The string is a syntactically correct postfix expression
    - No unary operators are present
    - No exponentiation operators are present
    - Operands are single lowercase letters that represent integer values

Algorithm for Postfix evaluation

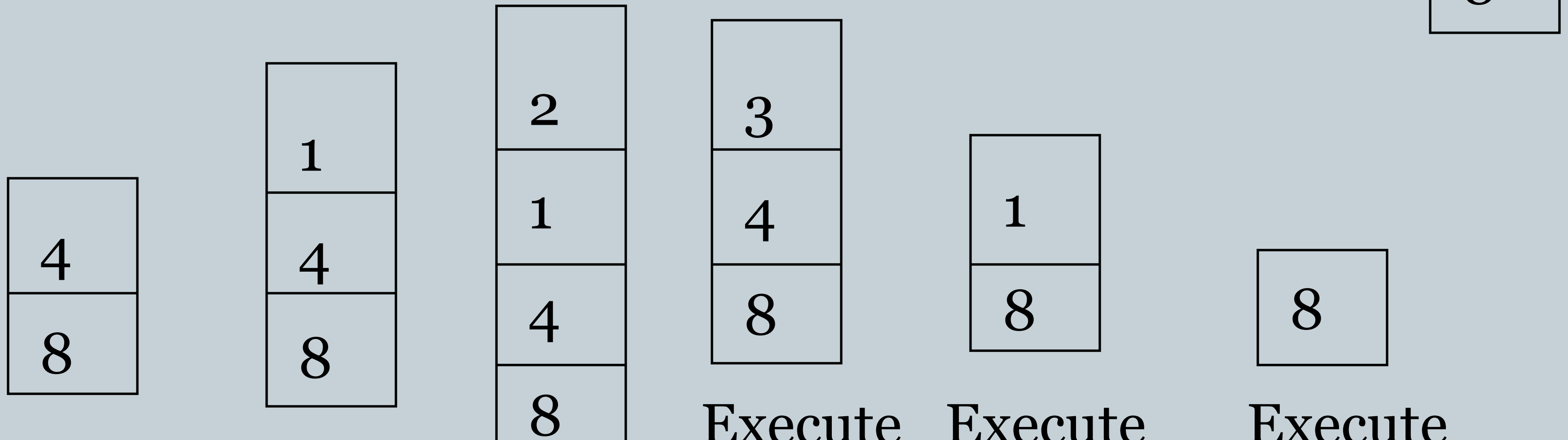
1. Empty the operand stack
2. **while** there are more tokens
3. Get the next token
4. **if** the first character of the token is a digit
5. Push the integer onto the stack
6. **else if** the token is an operator
7. Pop the right operand off the stack
8. Pop the left operand off the stack
9. Evaluate the operation
10. Push the result onto the stack
11. Pop the stack and return the result

Expression: 5 3 - 6 + 8 2 / 1 2 + - *



Execute 5 – 3

Execute 2 + 6



Execute 8 / 2

Execute 1 + 2

Execute 4 - 3

Execute 8 * 1

Is the End of the World Approaching?

- Problem complexity 2^n
- 64 gold discs
- Given 1 move a second

→ 600,000,000,000 years until the end of the world ☺

Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces

    `abc{defg{ijk}{l{mn}}op}qr`
  - An example of unbalanced braces

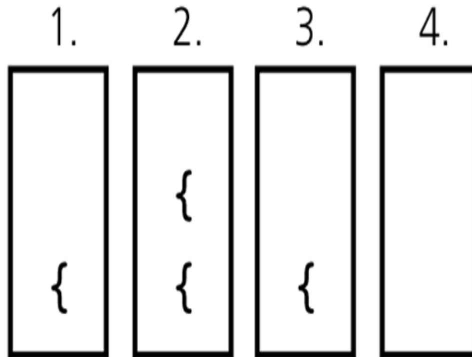
    `abc{def}}{ghij{kl}m`

- Requirements for balanced braces
  - Each time you encounter a "}", it matches an already encountered "{"
  - When you reach the end of the string, you have matched each "{"

Checking for Balanced Braces

Input string | Stack as algorithm executes

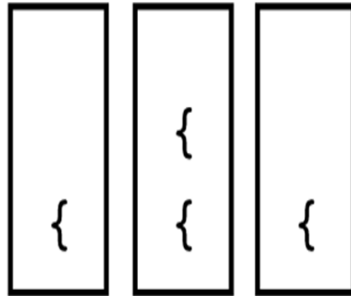
{a{b}c}



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

Stack empty ⟹ balanced

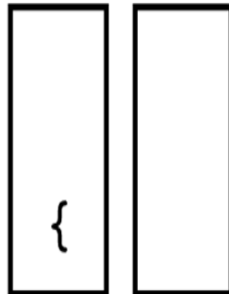
{a{bc}



1. push "{"
2. push "{"
3. pop

Stack not empty ⟹ not balanced

{ab}c}



1. push "{"
2. pop

Stack empty when last "}" encountered ⟹ not balanced

Converting Infix Expressions to Equivalent Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only “to the right” with respect to the operands
  - All parentheses are removed

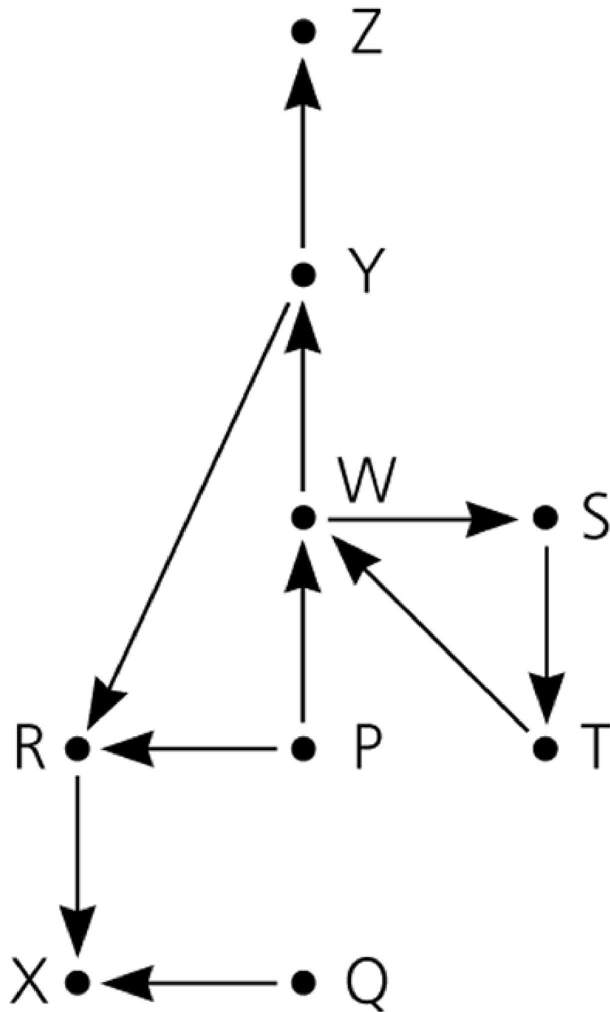
*a - (b + c * d)/e*

ch	Stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	`postfixExp` until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to `postfixExp`

Application: A Search Problem

- High Planes Airline Company (HPAir)
  - For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city
- The flight map for HPAir is a graph
  - Adjacent vertices are two vertices that are joined by an edge
  - A directed path is a sequence of directed edges

Application: A Search Problem



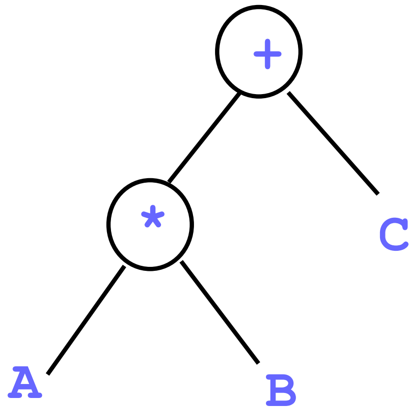
Flight map for HPAir

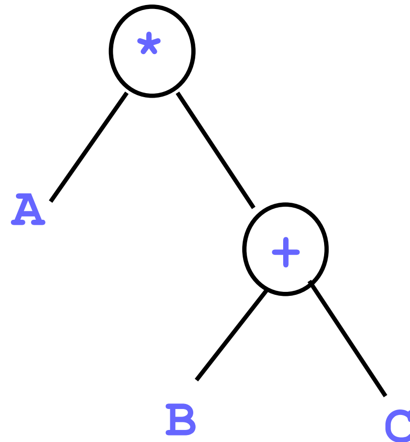
Converting Infix to Postfix

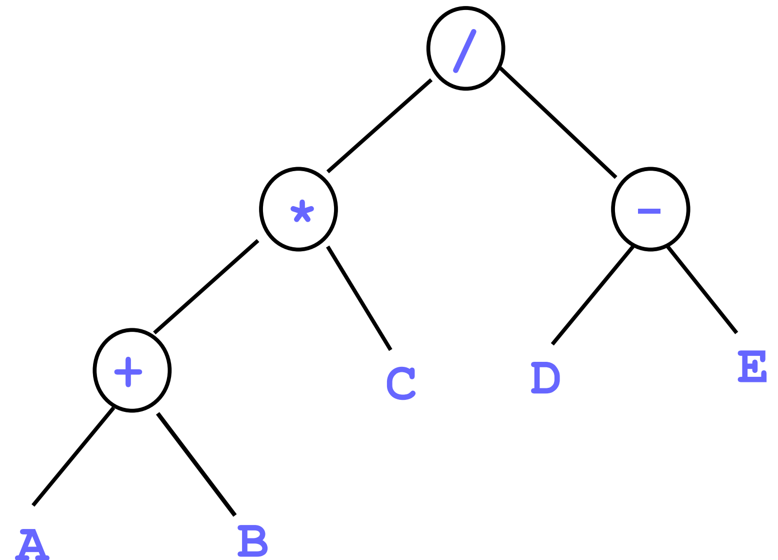
- *Operands* are in same order in infix and postfix
- *Operators* occur later in postfix
- Strategy:
  - Send operands straight to output
  - Send higher precedence operators first
  - If same precedence, send in left to right order
  - Hold pending operators on a stack

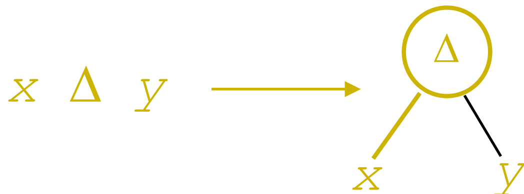
Converting Infix to Prefix

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

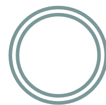




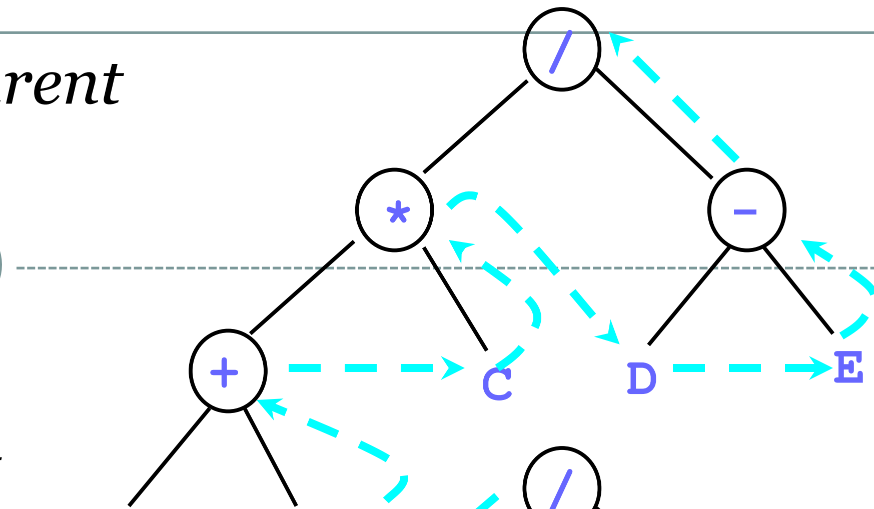




Traverse the tree in *Left-Right-Parent* order (*post-order*) to get RPN:

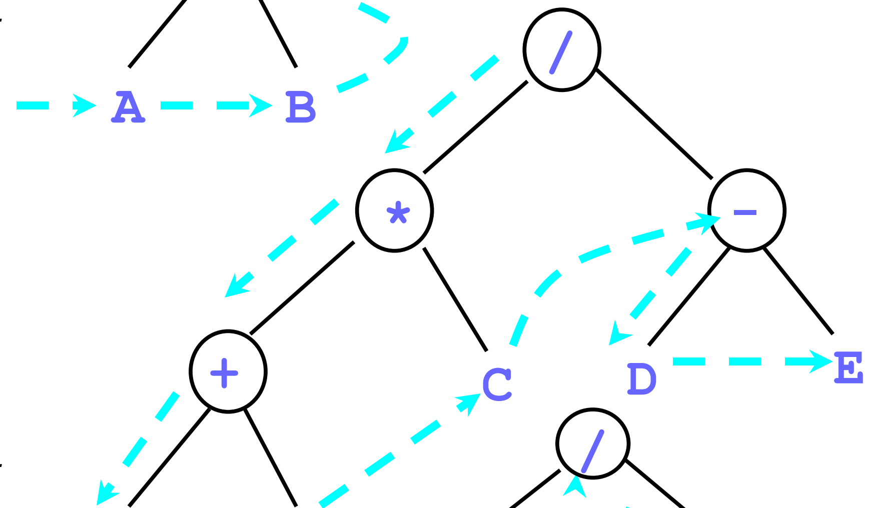


A B + C * D E - /



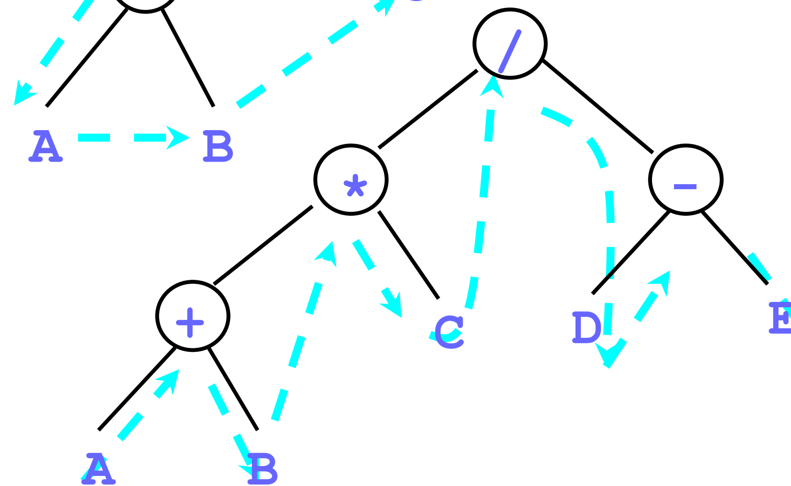
Traverse tree in *Parent-Left-Right* order (*pre-order*) to get **prefix:**

/ * + A B C - D E



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

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



A Non-recursive Solution That Uses a Stack

- The solution performs an exhaustive search
  - Beginning at the origin city, the solution will try every possible sequence of flights until either
    - It finds a sequence that gets to the destination city
    - It determines that no such sequence exists
- Backtracking can be used to recover from a wrong choice of a city

A Recursive Solution

- Possible outcomes of the recursive search strategy
  - You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination
  - You reach a city C from which there are no departing flights
  - You go around in circles

The Relationship Between Stacks and Recursion

- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a non-recursive version of a recursive algorithm

Recursion

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first

- Recursion is a technique that solves a problem by solving a smaller problem of the same type

- A procedure that is defined in terms of itself

Recursion

When you turn that into a program, you end with functions that call themselves:

<u>Recursive Functions</u>

What's behind this function ?

```
public int f(int a){
        if (a==1)
          return(1);
        else
          return(a * f( a-1));
}
```

It computes f! (factorial)

Factorial

<u>Factorial:</u>

$$a! = 1 * 2 * 3 * \ldots * (a-1) * a$$

<u>Note:</u>

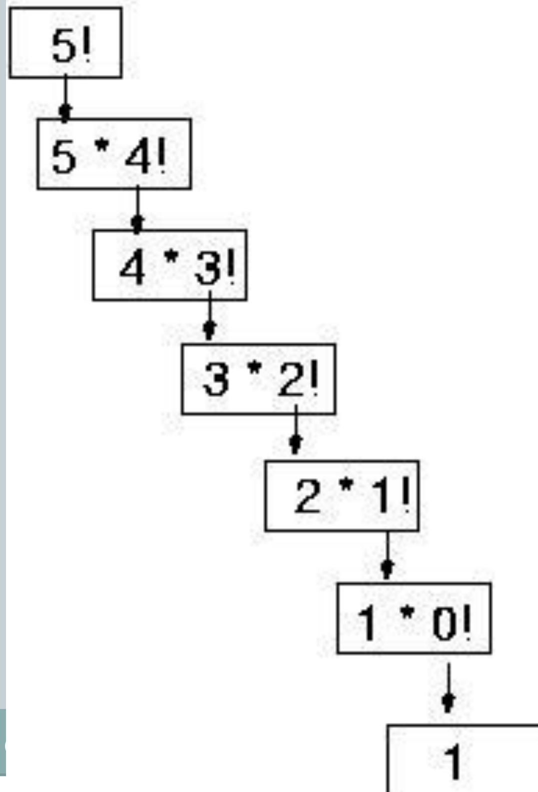
$$a! = a * (a-1)!$$

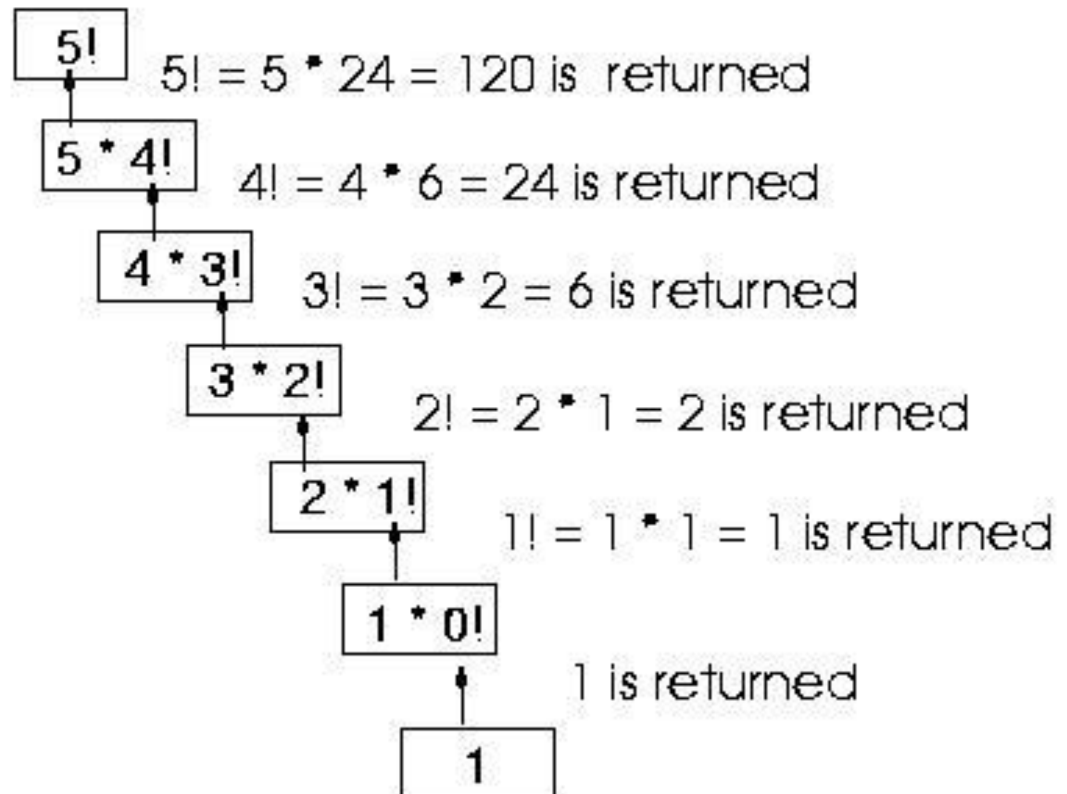
Splitting up the problem into a smaller problem of the same type...

Tracing the example

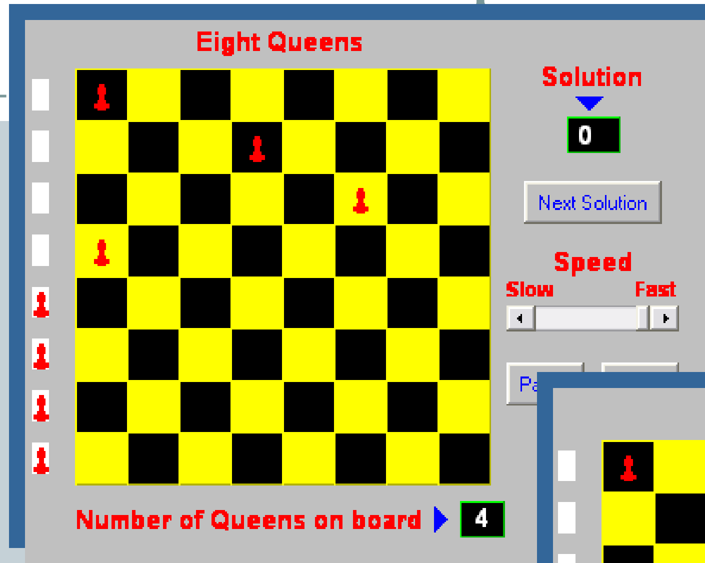
```
public int factorial(int a){
        if (a==0)
          return(1);
        else
          return(a * factorial( a-1));
}
```

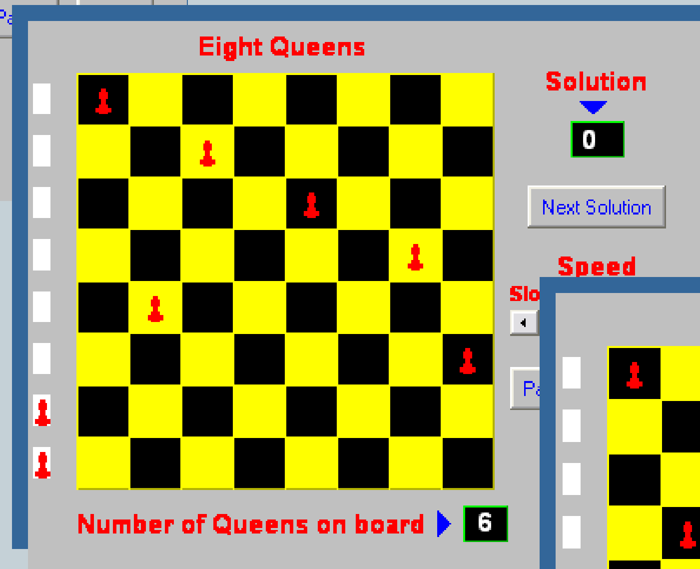
RECURSION !

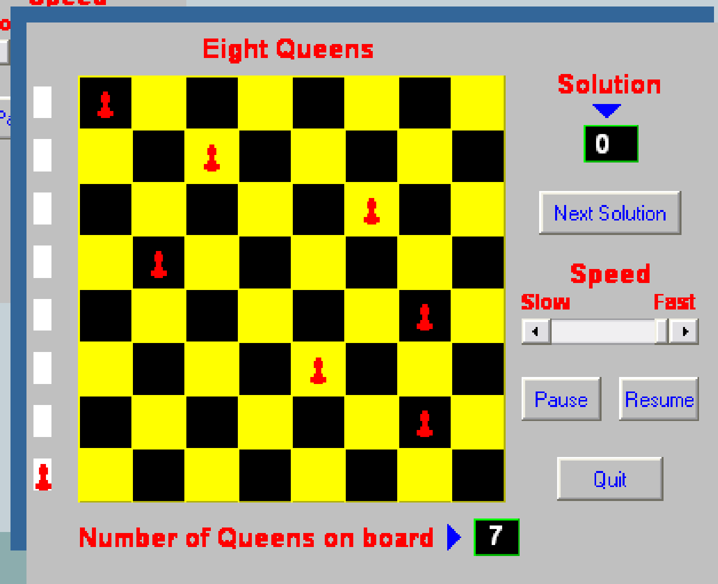




Examples: The 8 Queens Problem







Eight queens are to be placed on a chess board in such a way that no queen checks against any other queen