Data Structures

Stacks

Andres Mendez-Vazquez

August 25, 2016

Outline

Introduction

Definition of a stack

It is a linear list where:

- One end is called top
- Other end is called bottom

Additionally, adds and removes are at the top end only.

Introduction

Definition of a stack

It is a linear list where:

- One end is called top
- Other end is called bottom

Additionally, adds and removes are at the top end only.

Introduction

Definition of a stack

It is a linear list where:

- One end is called top
- Other end is called bottom

Additionally, adds and removes are at the top end only.

Introduction

Definition of a stack

It is a linear list where:

- One end is called top
- Other end is called bottom

Additionally, adds and removes are at the top end only.

What is a stack?

First

Stores a set of elements in a particular order.

Second

Stack principle: LAST IN FIRST OUT = LIFO

Meaning

It means: the last element inserted is the first one to be removed

What is a stack?

First

Stores a set of elements in a particular order.

Second

Stack principle: LAST IN FIRST OUT = LIFO

Meaning

It means: the last element inserted is the first one to be removed

What is a stack?

First

Stores a set of elements in a particular order.

Second

Stack principle: LAST IN FIRST OUT = LIFO

Meaning

It means: the last element inserted is the first one to be removed

Example in Real Life

Stack of Coins

Outline

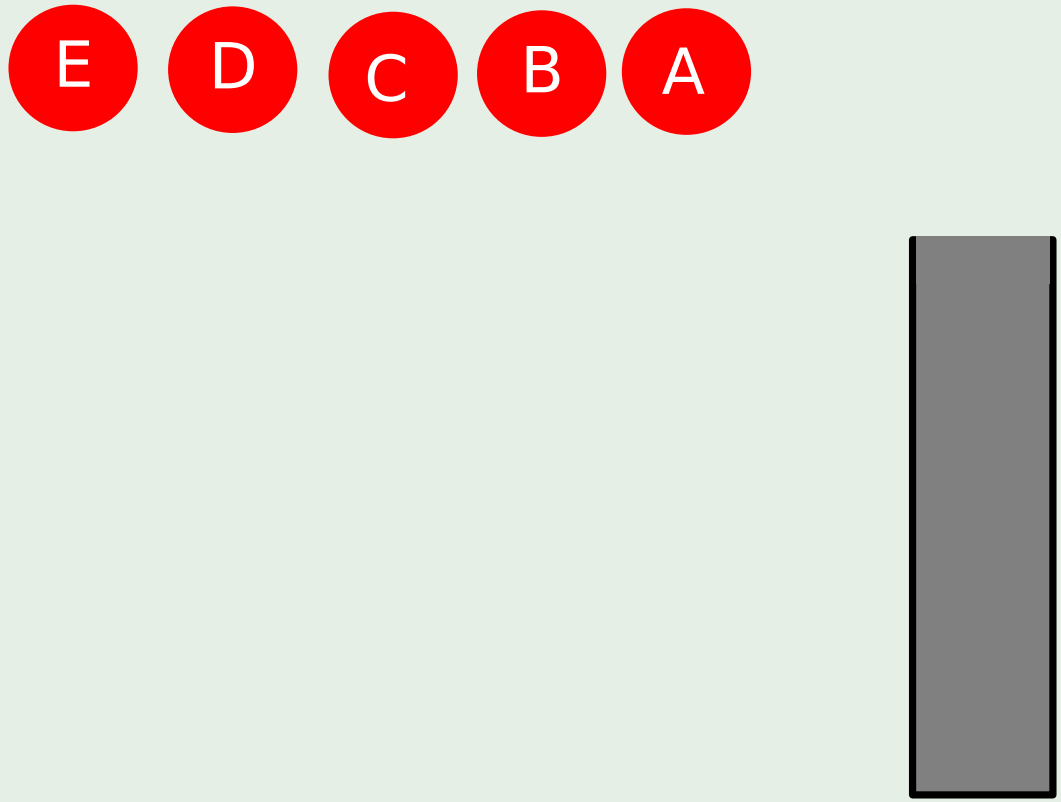
Example

Insert the following items into a stack

List = {A, B, C, D, E}

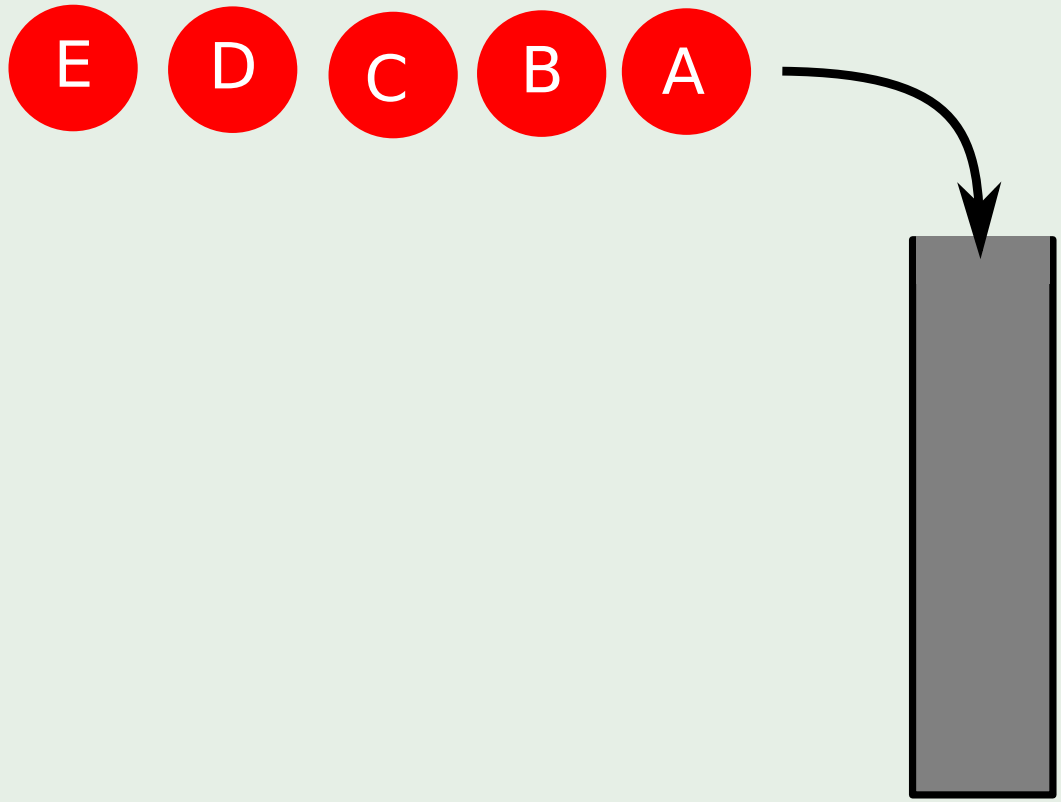
Example

List = {A, B, C, D, E}



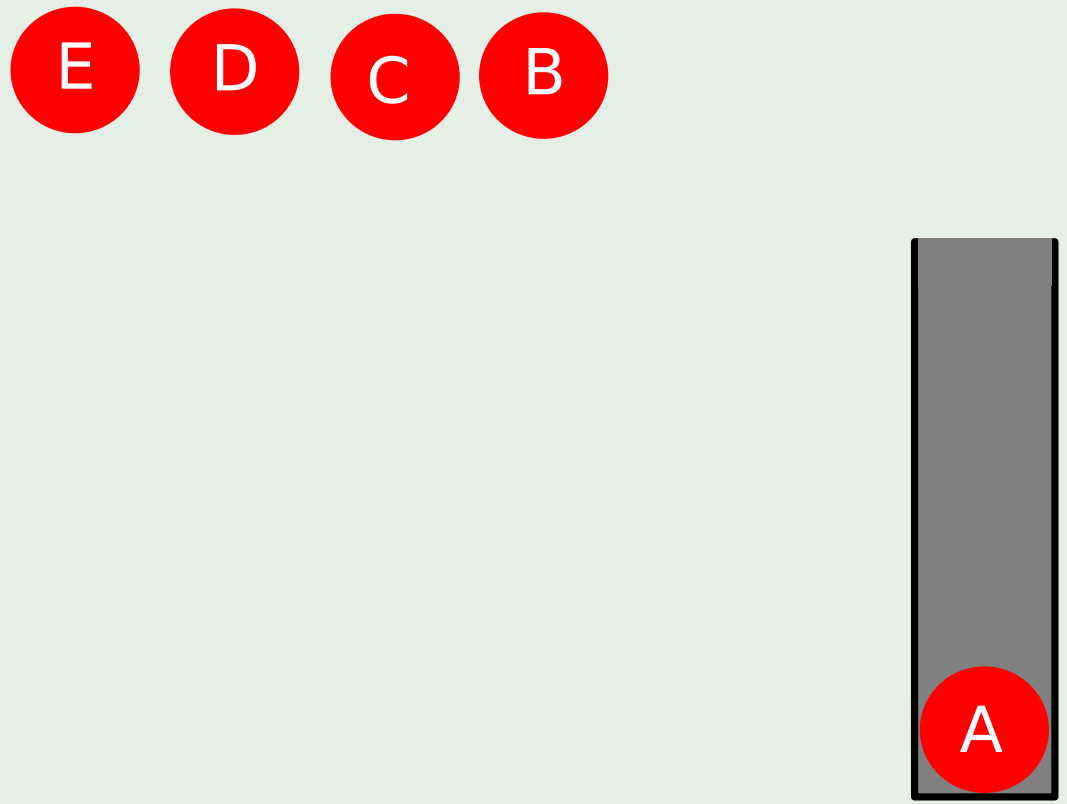
Example

List = {A, B, C, D, E}, Push A



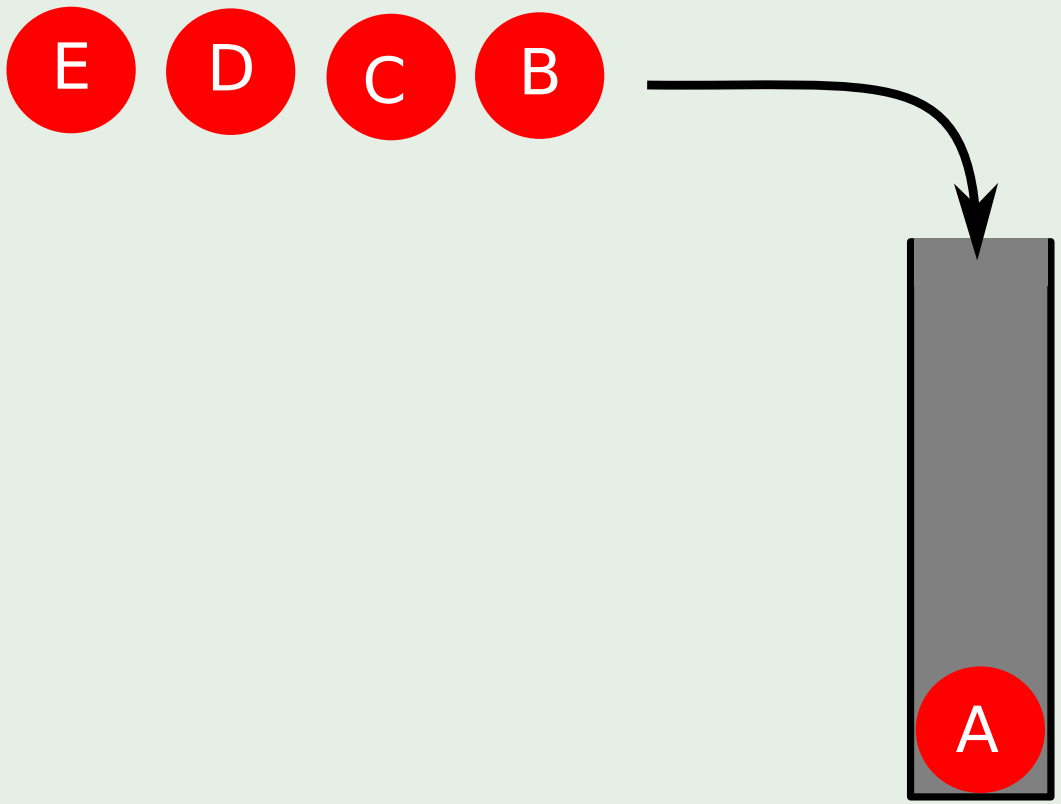
Example

List = {B, C, D, E}



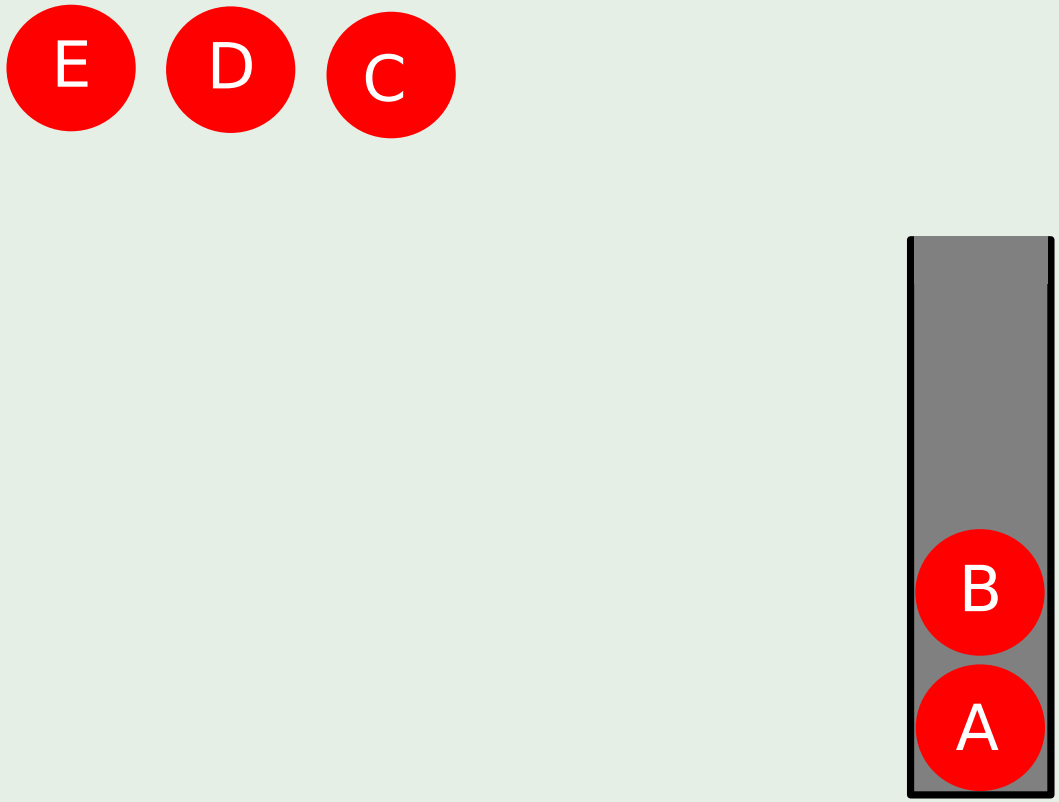
Example

List = {B, C, D, E}, Push B



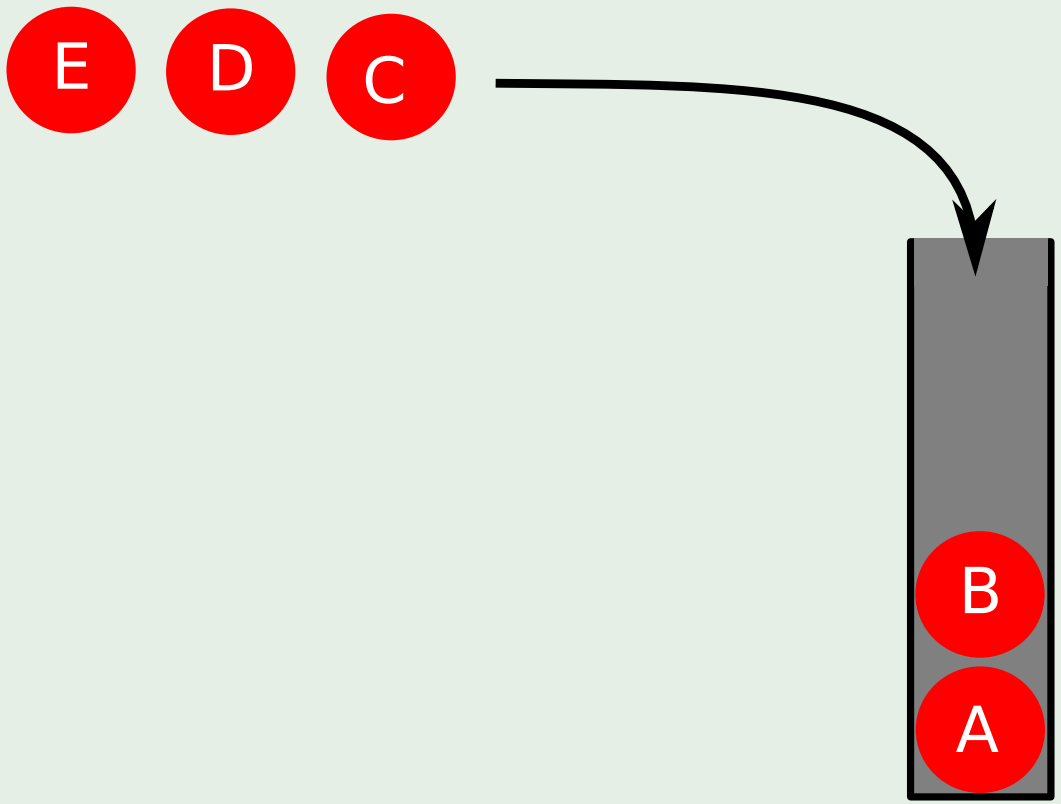
Example

List = {C, D, E}



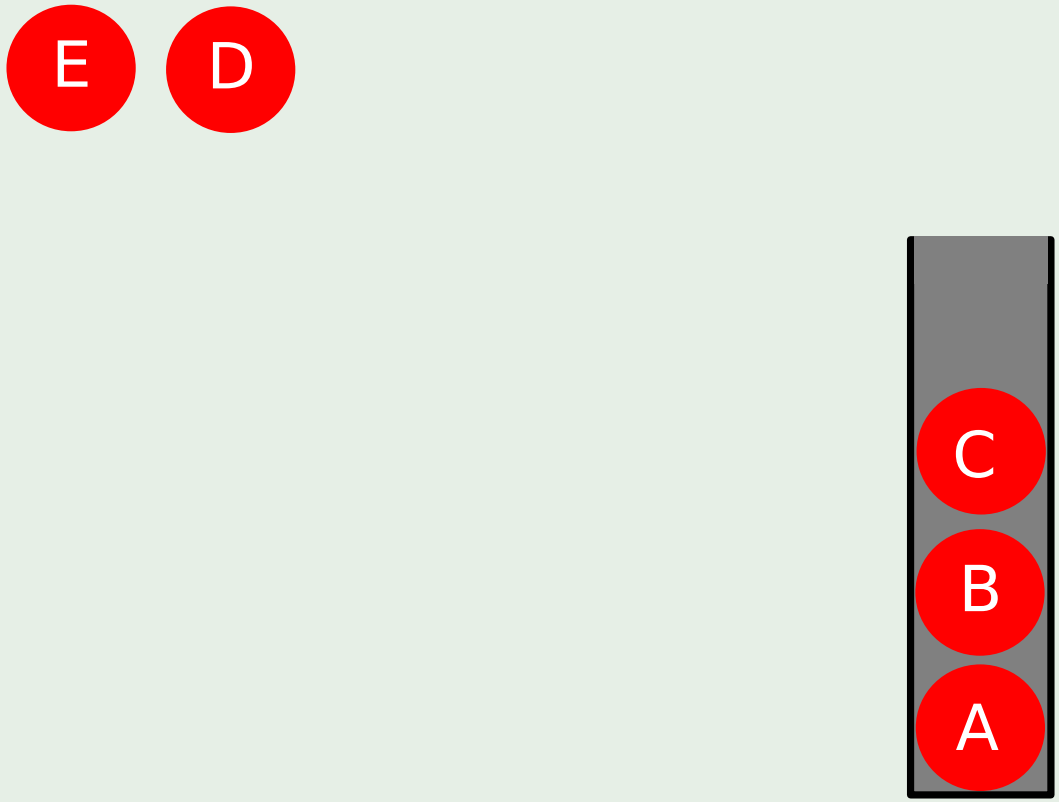
Example

List = {C, D, E}, Push C



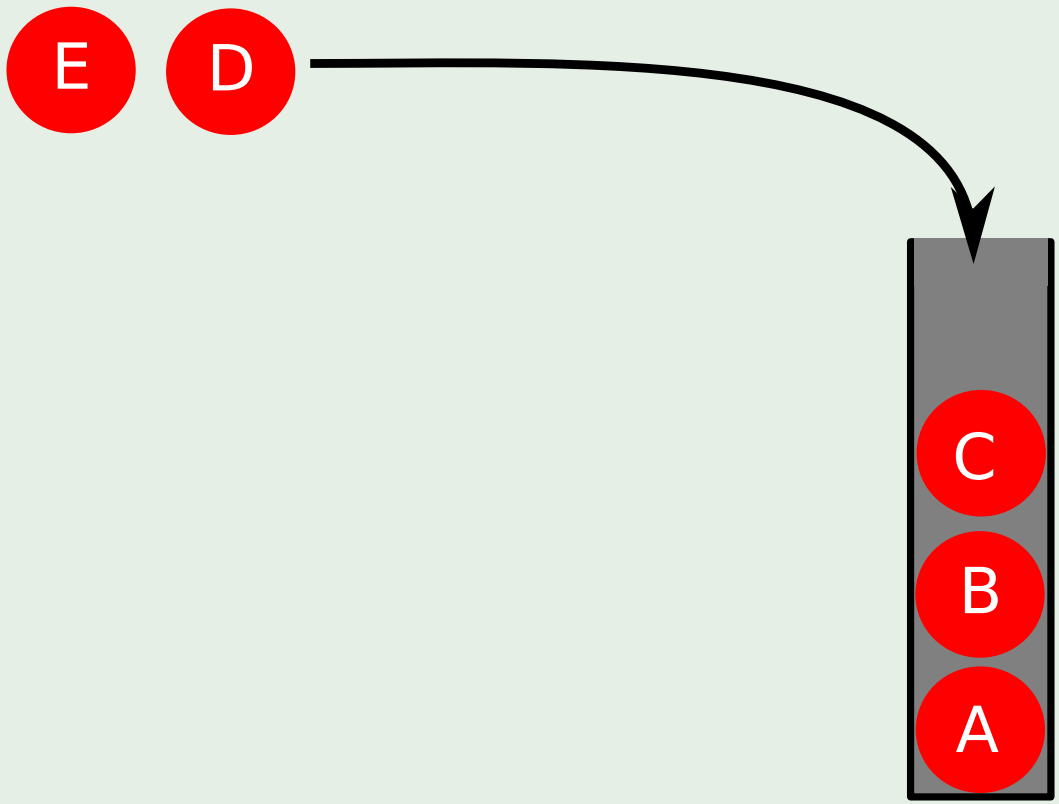
Example

List = {D, E}



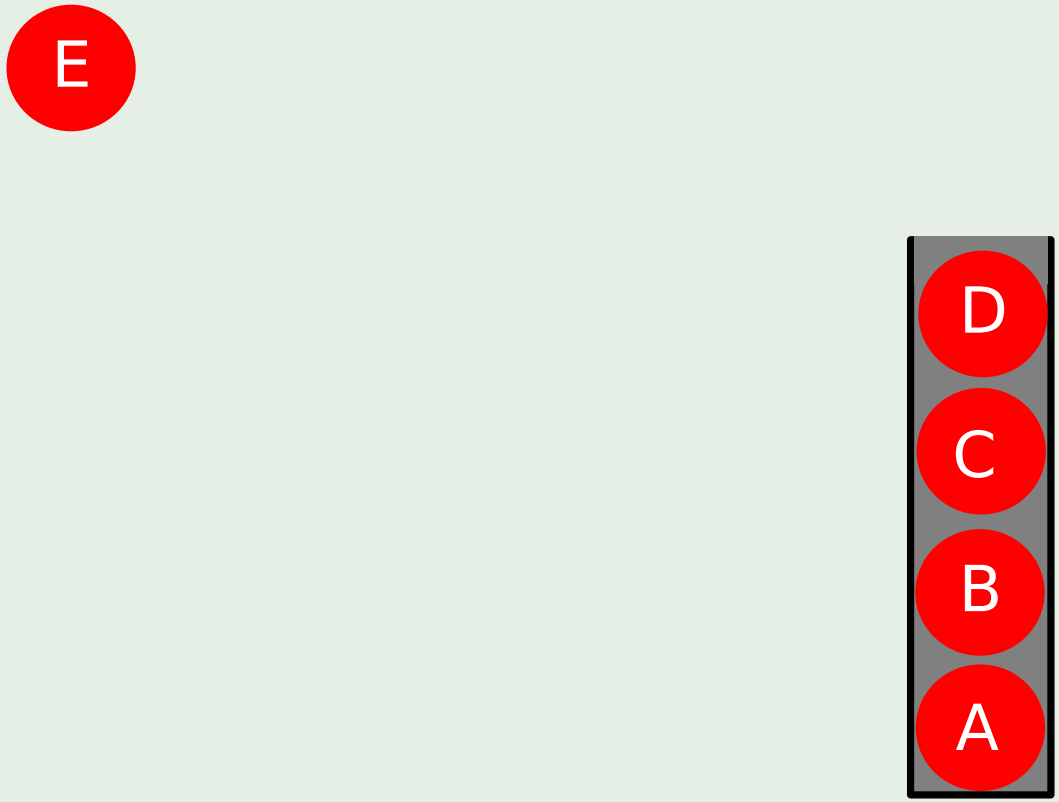
Example

List = {D, E}, Push D



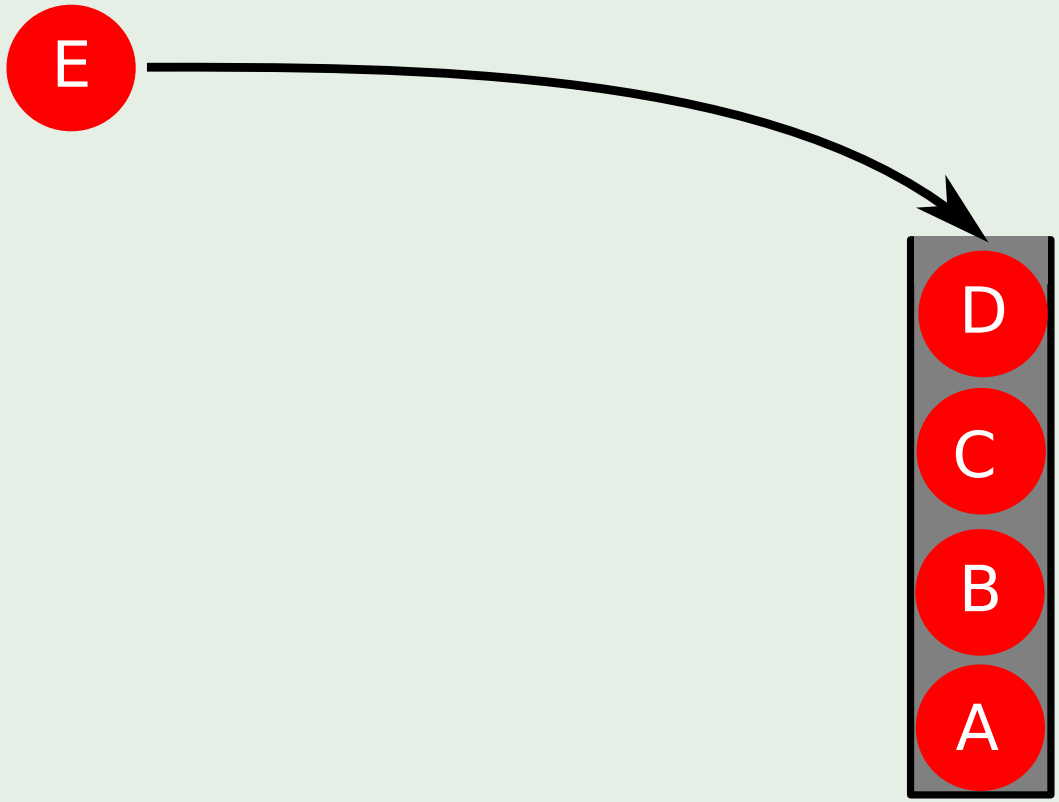
Example

List = {E}



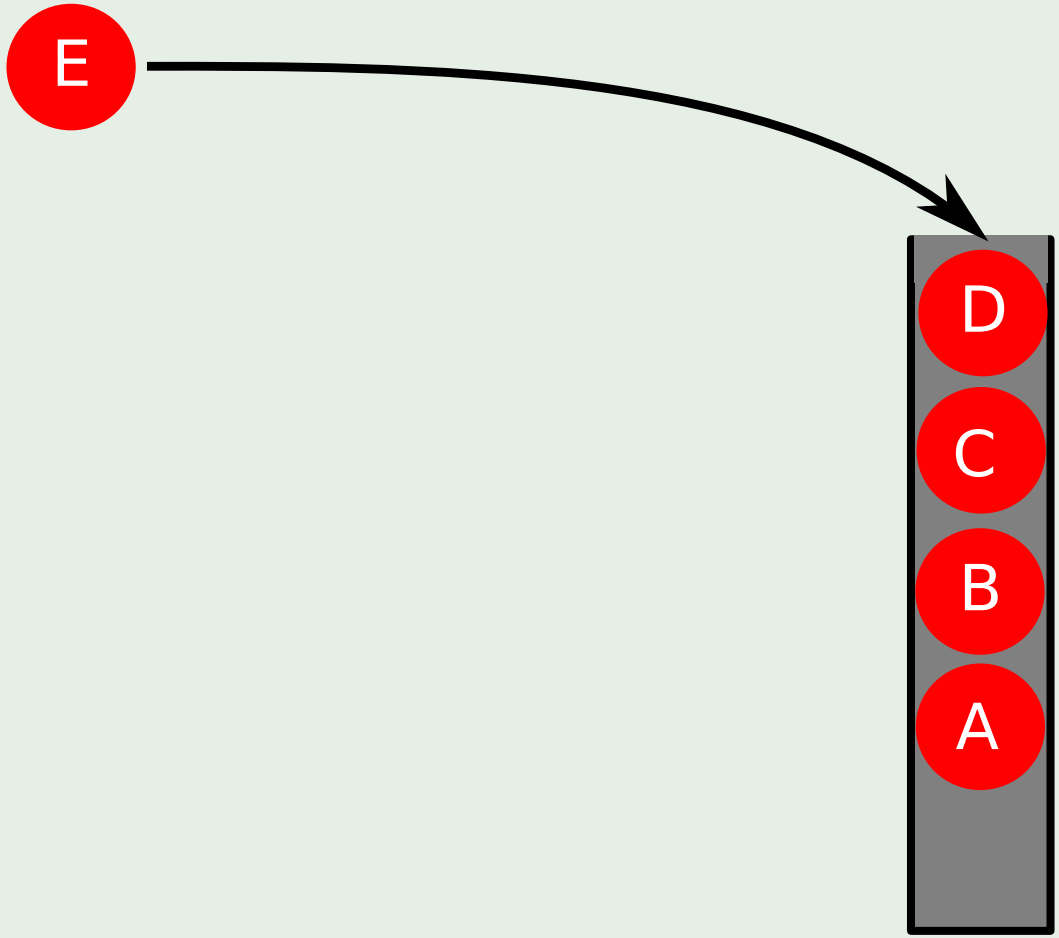
Example

List = {E}, Push E



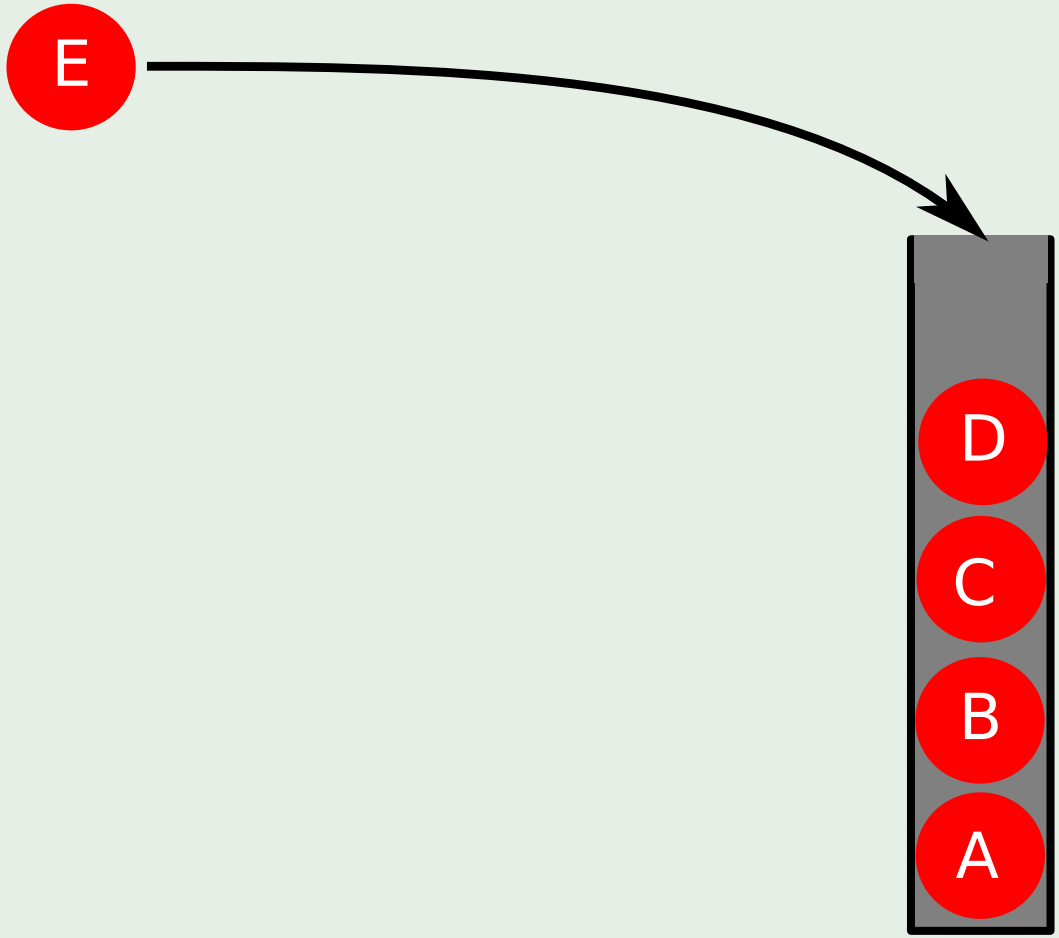
Example

List = {E}, No space!!!Make Space



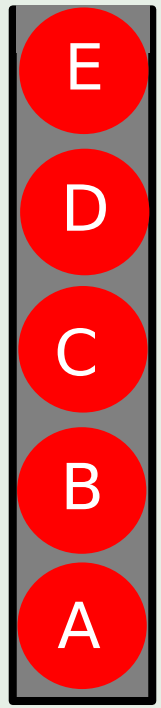
Example

List = {E}, Push E



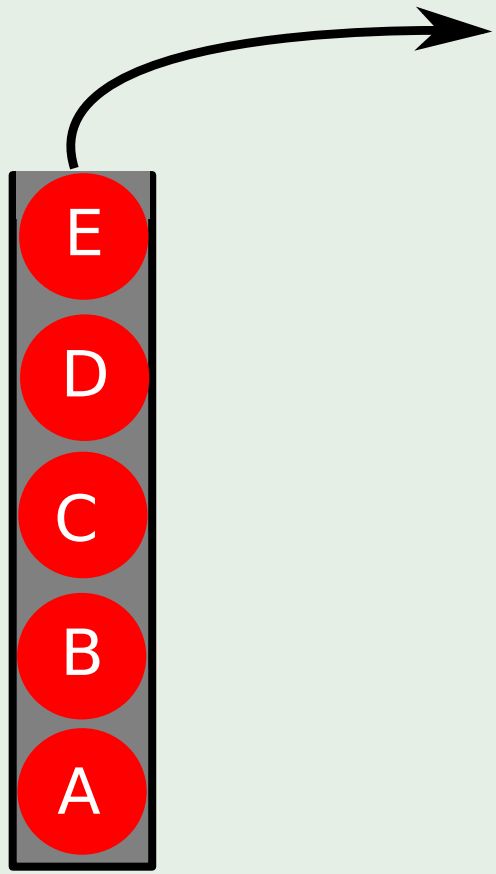
Example

List = {}



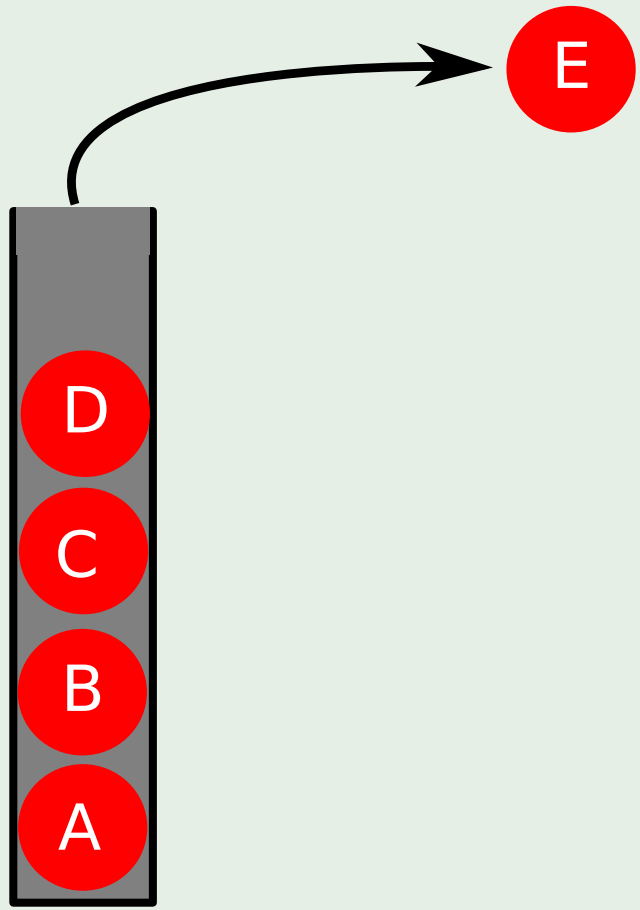
Example

List = {}, Pop



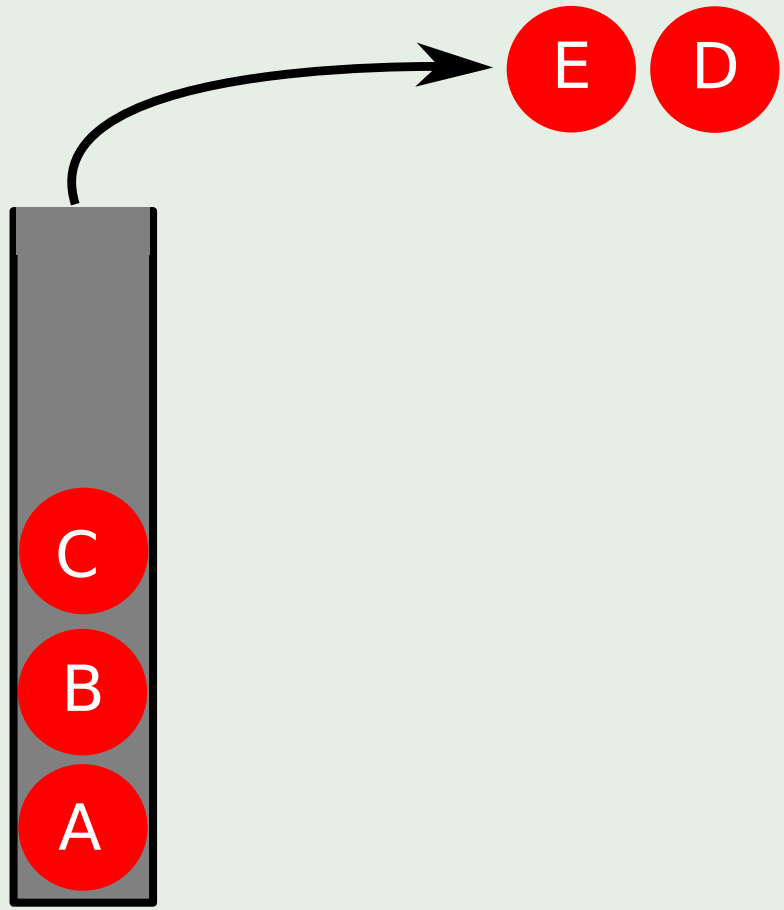
Example

List = {E}, Pop



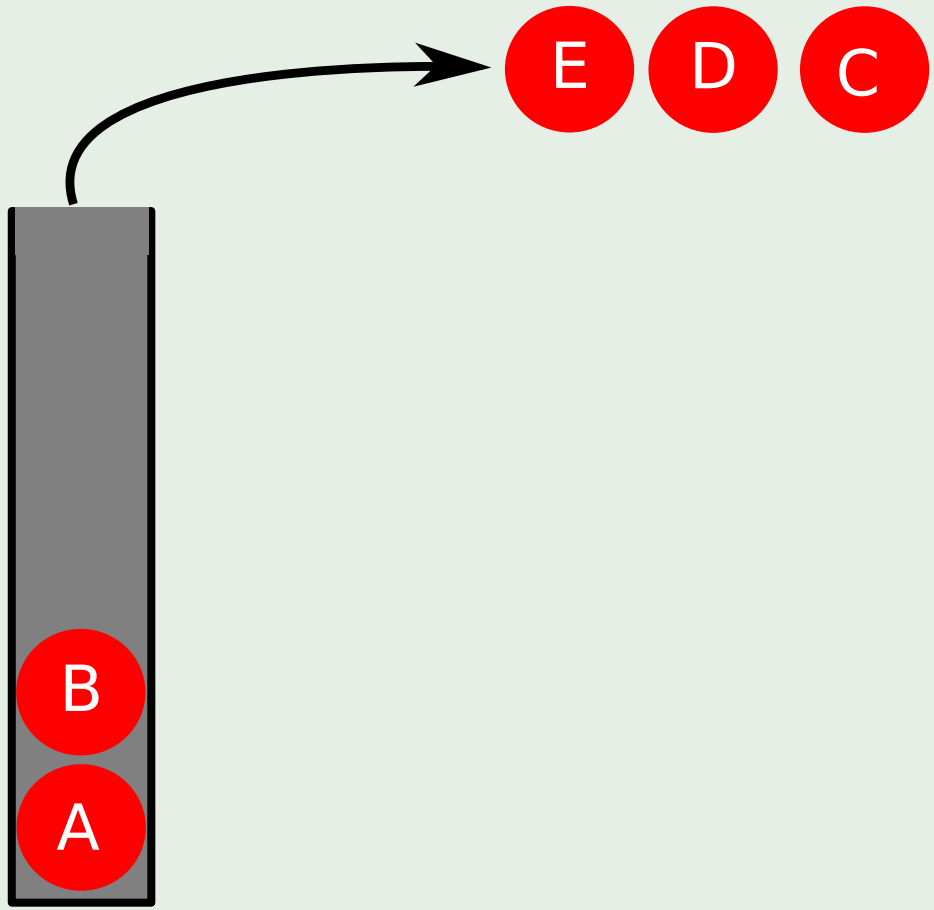
Example

List = {E,D}, Pop



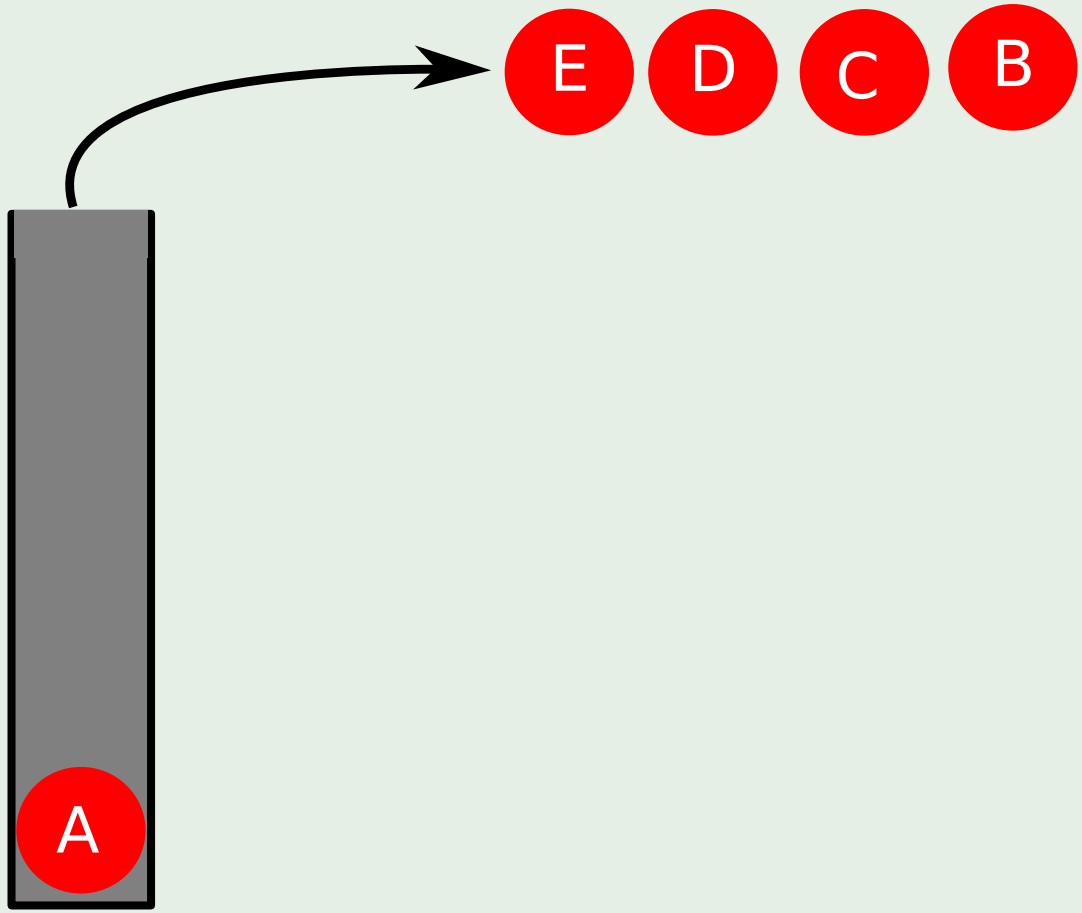
Example

List = {E,D,C}, Pop



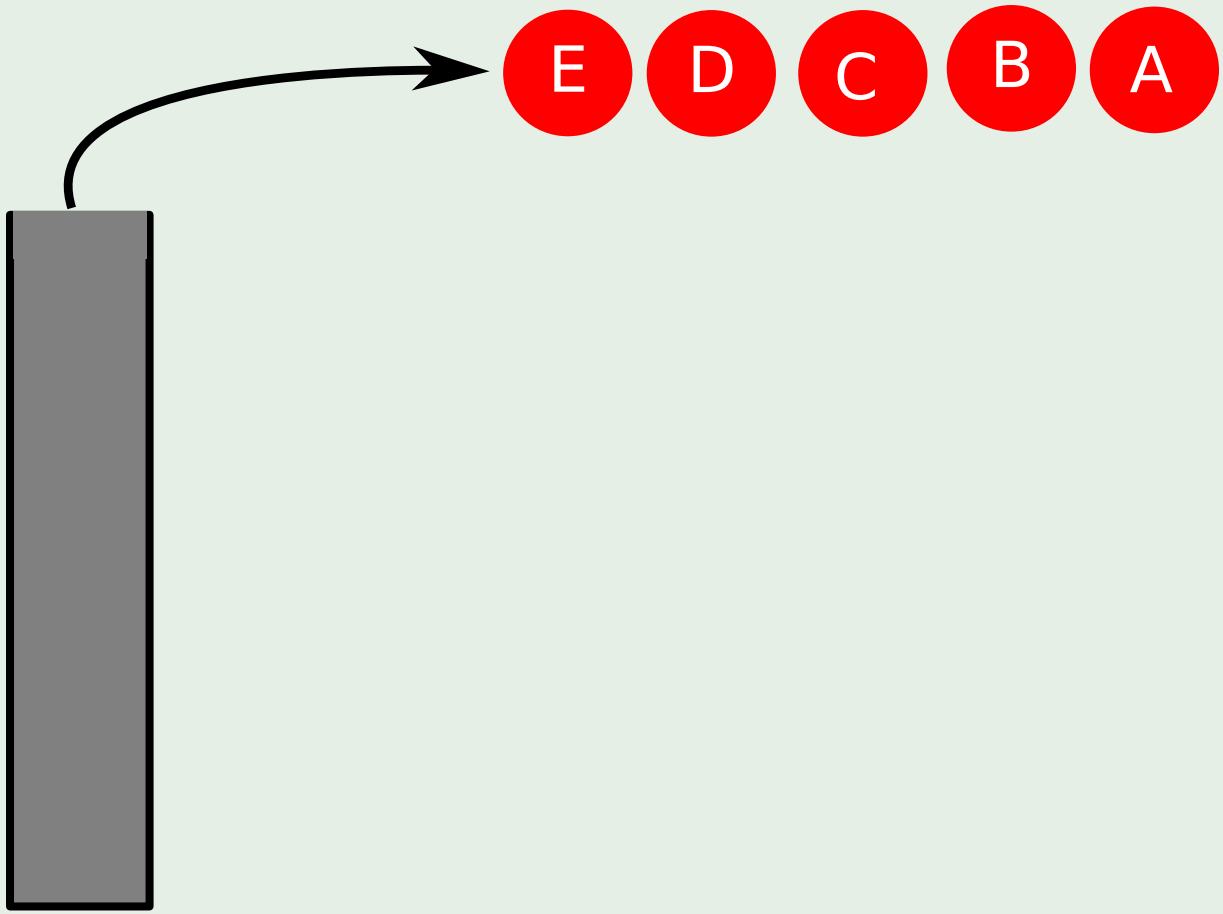
Example

List = {E,D,C,B}, Pop



Example

List = {E,D,C,B,A}



Outline

Stacks ADT

Interface

```
public interface Stack<Item>
{
    public boolean empty();
    public Item peek();
    public void push(Item TheObject);
    public Item pop();
}
```

Explanation of the ADT I

peek()

This method allows to look at the top of the stack without removing it!!!

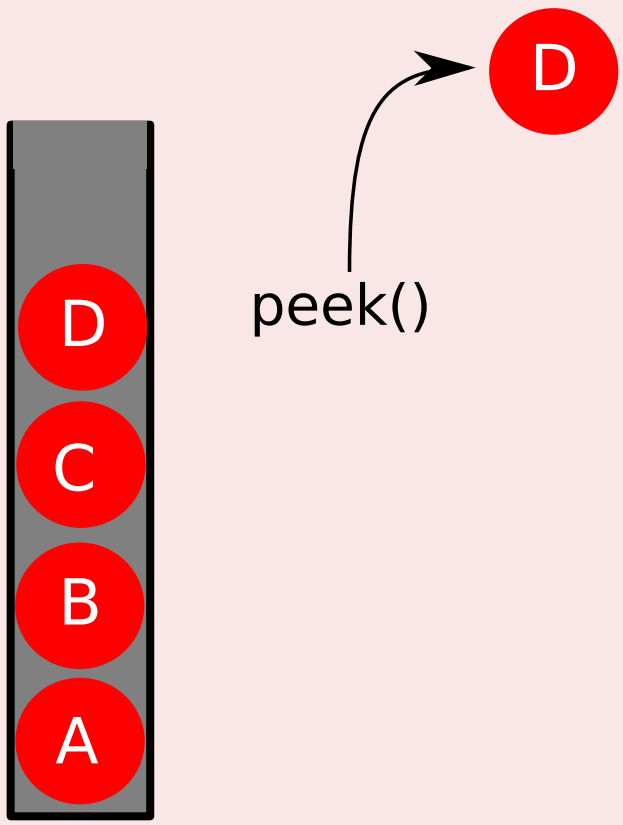
For Example

Explanation of the ADT I

peek()

This method allows to look at the top of the stack without removing it!!!

For Example



Explanation of the ADT II

pop()

This method allows to pop stuff from the top of the stack!!!

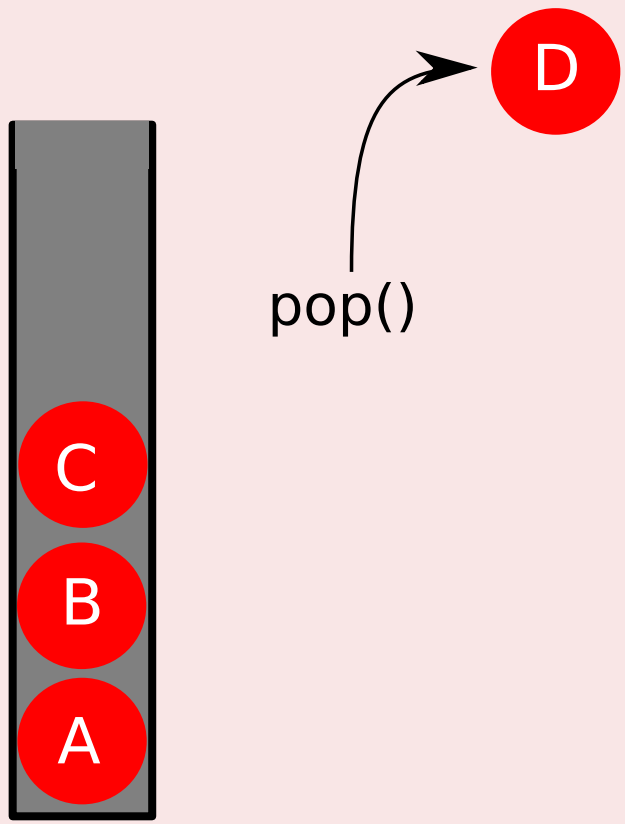
For Example

Explanation of the ADT II

pop()

This method allows to pop stuff from the top of the stack!!!

For Example



Explanation of the ADT III

push()

This method allows to push stuff to the top of the stack!!!

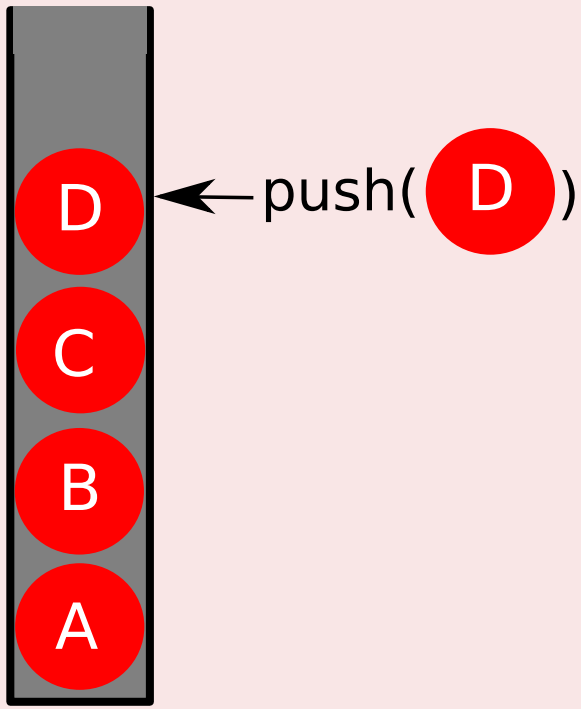
For Example

Explanation of the ADT III

push()

This method allows to push stuff to the top of the stack!!!

For Example



Explanation of the ADT III

empty()

This method allows to know if the stack is empty!!!

Stack Applications

Real life

- Pile of books
- Plate trays

Stack Applications

Real life

- Pile of books
- Plate trays

More applications related to computer science

- Program execution stack (You will know about this in OS or CA)
- Evaluating expressions

Stack Applications

Real life

- Pile of books
- Plate trays

More applications related to computer science

- Program execution stack (You will know about this in OS or CA)
- Evaluating expressions

Stack Applications

Real life

- Pile of books
- Plate trays

More applications related to computer science

- Program execution stack (You will know about this in OS or CA)
- Evaluating expressions

What do we do now?

First

Instead of going toward the implementations!!!

What do we do now?

First

Instead of going toward the implementations!!!

Why not examples?

1. Parentheses Matching
2. Towers Of Hanoi/Brahma
3. Switch Box Routing
4. Try-Throw-Catch in Java
5. Rat In A Maze

What do we do now?

First

Instead of going toward the implementations!!!

Why not examples?

1. Parentheses Matching
2. Towers Of Hanoi/Brahma
3. Switch Box Routing
4. Try-Throw-Catch in Java
5. Rat In A Maze

What do we do now?

First

Instead of going toward the implementations!!!

Why not examples?

1. Parentheses Matching
2. Towers Of Hanoi/Brahma
3. Switch Box Routing
4. Try-Throw-Catch in Java
5. Rat In A Maze

What do we do now?

First

Instead of going toward the implementations!!!

Why not examples?

1. Parentheses Matching
2. Towers Of Hanoi/Brahma
3. Switch Box Routing
4. Try-Throw-Catch in Java
5. Rat In A Maze

What do we do now?

First

Instead of going toward the implementations!!!

Why not examples?

1. Parentheses Matching
2. Towers Of Hanoi/Brahma
3. Switch Box Routing
4. Try-Throw-Catch in Java
5. Rat In A Maze

Outline

Parentheses Matching

Example - Input

(((a+b)*c+d-e)/(f+g)-(h+j))

$($	$($	$($	a	$+$	b	$)$	$*$	c	$+$	d	$-$	e	$)$	$/$	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$($	f	$+$	g	$)$	$-$	$($	h	$+$	j	$)$	$)$				
15	16	17	18	19	20	21	22	23	24	25	26				

Output

Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.

Or

(2,6) (1,13) (15,19) (21,25) (0,26)

Parentheses Matching

Example - Input

(((a+b)*c+d-e)/(f+g)-(h+j))

$($	$($	$($	a	$+$	b	$)$	$*$	c	$+$	d	$-$	e	$)$	$/$	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$($	f	$+$	g	$)$	$-$	$($	h	$+$	j	$)$	$)$				
15	16	17	18	19	20	21	22	23	24	25	26				

Output

Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.

Or

(2,6) (1,13) (15,19) (21,25) (0,26)

Parentheses Matching

Example - Input

(((a+b)*c+d-e)/(f+g)-(h+j))

$($	$($	$($	a	$+$	b	$)$	$*$	c	$+$	d	$-$	e	$)$	$/$	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$($	f	$+$	g	$)$	$-$	$($	h	$+$	j	$)$	$)$				
15	16	17	18	19	20	21	22	23	24	25	26				

Output

Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.

Or

(2,6) (1,13) (15,19) (21,25) (0,26)

Wrong Matching

Input

(a+b))*((c+d)

Wrong Matching

Input

(a+b))*((c+d)

Output

1. (0,4)
2. WRONG!!! Right parenthesis at 5 has no matching left parenthesis
3. (8,12)
4. WRONG!!! Left parenthesis at 7 has no matching right parenthesis

Wrong Matching

Input

(a+b))*((c+d)

Output

1. (0,4)
2. WRONG!!! Right parenthesis at 5 has no matching left parenthesis
3. (8,12)
4. WRONG!!! Left parenthesis at 7 has no matching right parenthesis

Wrong Matching

Input

(a+b))*((c+d)

Output

1. (0,4)
2. WRONG!!! Right parenthesis at 5 has no matching left parenthesis
3. (8,12)
4. WRONG!!! Left parenthesis at 7 has no matching right parenthesis

Wrong Matching

Input

(a+b))*((c+d)

Output

1. (0,4)
2. WRONG!!! Right parenthesis at 5 has no matching left parenthesis
3. (8,12)
4. WRONG!!! Left parenthesis at 7 has no matching right parenthesis

Wrong Matching

Input

(a+b))*((c+d)

Output

1. (0,4)
2. WRONG!!! Right parenthesis at 5 has no matching left parenthesis
3. (8,12)
4. WRONG!!! Left parenthesis at 7 has no matching right parenthesis

Developing a Recursive Solution I

First

What do we do? Ideas

Look at this

What if we have (a+b)?

Developing a Recursive Solution I

First

What do we do? Ideas

Look at this

What if we have (a+b)?

Initial Idea

boolean Rec-Paren(Chain List)

1. if (List.get(0)==’(’)
   1. List.remove(0)
   2. return Rec-Paren(List)

Initial Idea

boolean Rec-Paren(Chain List)

1. if (List.get(0)==')')
   1. List.remove(0)
   2. return Rec-Paren(List)

Now

What else?

Initial Idea

boolean Rec-Paren(Chain List)

1. if (List.get(0)==’(’)
   1. List.remove(0)
   2. return Rec-Paren(List)

Now

What else?

Initial Idea

boolean Rec-Paren(Chain List)

1. if (List.get(0)==’(’)
    1. List.remove(0)
    2. return Rec-Paren(List)

Now

What else?

Next Case

Cont...

2. else if (List.get(0)==’[0-9]|[+-]’)
    1. List.remove(0)
    2. return Rec-Paren(List)

Next Case

Cont...

2. else if (List.get(0)==’[0-9]|[+-]’)
   1. List.remove(0)
   2. return Rec-Paren(List)

Last Step

3. else if (List.get(0)==’)’)
   1. List.remove(0)
   2. return true
3. else return false

Next Case

Cont...

2. else if (List.get(0)==’[0-9]|[+-]’)
   1. List.remove(0)
   2. return Rec-Paren(List)

Last Step

3. else if (List.get(0)==’)’)
   1. List.remove(0)
   2. return true
3. else return false

Next Case

Cont...

2. else if (List.get(0)==’[0-9]|[+-]’)
   1. List.remove(0)
   2. return Rec-Paren(List)

Last Step

3. else if (List.get(0)==’)’)
   1. List.remove(0)
   2. return true
3. else return false

Next Case

Cont...

2. else if (List.get(0)==’[0-9]|[+-]’)
    1. List.remove(0)
    2. return Rec-Paren(List)

Last Step

3. else if (List.get(0)==’)’)
    1. List.remove(0)
    2. return true

3. else return false

Next Case

Cont...

2. else if (List.get(0)==’[0-9]|[+-]’)
    1. List.remove(0)
    2. return Rec-Paren(List)

Last Step

3. else if (List.get(0)==’)’)
    1. List.remove(0)
    2. return true
3. else return false

Next Case

Cont...

2. else if (List.get(0)==’[0-9]|[+-]’)
    1. List.remove(0)
    2. return Rec-Paren(List)

Last Step

3. else if (List.get(0)==’)’)
    1. List.remove(0)
    2. return true
3. else return false

Developing a Recursive Solution II

What if you have?

What if we have (a+b?

What about

What if we have a+b)?

Developing a Recursive Solution II

What if you have?

What if we have (a+b?

What about

What if we have a+b)?

This solution fails!!!

So, we need to send something down the recursion

What do we do? Ideas

What about...?

what if

We send down a flag!!

Thus

We need a flag to send down the recursion!!!

To tell the logic if we saw a left parenthesis

Ok

For simple problems fine!!! However...

Thus

We need a flag to send down the recursion!!!

To tell the logic if we saw a left parenthesis

Ok

For simple problems fine!!! However...

Then

For problems like these ones

((a+b)

Nope

It does not work

Then

For problems like these ones

((a+b)

Nope

It does not work

We need something more complex

A counter!!!

To see how many "(" we have seen down the recursion!!!

Recursive Solution - You assume a list of characters

```
public static boolean Balanced(ChainLinearList List,
                               int Counter){
        if (List.isEmpty()){ // Check empty
        if (Counter == 0)// Check Counter
                    return true;
        else
            return false;}
    if (List.get(0)=='(')// Case (
       {
          Counter++;
          List.remove(0);
          return Balanced(List,Counter);
       } else if (List.get(0)=='[0-9]|[+-]') // Case Number or -+
          {
            List.remove(0);
            return Balanced(List, Counter);
          } else if (List.get(0)==')')  // Case )
            {
               if (Counter > 0){
                   Counter--;
                   List.remove(0);
                   return Balanced(List, Counter)
               } else return false;
            } else return false
}
```

Can we simplify our code?

Yes

Using this memory container the STACK!!!

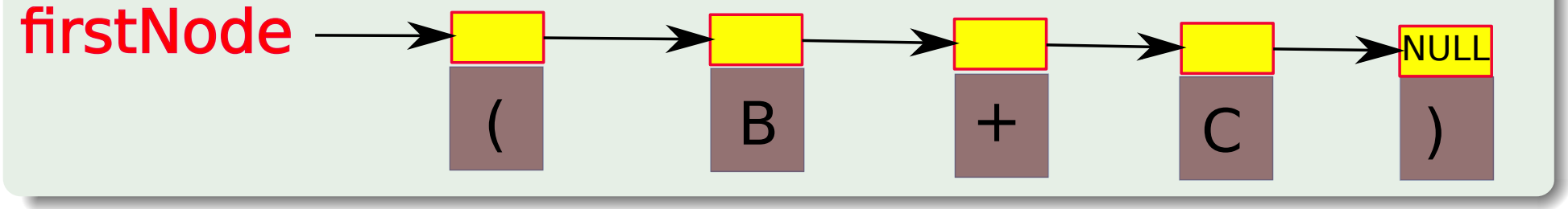
Before Anything Else

Did you notice something about my Chain during the recursion?

What?

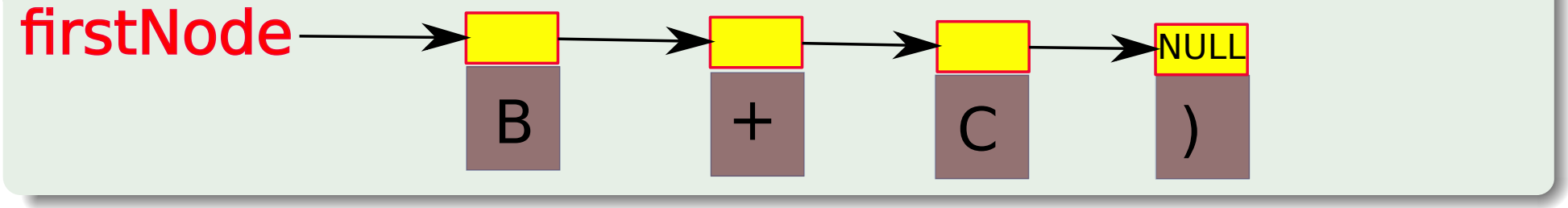
Yepi!!!

Removing Elements



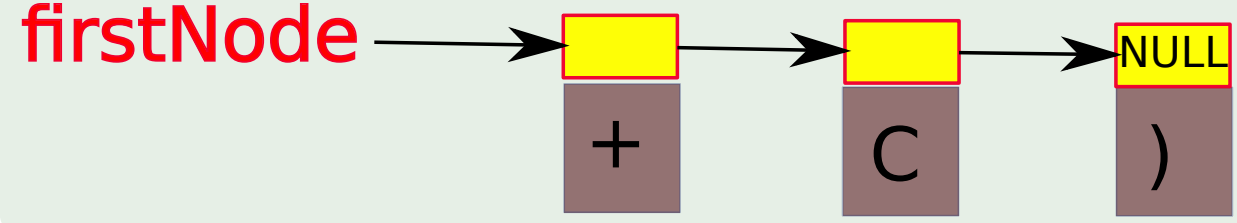
Yepi!!!

Removing Elements



Yepi!!!

Removing Elements



So, we can simulate the recursion using what?

First a way to...

ITERATE

A memory storage

A Stack

So somebody?

Can give a process for it?

So, we can simulate the recursion using what?

First a way to...

ITERATE

A memory storage

A Stack

So somebody?

Can give a process for it?

So, we can simulate the recursion using what?

First a way to...

ITERATE

A memory storage

A Stack

So somebody?

Can give a process for it?

Iterative Solution - You assume a list of characters

```
public static boolean Balanced(ChainLinearList List){
    Stack It = new Stack();
    if (List.isEmpty())
                return true
    while (!List.isEmpty()){ // Use a loop
          if (List.get(0)=='(')
             {
                 It.push(List.get(0));
                 List.remove(0);
             }
          else if (List.get(0)=='[0-9]|[+-]')
             { List.remove(0) }
          else if (List.get(0)==')')
             {
                 if (!It.empty())
                      It.pop();
                 else
                      return false;
                 List.remove(0); // Yes remove the
                                 // last ")"
      }
  if (It.empty())
           return true;
  return false
}
```

Outline

The Puzzle

The Leyend

- There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks.

Something Notable

- Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time.

So

According to the legend, when the last move of the puzzle will be completed, the world will end.

The Puzzle

The Leyend

- There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks.

Something Notable

- Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time.

So

According to the legend, when the last move of the puzzle will be completed, the world will end.

The Puzzle

The Leyend

- There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks.

Something Notable

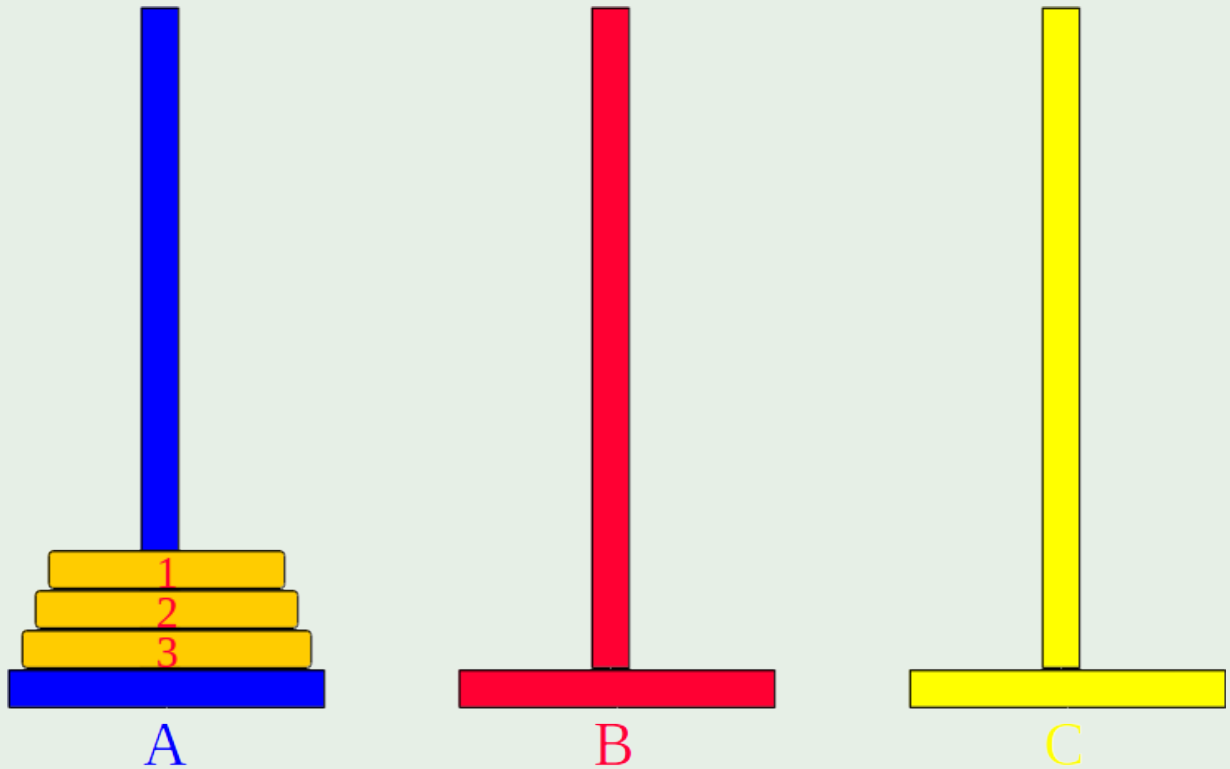
- Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time.

So

According to the legend, when the last move of the puzzle will be completed, the world will end.

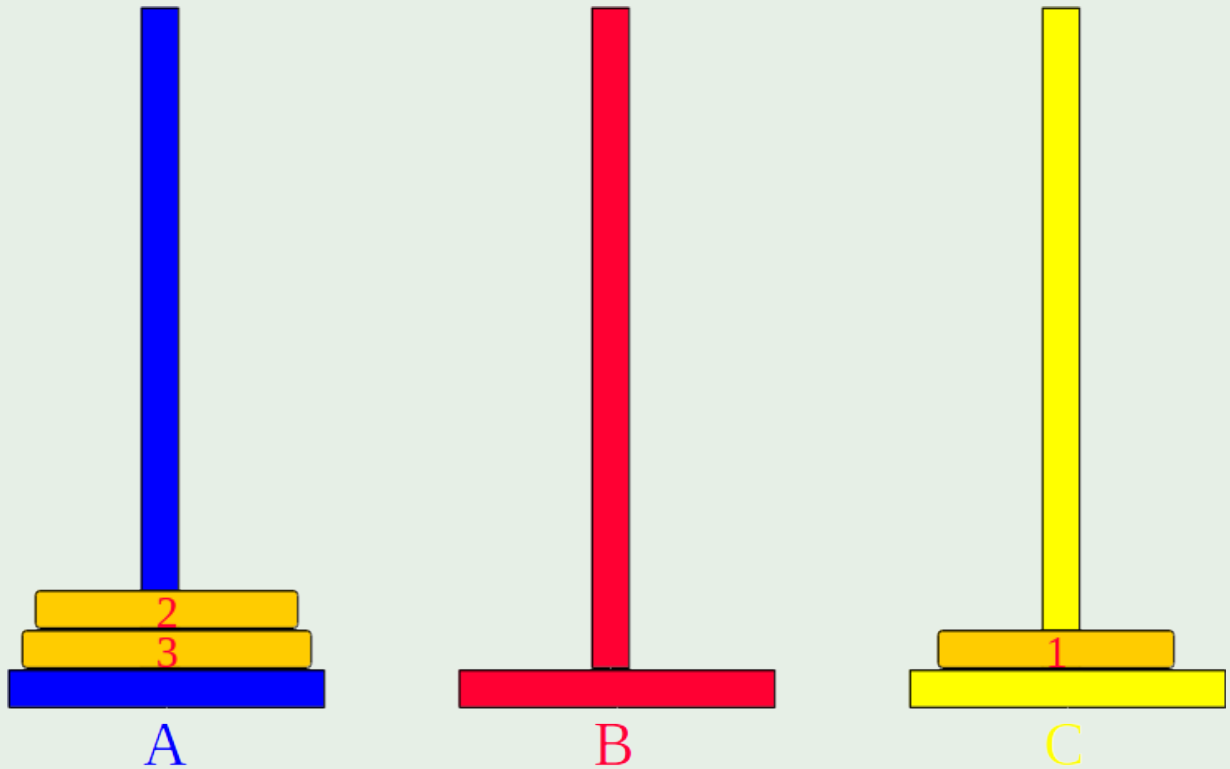
A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



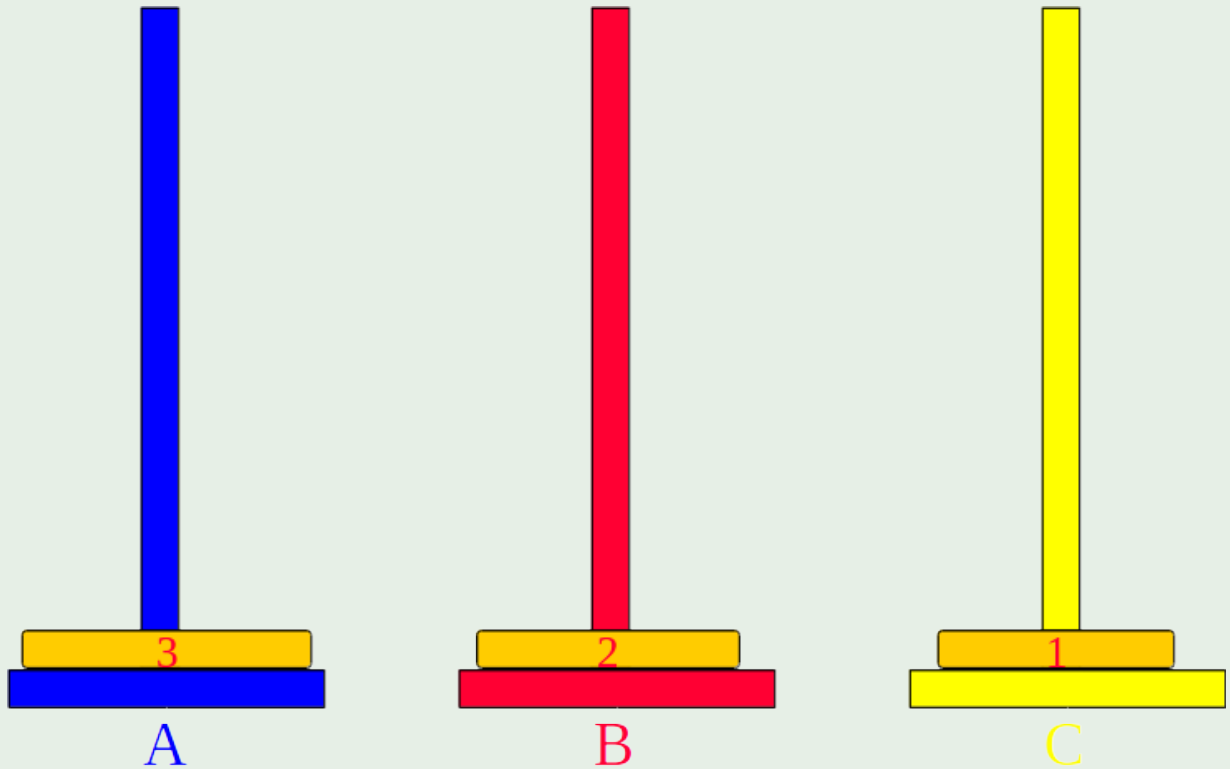
A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



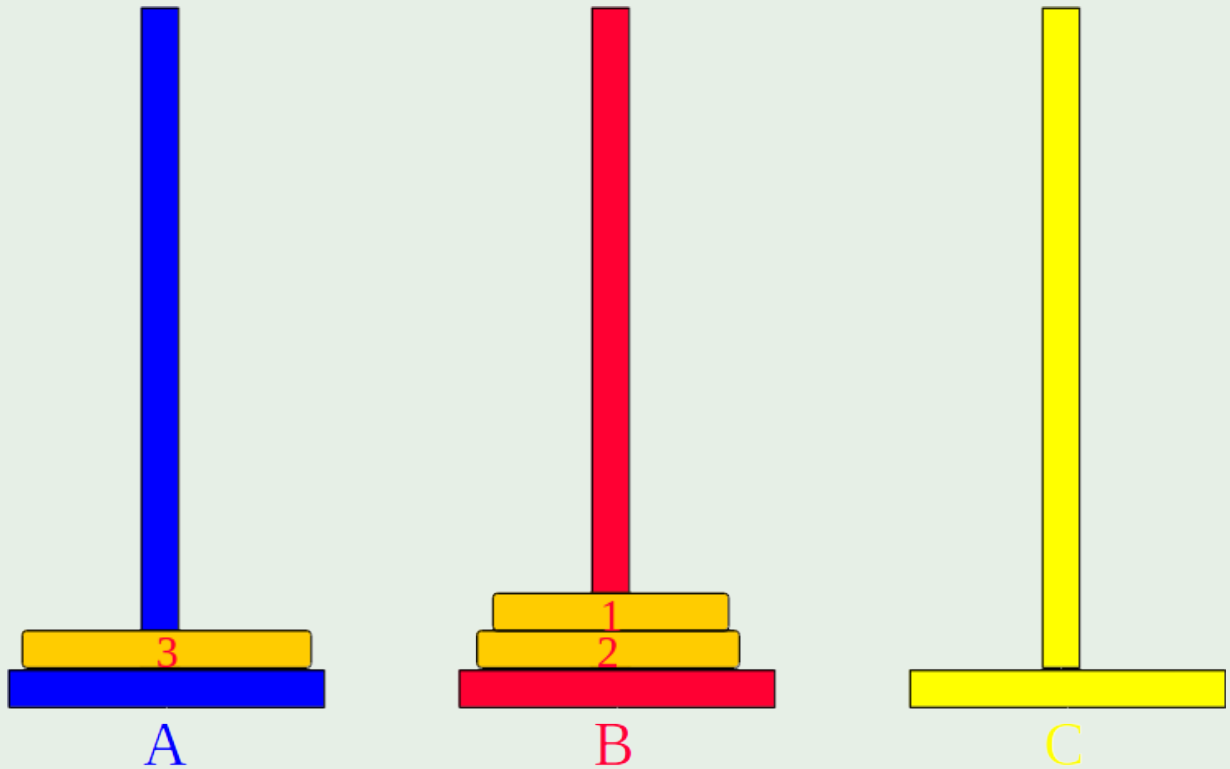
A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



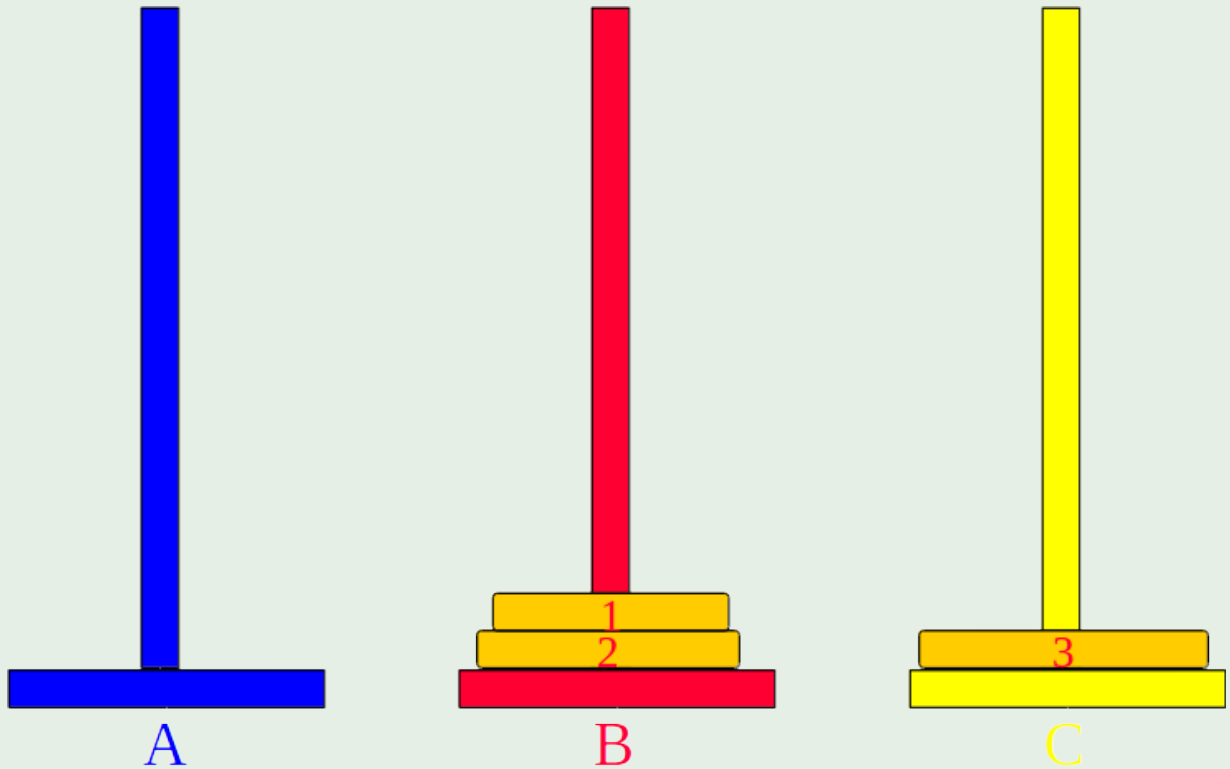
A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



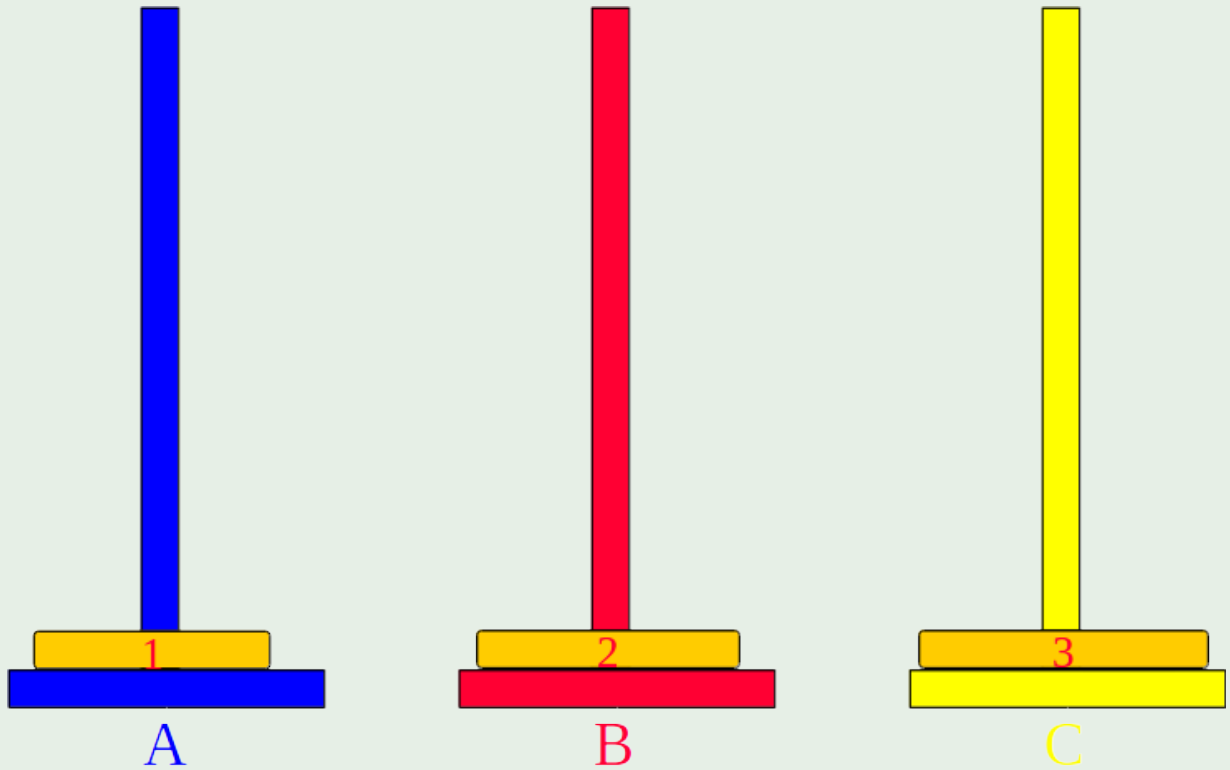
A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



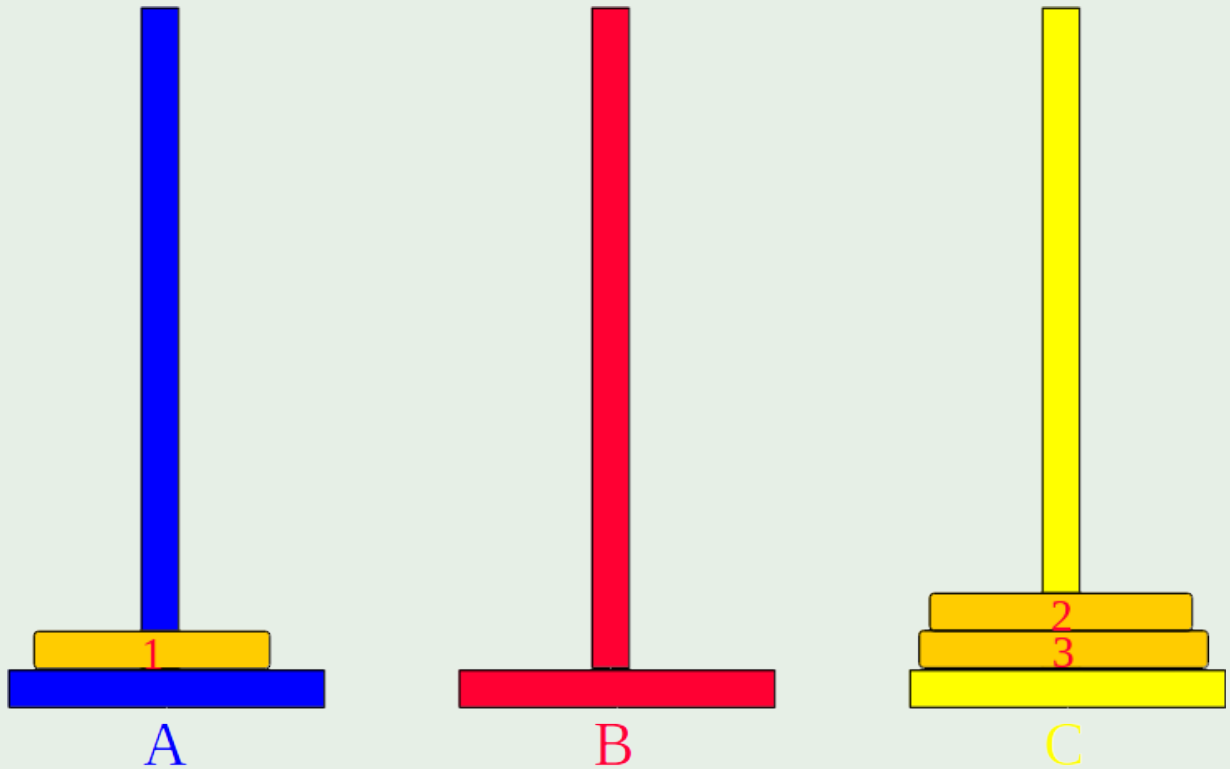
A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



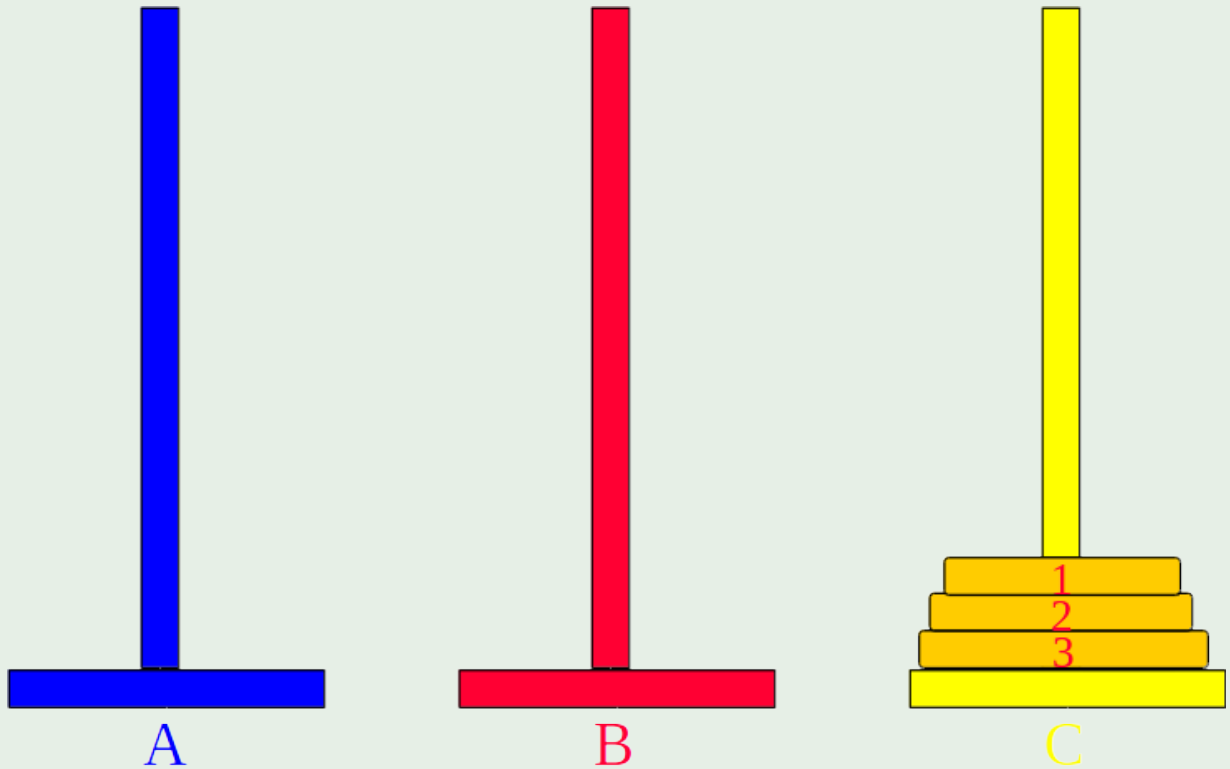
A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



The Rules

First One

Only one disk can be moved at a time.

Second One

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

Third One

No disk may be placed on top of a smaller disk.

The Rules

First One

Only one disk can be moved at a time.

Second One

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

Third One

No disk may be placed on top of a smaller disk.

The Rules

First One

Only one disk can be moved at a time.

Second One

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

Third One

No disk may be placed on top of a smaller disk.

Properties

With Three disk

You can solve it in seven moves

Thus, the minimum number of moves required for n disks

$$2^n - 1 \tag{1}$$

So how we solve the problem recursively?

Ideas?

Properties

With Three disk

You can solve it in seven moves

Thus, the minimum number of moves required for n disks

$$2^n - 1 \tag{1}$$

So how we solve the problem recursively?

Ideas?

Properties

With Three disk

You can solve it in seven moves

Thus, the minimum number of moves required for n disks

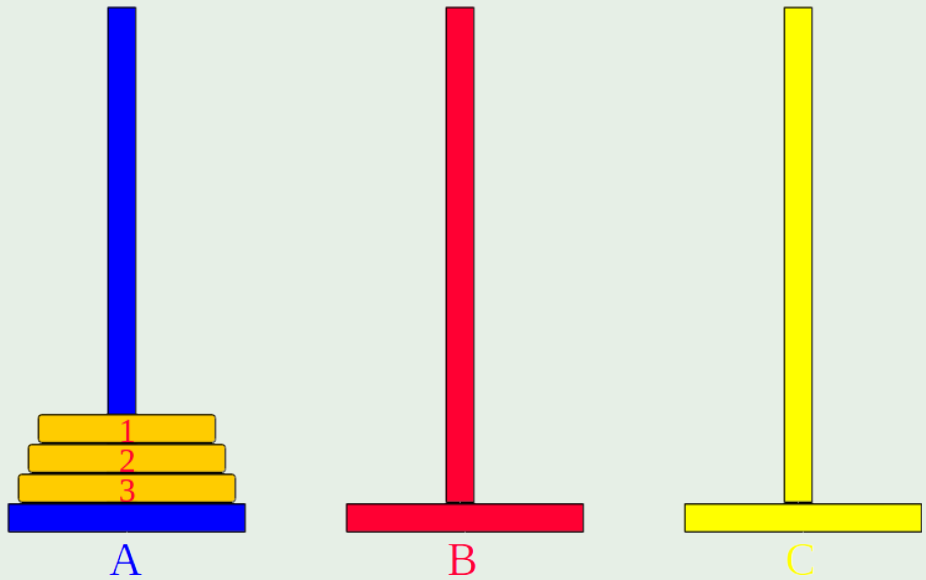
$$2^n - 1 \tag{1}$$

So how we solve the problem recursively?

Ideas?

Recursive Idea

Label the posts



Next

The other things

- Let n be the total number of discs.
- Number the discs from 1 (smallest, topmost) to n (largest, bottommost)

Next

The other things

- Let n be the total number of discs.
- Number the discs from 1 (smallest, topmost) to n (largest, bottommost)

How we solve this problem?

We can use the technique of "Divide and Conquer"

Next

The other things

- Let n be the total number of discs.
- Number the discs from 1 (smallest, topmost) to n (largest, bottommost)

How we solve this problem?

We can use the technique of "Divide and Conquer"

Divide and Conquer

Divide and Conquer

It is an important algorithm design paradigm based on multi-branched recursion.

Divide

- This phase of the algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type.

The Conquer I

- Then these subproblems become simple enough to be solved directly.

Divide and Conquer

Divide and Conquer

It is an important algorithm design paradigm based on multi-branched recursion.

Divide

- This phase of the algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type.

The Conquer I

- Then these subproblems become simple enough to be solved directly.

Divide and Conquer

Divide and Conquer

It is an important algorithm design paradigm based on multi-branched recursion.

Divide

- This phase of the algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type.

The Conquer I

- Then these subproblems become simple enough to be solved directly.

Divide and Conquer

The Conquer II

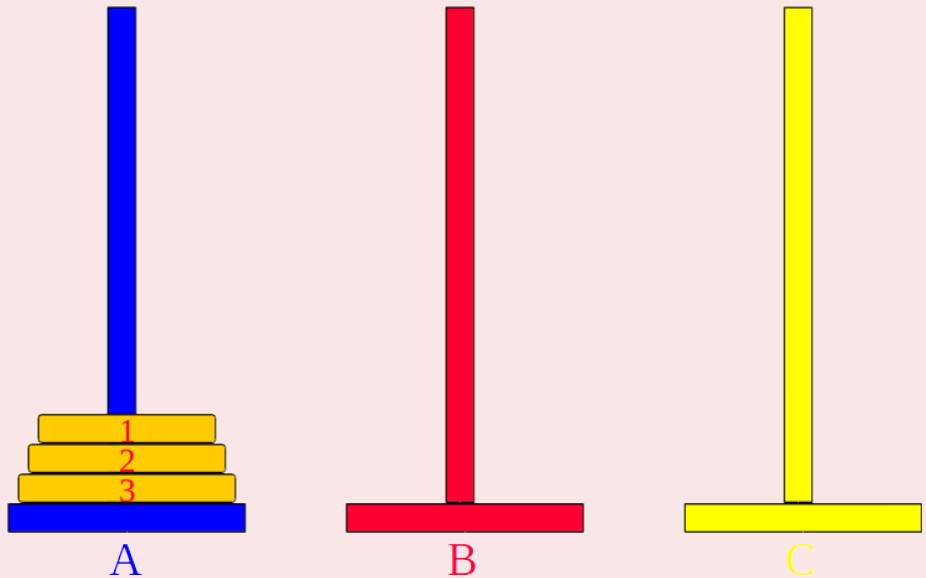
- The solutions to the sub-problems are then combined to give a solution to the original problem.

Question!!

First

What is the clever thing to do?

Example



A Simple Divide Phase

Move everything

The things that are in front of disk n.

- In the previous case $n = 3$

Ok

How the recursion looks as simple logic steps?

A Simple Divide Phase

Move everything

The things that are in front of disk n.

- In the previous case $n = 3$

Ok

How the recursion looks as simple logic steps?

The Initial Recursion

Logical Steps

To move n discs from tower A to tower C:

1. Move $n-1$ discs from A to B using C as a temporary!!!
2. This leaves disc n alone on tower A move disc n from A to C.
3. Move $n-1$ discs from B to C so they sit on disc n using A as temporary!!!

The Initial Recursion

Logical Steps

To move n discs from tower A to tower C:

1. Move $n-1$ discs from A to B using C as a temporary!!!
2. This leaves disc n alone on tower A move disc n from A to C.
3. Move $n-1$ discs from B to C so they sit on disc n using A as temporary!!!

The Initial Recursion

Logical Steps

To move n discs from tower A to tower C:

1. Move $n-1$ discs from A to B using C as a temporary!!!
2. This leaves disc n alone on tower A move disc n from A to C.
3. Move $n-1$ discs from B to C so they sit on disc n using A as temporary!!!

The Initial Recursion

Logical Steps

To move n discs from tower A to tower C:

1. Move $n-1$ discs from A to B using C as a temporary!!!
2. This leaves disc n alone on tower A move disc n from A to C.
3. Move $n-1$ discs from B to C so they sit on disc n using A as temporary!!!

Recursive Solution

Code

```
// Assume the list in A is in decreasing order
public static String TH(int n, Char Origin,
                        Char Destination,
                        Char Temp){
   String result;

   if (n == 1){
      return "Move␣Disk␣"+n+"␣from␣"+ Origin +
             "␣to␣" + Destination + "\n";
   }

    result = TH(n-1, Origin, Temp, Destination);
    result+= "Move␣Disk␣"+n+"␣from␣"+ Origin +
             "␣to␣" + Destination + "\n";
    result+= TH(n-1,Temp, Destination, Origin);

    return results;
}
```

What do we need for the iterative solution?

Storage for the disks

Use one stack per tower!!!

A while loop to simulate the recursion

Doing What?

What do we need for the iterative solution?

Storage for the disks

Use one stack per tower!!!

A while loop to simulate the recursion

Doing What?

Simple Logic to the Iterative Version

Logic

A simple solution for the toy puzzle:

1. Alternate moves between the smallest piece and a non-smallest piece.
2. When moving the smallest piece, always move it to the next position in the same direction.
3. If there is no tower position in the chosen direction, move the piece to the opposite end, but then continue to move in the correct direction.

Simple Logic to the Iterative Version

Logic

A simple solution for the toy puzzle:

1. Alternate moves between the smallest piece and a non-smallest piece.
2. When moving the smallest piece, always move it to the next position in the same direction.
3. If there is no tower position in the chosen direction, move the piece to the opposite end, but then continue to move in the correct direction.

Simple Logic to the Iterative Version

Logic

A simple solution for the toy puzzle:

1. Alternate moves between the smallest piece and a non-smallest piece.
2. When moving the smallest piece, always move it to the next position in the same direction.
3. If there is no tower position in the chosen direction, move the piece to the opposite end, but then continue to move in the correct direction.

Simple Logic to the Iterative Version

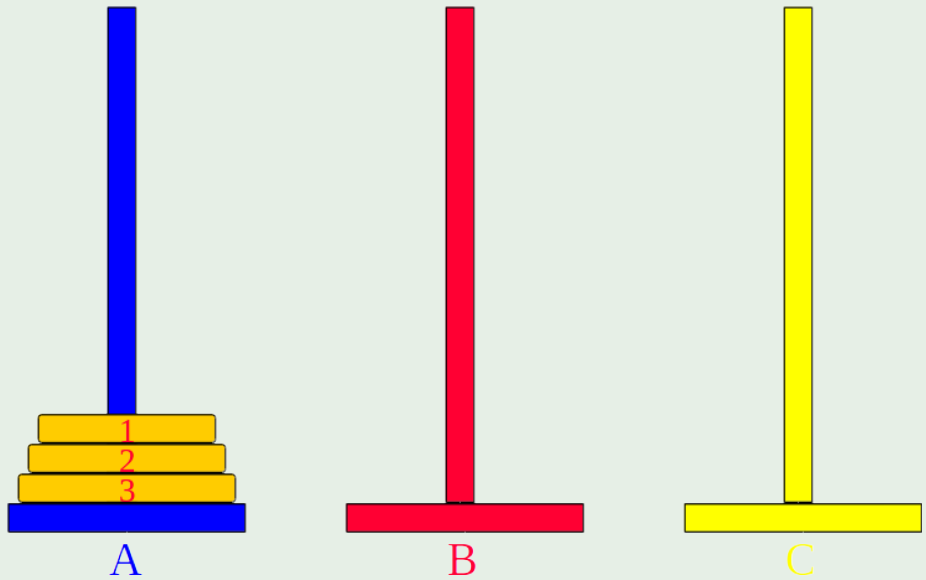
Logic

A simple solution for the toy puzzle:

1. Alternate moves between the smallest piece and a non-smallest piece.
2. When moving the smallest piece, always move it to the next position in the same direction.
3. If there is no tower position in the chosen direction, move the piece to the opposite end, but then continue to move in the correct direction.

A simple example with 3 disks

The objective of the puzzle is to move the entire stack to the end rod



We have two cases

When there are n disks

n is even

When there are n disk

n is odd

Then a really simple solution involving $2^{n-1} - 1$ moves

- You have three stacks: A, B, C.

We have two cases

When there are n disks

n is even

When there are n disk

n is odd

Then a really simple solution involving $2^{n-1} - 1$ moves

- You have three stacks: A, B, C.

We have two cases

When there are n disks

n is even

When there are n disk

n is odd

Then a really simple solution involving $2^{n-1} - 1$ moves

- You have three stacks: A, B, C.

We have for n even

Use Circular moves

$$A \Longrightarrow B \Longrightarrow C \Longrightarrow A \tag{2}$$

We have for n odd

Use Circular Moves

$$A \Longrightarrow C \Longrightarrow B \Longrightarrow A \tag{3}$$

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i = 1 to 2^{n-1}
   1. If (n%2 == 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i $= 1$ to 2^{n-1}
   1. If (n%2 $==$ 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i $= 1$ to 2^{n-1}
   1. If (n%2 $==$ 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i $= 1$ to 2^{n-1}
   1. If (n%2 $==$ 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i $= 1$ to 2^{n-1}
   1. If (n%2 $==$ 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i $= 1$ to 2^{n-1}
   1. If (n%2 $==$ 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i $= 1$ to 2^{n-1}
   1. If (n%2 $==$ 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i $= 1$ to 2^{n-1}
   1. If (n%2 $==$ 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

Process

IterativeHT using three stacks : towerA, towerB, towerC

1. for i $= 1$ to 2^{n-1}
   1. If (n%2 $==$ 0)
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for even order.
      3. Move the next largest disk to the only possible stack
   2. else
      1. Select the smallest disk at the top of the stacks
      2. Move the smallest disk one position in the direction of the cyclic order for odd order.
      3. Move the next largest disk to the only possible stack

What about the Most Efficient Version?

You need a node state

1. Number of Disks
2. Origin Tower
3. Destination Tower
4. Temporary Tower

What about the Most Efficient Version?

You need a node state

1. Number of Disks
2. Origin Tower
3. Destination Tower
4. Temporary Tower

What about the Most Efficient Version?

You need a node state

1. Number of Disks
2. Origin Tower
3. Destination Tower
4. Temporary Tower

What about the Most Efficient Version?

You need a node state

1. Number of Disks
2. Origin Tower
3. Destination Tower
4. Temporary Tower

We have then the following logic procedure

Iterative-Hanoi(n)

1. let S be an stack
2. let Result be a empty string
3. S.push(Node(n, A, C, B, Result))
4. while S is not empty
5. v = S.pop()
6. if (v.n == 1)
7. print "Move one from v.Origin to v.Destination \n"
8. else
9. S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10. S.push(Node(1, v.Origin, v.Destination, v.Temp))
11. S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))

We have then the following logic procedure

Iterative-Hanoi(n)

1. let S be an stack
2. let Result be a empty string
3. S.push(Node(n, A, C, B, Result))
4. while S is not empty
5. v = S.pop()
6. if (v.n == 1)
7. print "Move one from v.Origin to v.Destination \n"
8. else
9. S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10. S.push(Node(1, v.Origin, v.Destination, v.Temp))
11. S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))

We have then the following logic procedure

Iterative-Hanoi(n)

1. let S be an stack
2. let Result be a empty string
3. S.push(Node(n, A, C, B, Result))
4. while S is not empty
5. v = S.pop()
6. if (v.n == 1)
7. print "Move one from v.Origin to v.Destination \n"
8. else
9. S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10. S.push(Node(1, v.Origin, v.Destination, v.Temp))
11. S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))

We have then the following logic procedure

Iterative-Hanoi(n)

1. let S be an stack
2. let Result be a empty string
3. S.push(Node(n, A, C, B, Result))
4. while S is not empty
5. v = S.pop()
6. if (v.n == 1)
7. print "Move one from v.Origin to v.Destination \n"
8. else
9. S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10. S.push(Node(1, v.Origin, v.Destination, v.Temp))
11. S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))

We have then the following logic procedure

Iterative-Hanoi(n)

1. let S be an stack
2. let Result be a empty string
3. S.push(Node(n, A, C, B, Result))
4. while S is not empty
5. v = S.pop()
6. if (v.n == 1)
7. print "Move one from v.Origin to v.Destination \n"
8. else
9. S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10. S.push(Node(1, v.Origin, v.Destination, v.Temp))
11. S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))

We have then the following logic procedure

Iterative-Hanoi(n)

1. let S be an stack
2. let Result be a empty string
3. S.push(Node(n, A, C, B, Result))
4. while S is not empty
5. v = S.pop()
6. if (v.n == 1)
7. print "Move one from v.Origin to v.Destination \n"
8. else
9. S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10. S.push(Node(1, v.Origin, v.Destination, v.Temp))
11. S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))

We have then the following logic procedure

Iterative-Hanoi(n)

1. let S be an stack
2. let Result be a empty string
3. S.push(Node(n, A, C, B, Result))
4. while S is not empty
5. v = S.pop()
6. if (v.n == 1)
7. print "Move one from v.Origin to v.Destination \n"
8. else
9. S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10. S.push(Node(1, v.Origin, v.Destination, v.Temp))
11. S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))

We have then the following logic procedure

Iterative-Hanoi(n)

1. let S be an stack
2. let Result be a empty string
3. S.push(Node(n, A, C, B, Result))
4. while S is not empty
5. v = S.pop()
6. if (v.n == 1)
7. print "Move one from v.Origin to v.Destination \n"
8. else
9. S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10. S.push(Node(1, v.Origin, v.Destination, v.Temp))
11. S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))

We have then the following logic procedure

Iterative-Hanoi(n)

```
1  let S be an stack
2  let Result be a empty string
3  S.push(Node(n, A, C, B, Result))
4  while S is not empty
5        v = S.pop()
6        if (v.n == 1)
7              print "Move one from v.Origin to v.Destination \n"
8        else
9              S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10             S.push(Node(1, v.Origin, v.Destination, v.Temp))
11             S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))
```

We have then the following logic procedure

Iterative-Hanoi(n)

```
1  let S be an stack
2  let Result be a empty string
3  S.push(Node(n, A, C, B, Result))
4  while S is not empty
5        v = S.pop()
6        if (v.n == 1)
7              print "Move one from v.Origin to v.Destination \n"
8        else
9              S.push(Node(v.n-1, v.Temp, v.Destination, v.Origin))
10             S.push(Node(1, v.Origin, v.Destination, v.Temp))
11             S.push(Node(v.n-1,v.Origin, v.Temp, v.Destination))
```

What about the Legend?

To solve the towers of Hanoi for 64 disks

We need $\approx 1.8 * 10^{19}$ moves

With a computer making 10^9 moves/second

A computer would take about 570 years to complete.

- The sun will destroy the life on earth in 2 billion years.

What about the Legend?

To solve the towers of Hanoi for 64 disks

We need $\approx 1.8 * 10^{19}$ moves

With a computer making 10^9 moves/second

A computer would take about 570 years to complete.

At 1 disk move/min

- The monks will take about 3.4×10^{13} years.
- The sun will destroy the life on earth in 2 billion years.

What about the Legend?

To solve the towers of Hanoi for 64 disks

We need $\approx 1.8 * 10^{19}$ moves

With a computer making 10^9 moves/second

A computer would take about 570 years to complete.

At 1 disk move/min

- The monks will take about 3.4×10^{13} years.
- The sun will destroy the life on earth in 2.8×10^9.

Outline

Chess

As in the Tower of Hanoi

It is possible to devise a massive search solution based in the actual state of the chess board.

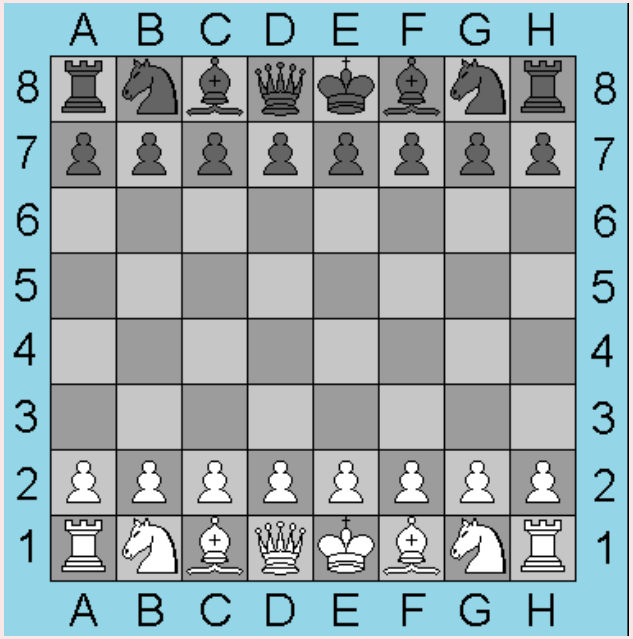
State 0

Chess

As in the Tower of Hanoi

It is possible to devise a massive search solution based in the actual state of the chess board.

State 0



Then

Something Notable

- If you put 1 penny for the first square, 2 for next, 4 for next, 8 for next, and so on.

You have

- $\$3.6 * 10^{17}$ (federal budget $\sim 2 * 10^{12}$) .

Then

Something Notable

- If you put 1 penny for the first square, 2 for next, 4 for next, 8 for next, and so on.

You have

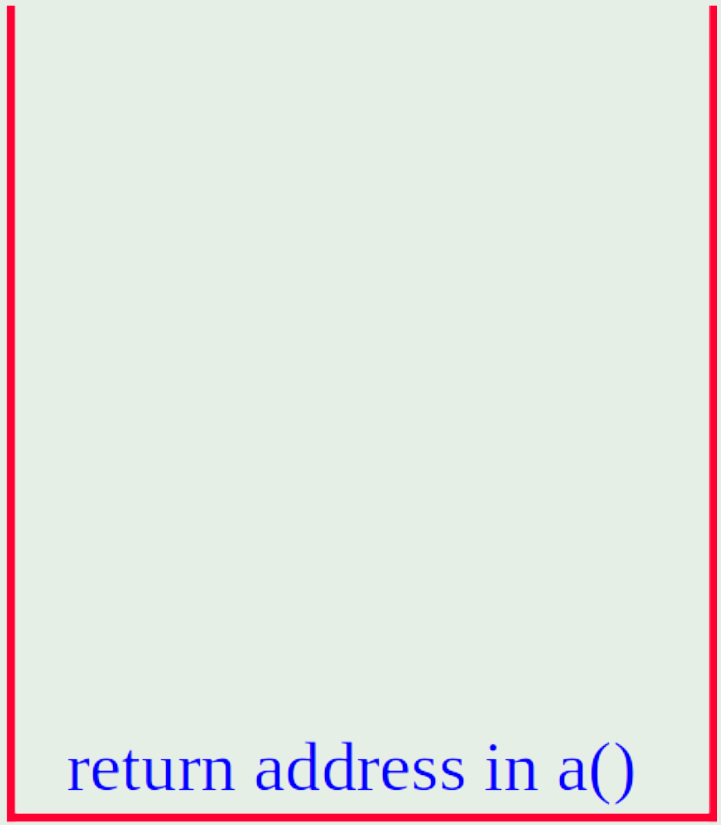
- $\$3.6 * 10^{17}$ (federal budget $\sim 2 * 10^{12}$) .

Outline

Method Invocation And Return

We have stack in the memory system

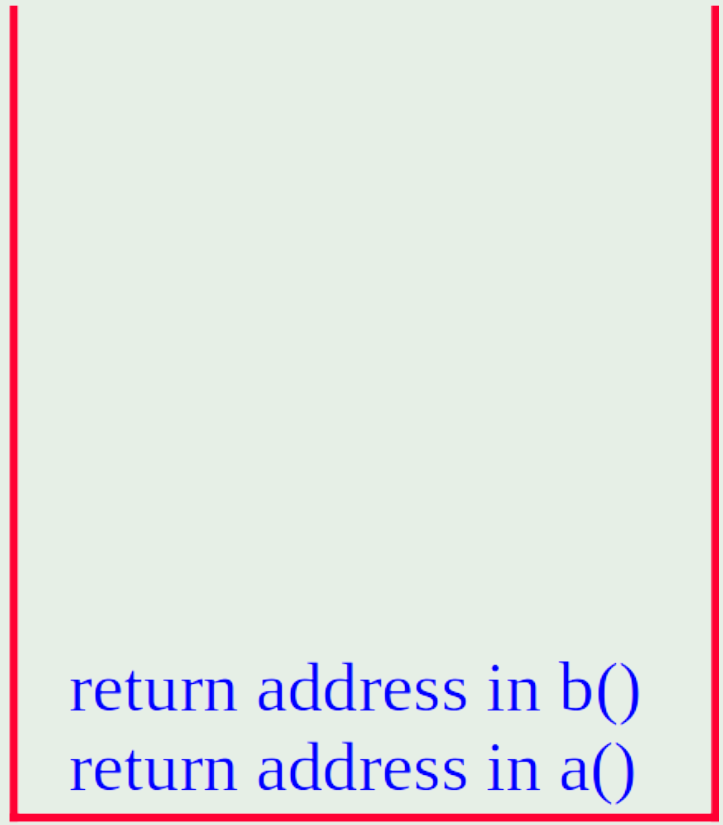
```
public void a()
{ ...; b(); ...}
```



Method Invocation And Return

We have stack in the memory system

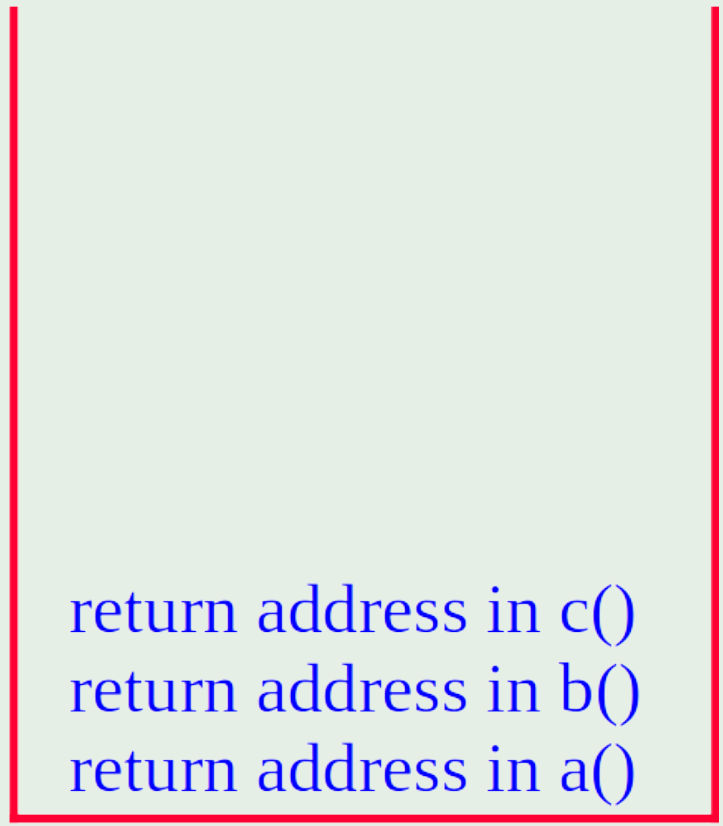
```
public void a()
{ …; b(); …}
public void b()
{ …; c(); …}
```



Method Invocation And Return

We have stack in the memory system

```
public void a()
{ …; b(); …}
public void b()
{ …; c(); …}
public void c()
{ …; d(); …}
```



Outline

Try-Throw-Catch

Thus

1. When you enter a try block, push the address of this block on a stack.
2. When an exception is thrown, pop the try block that is at the top of the stack (if the stack is empty, terminate).
3. If the popped try block has no matching catch block, go back to the preceding step.
4. If the popped try block has a matching catch block, execute the matching catch block.

Try-Throw-Catch

Thus

1. When you enter a try block, push the address of this block on a stack.
2. When an exception is thrown, pop the try block that is at the top of the stack (if the stack is empty, terminate).
3. If the popped try block has no matching catch block, go back to the preceding step.
4. If the popped try block has a matching catch block, execute the matching catch block.

Try-Throw-Catch

Thus

1. When you enter a try block, push the address of this block on a stack.
2. When an exception is thrown, pop the try block that is at the top of the stack (if the stack is empty, terminate).
3. If the popped try block has no matching catch block, go back to the preceding step.
4. If the popped try block has a matching catch block, execute the matching catch block.

Try-Throw-Catch

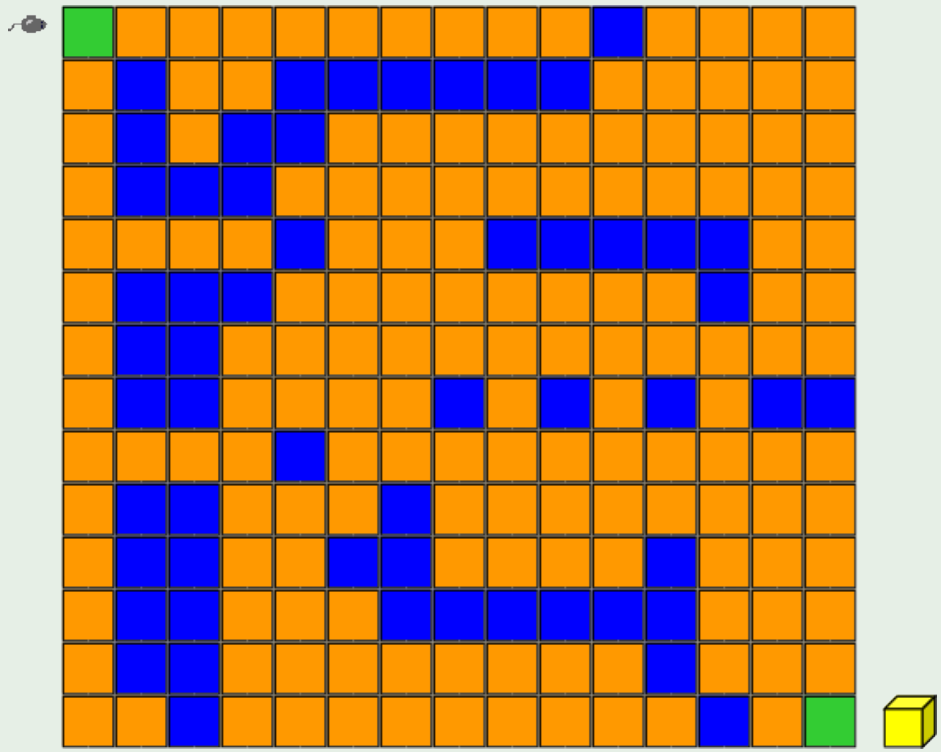
Thus

1. When you enter a try block, push the address of this block on a stack.
2. When an exception is thrown, pop the try block that is at the top of the stack (if the stack is empty, terminate).
3. If the popped try block has no matching catch block, go back to the preceding step.
4. If the popped try block has a matching catch block, execute the matching catch block.

Outline

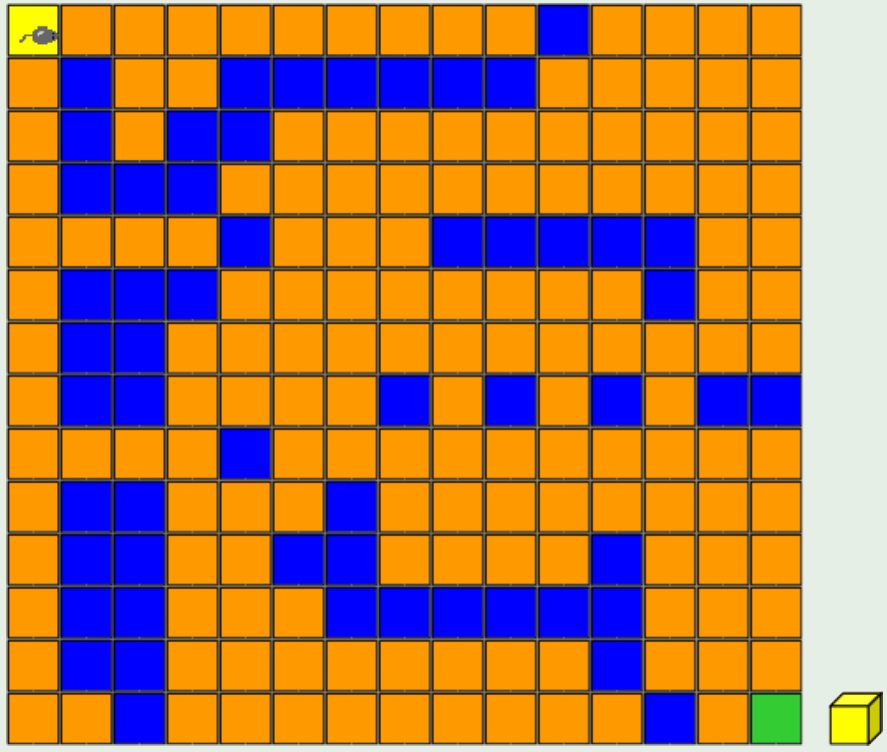
Rat In A Maze

Example



Rat In A Maze

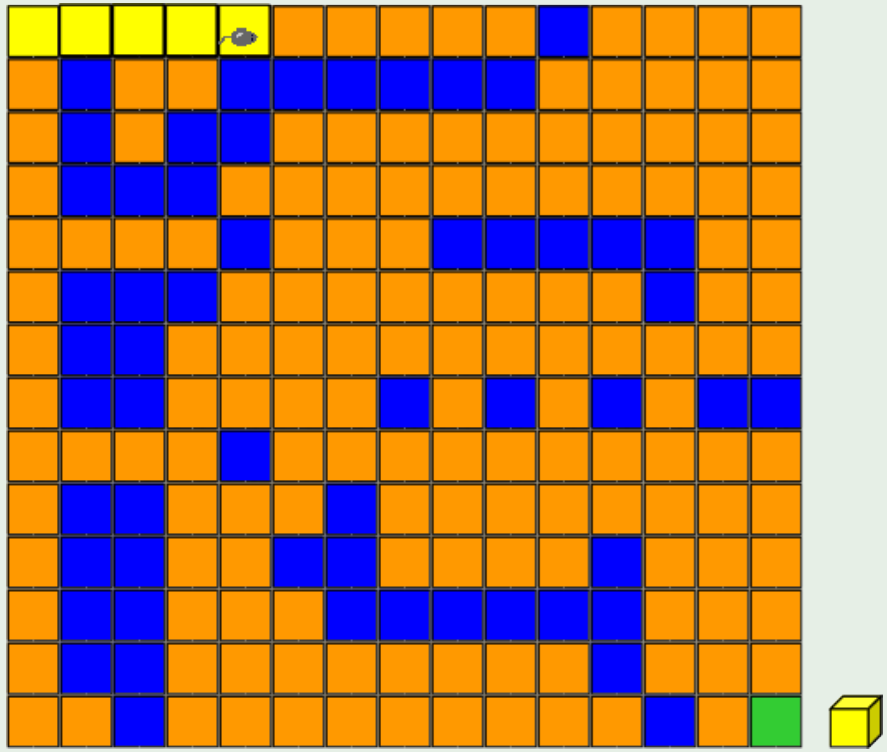
Example



- Move order is: right, down, left, up
- Block positions to avoid revisit.

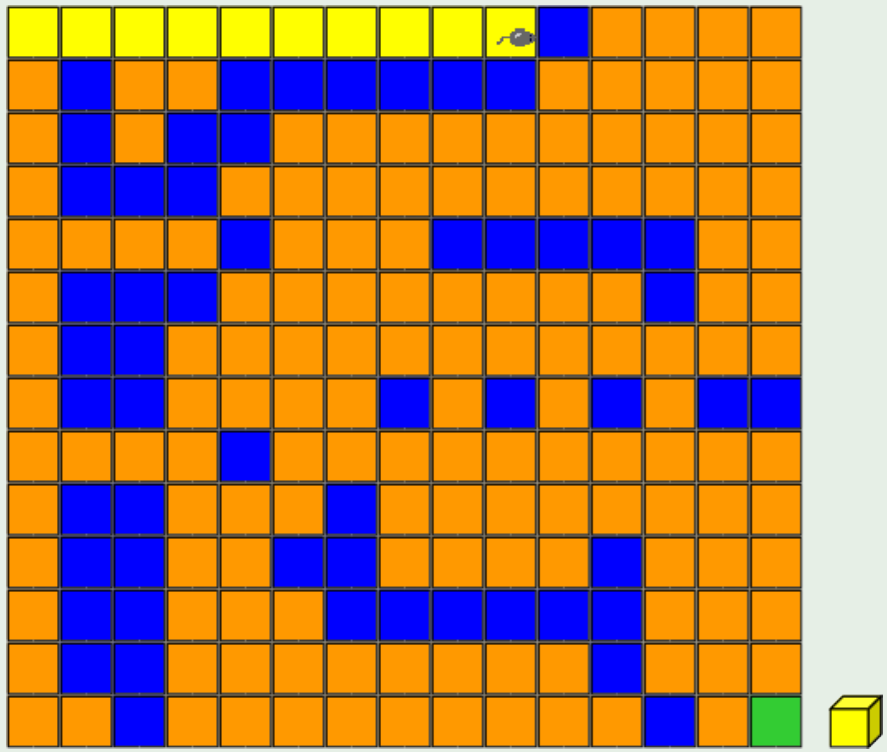
Rat In A Maze

Example



Rat In A Maze

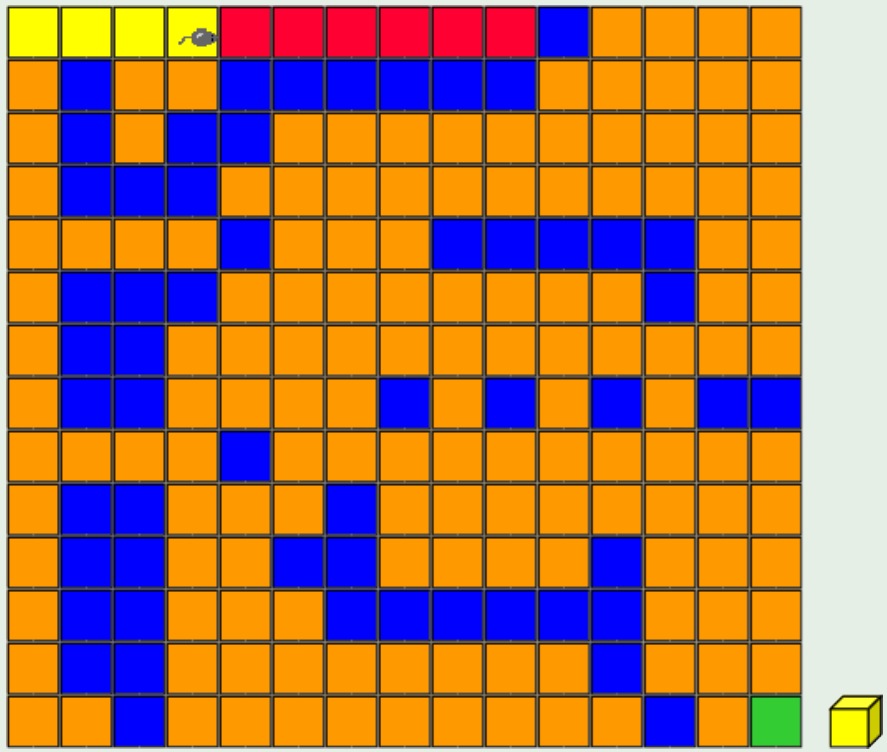
Example



- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze

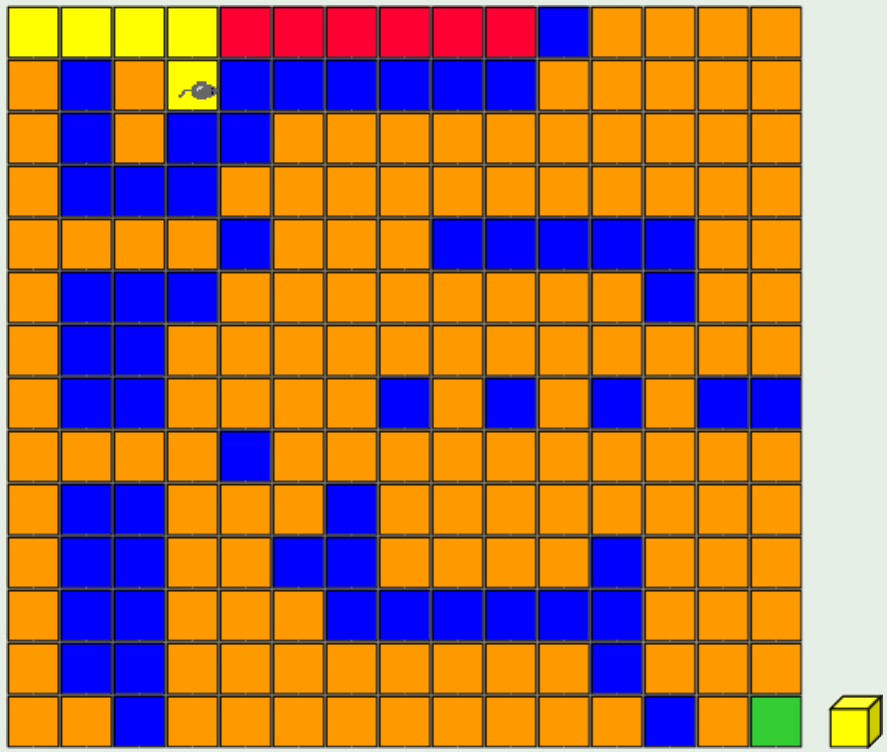
Example



- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze

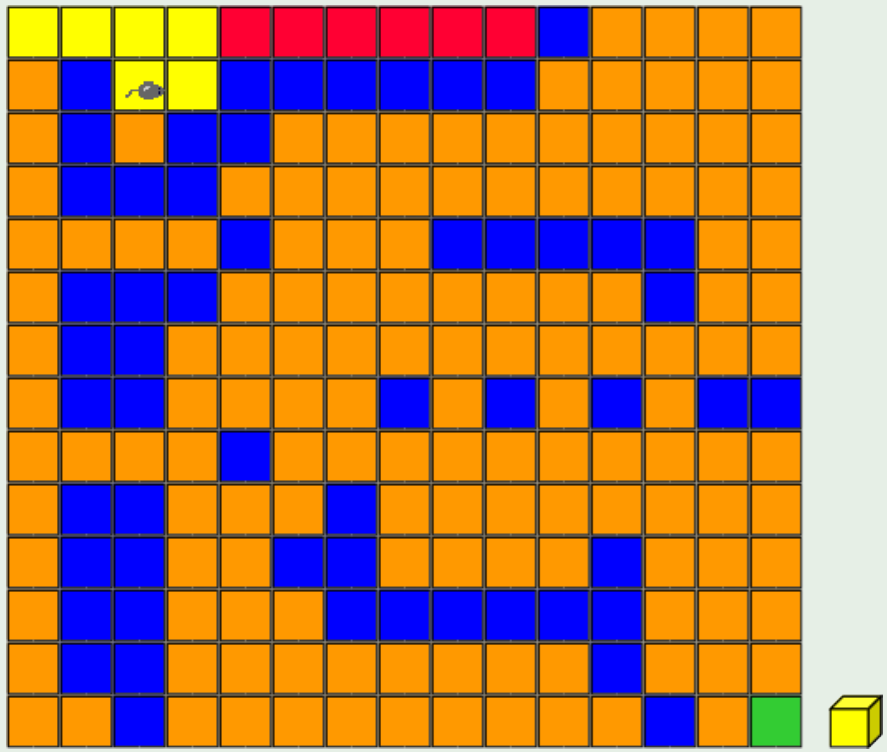
Example



- MOVE DOWN

Rat In A Maze

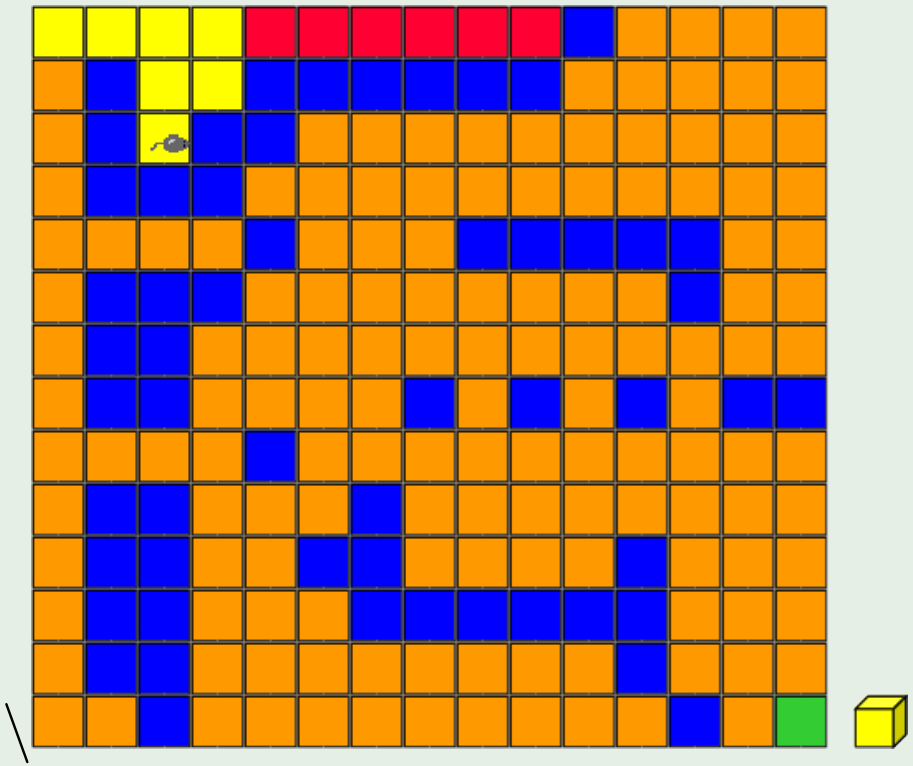
Example



- MOVE LEFT

Rat In A Maze

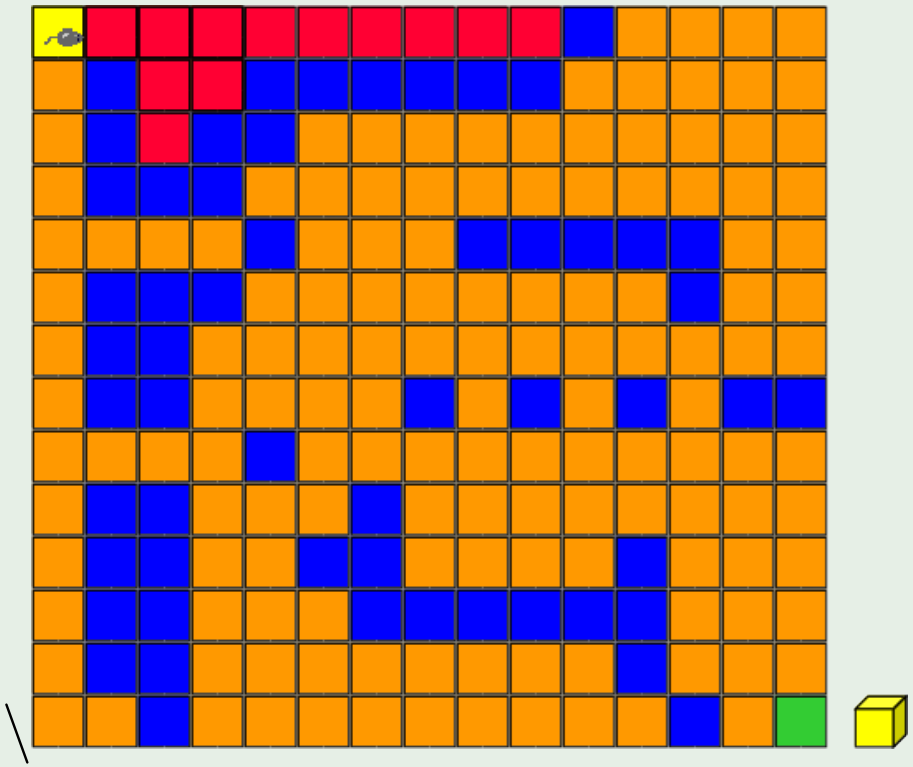
Example



- MOVE DOWN

Rat In A Maze

Example



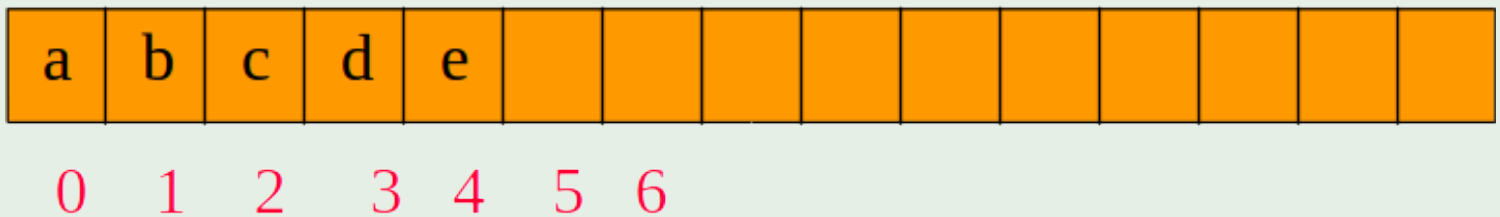
- Move backward until we reach a square from which a forward move is possible.

Outline

Derivation From ArrayLinearList

We can do the following

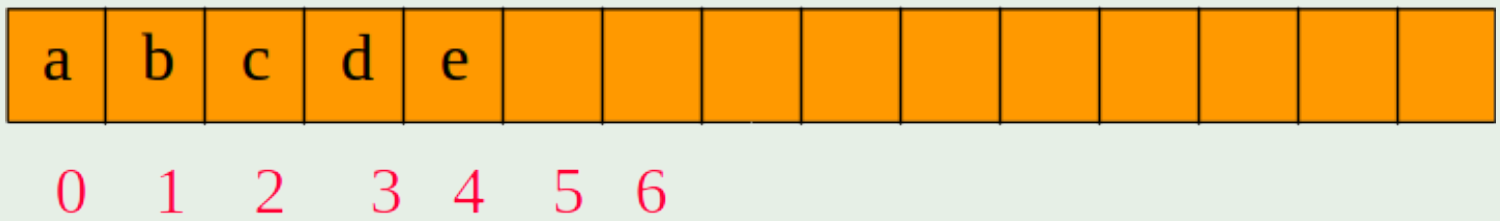
Stack top is either left end or right end of linear list



Derivation From ArrayLinearList

We can do the following

Stack top is either left end or right end of linear list



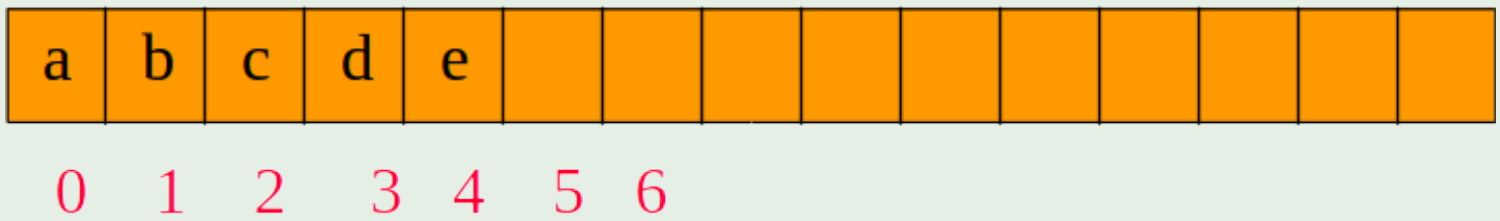
For any possible top

- empty() $\Longrightarrow$ isEmpty()
  - O(1) time
- peek() $\Longrightarrow$ get(0) or get(size() - 1)
  - O(1) time

Derivation From ArrayLinearList

We can do the following

Stack top is either left end or right end of linear list



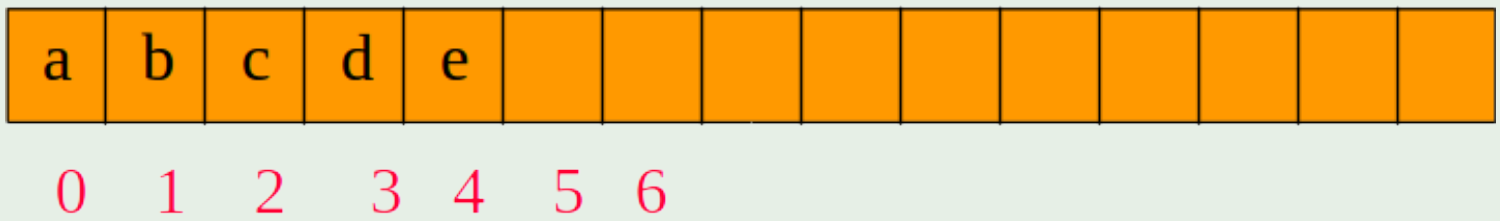
For any possible top

- empty() $\Longrightarrow$ isEmpty()
  - O(1) time
- peek() $\Longrightarrow$ get(0) or get(size() - 1)
  - O(1) time

Derivation From ArrayLinearList

We can do the following

Stack top is either left end or right end of linear list



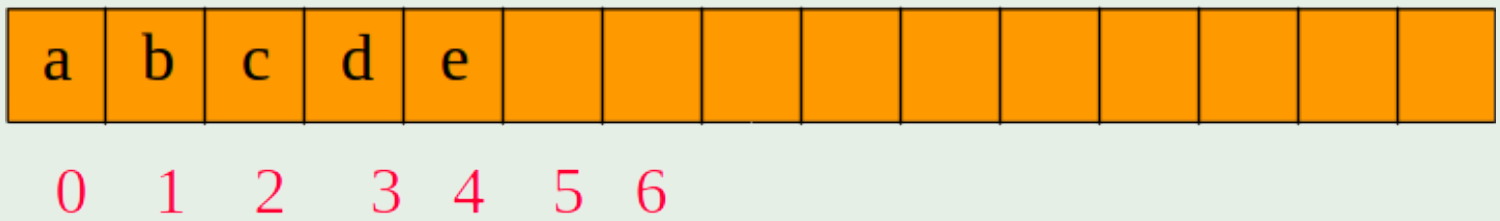
For any possible top

- empty() $\Longrightarrow$ isEmpty()
  - O(1) time
- peek() $\Longrightarrow$ get(0) or get(size() - 1)
  - O(1) time

Derivation From ArrayLinearList

We can do the following

Stack top is either left end or right end of linear list



For any possible top

- empty() $\Longrightarrow$ isEmpty()
  - O(1) time
- peek() $\Longrightarrow$ get(0) or get(size() - 1)
  - O(1) time

Derivation From ArrayLinearList

When top is left end of linear list

- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(0)
  - O(size) time

Derivation From ArrayLinearList

When top is left end of linear list

- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(0)
  - O(size) time

Derivation From ArrayLinearList

When top is left end of linear list

- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(0)
  - O(size) time

Derivation From ArrayLinearList

When top is left end of linear list

- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(0)
  - O(size) time

Derivation From ArrayLinearList

When top is right end of linear list

For any possible top

- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(1) time

Derivation From ArrayLinearList

When top is right end of linear list

For any possible top

- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(1) time

Derivation From ArrayLinearList

When top is right end of linear list

For any possible top

- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(1) time

Derivation From ArrayLinearList

When top is right end of linear list

For any possible top

- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(1) time

Derivation From ArrayLinearList

When top is right end of linear list

For any possible top

- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(1) time

Thus

The Moral of the Story

You must use the right end of list as top of stack

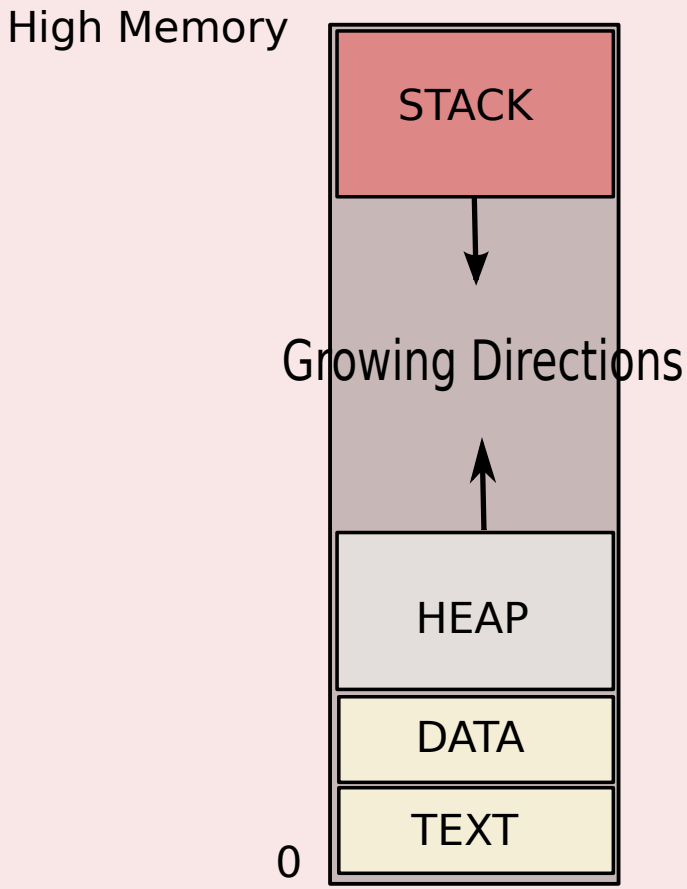
That is way

Thus

The Moral of the Story

You must use the right end of list as top of stack

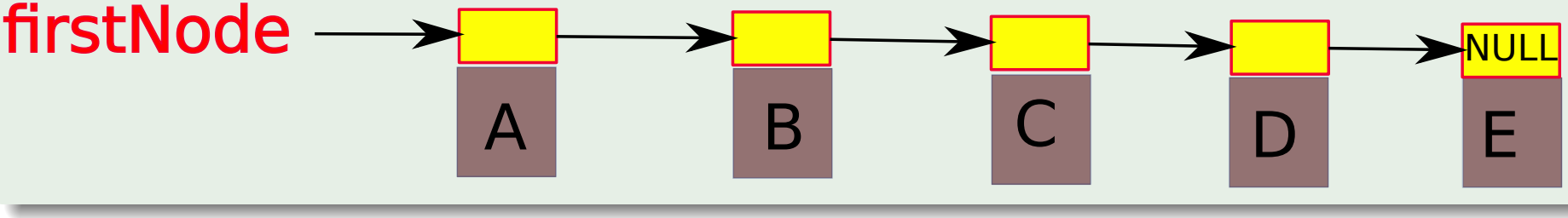
That is way



Outline

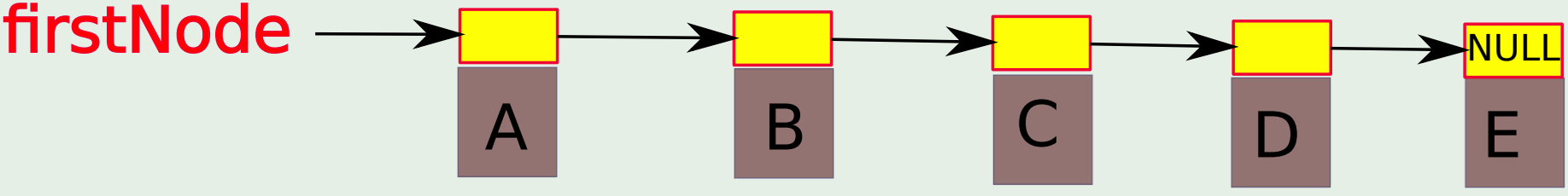
Derivation from a Chain

Stack top is either left end or right end of linear list



Derivation from a Chain

Stack top is either left end or right end of linear list

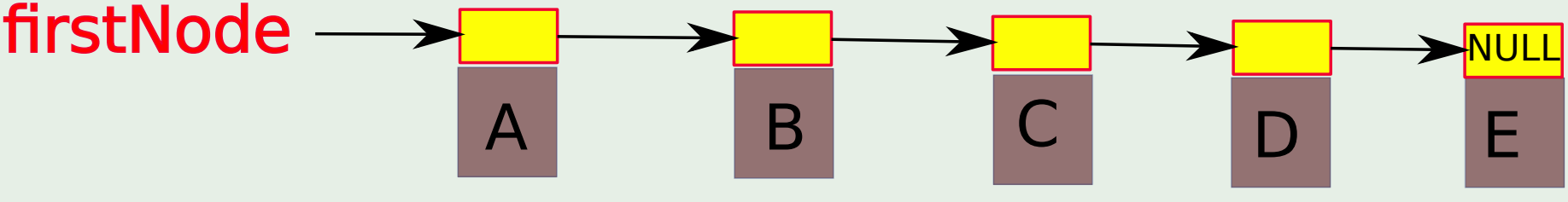


Thus

- empty() $\Longrightarrow$ isEmpty()
  - O(1) time

Derivation from a Chain

Stack top is either left end or right end of linear list

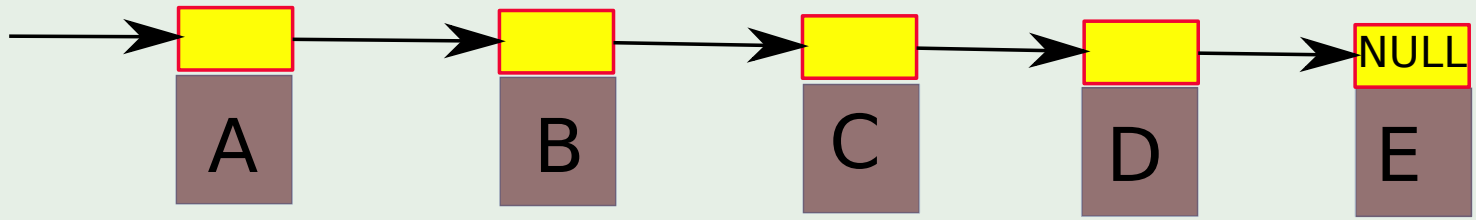


Thus

- empty() $\Longrightarrow$ isEmpty()
  - O(1) time

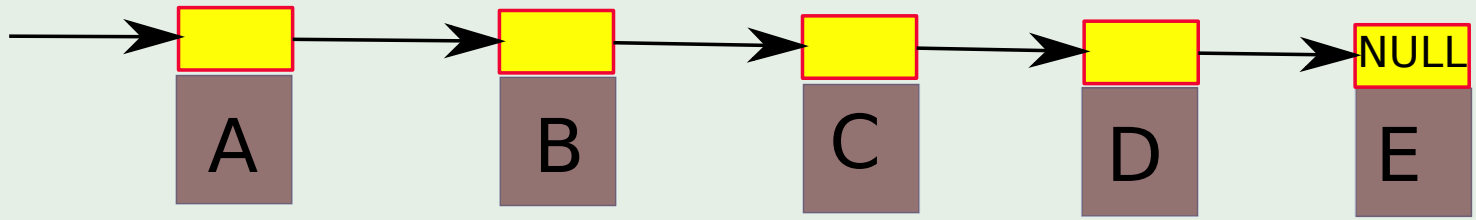
Derivation from a Chain

When top is left end of linear list



Derivation from a Chain

When top is left end of linear list

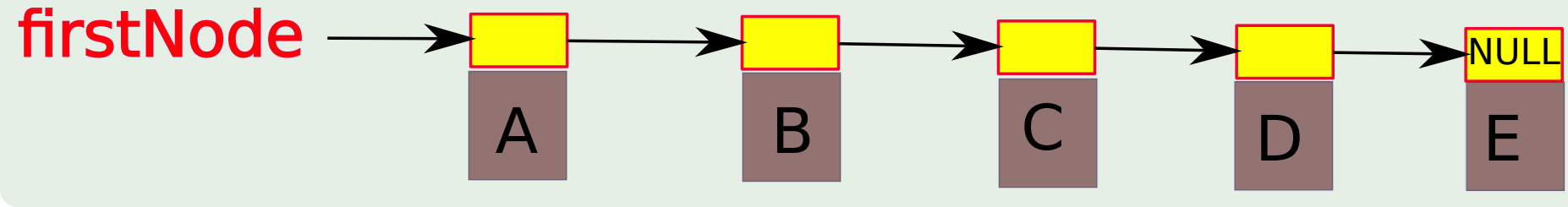


Thus

- peek() $\Longrightarrow$ get(0)
  - O(1) time
- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(0)
  - O(1) time

Derivation from a Chain

When top is left end of linear list

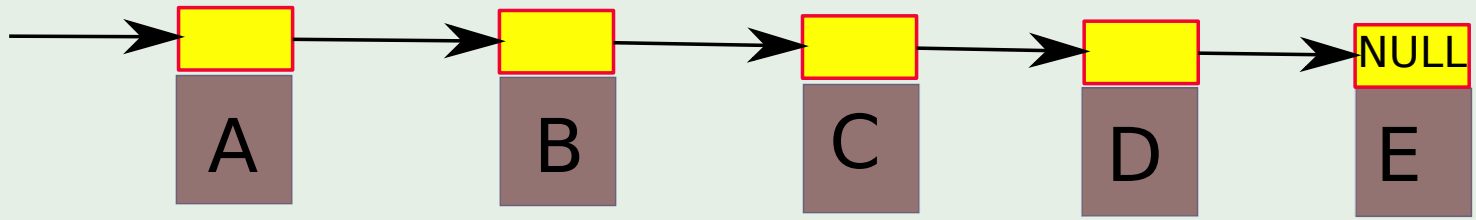


Thus

- peek() $\Longrightarrow$ get(0)
  - O(1) time
- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(0)
  - O(1) time

Derivation from a Chain

When top is left end of linear list

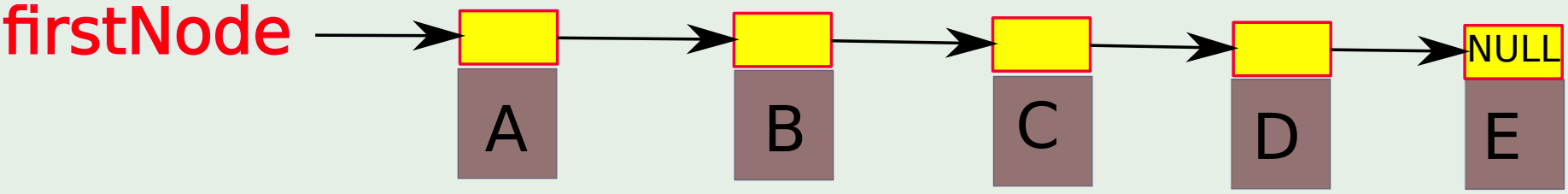


Thus

- peek() $\Longrightarrow$ get(0)
  - O(1) time
- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(0)
  - O(1) time

Derivation from a Chain

When top is left end of linear list

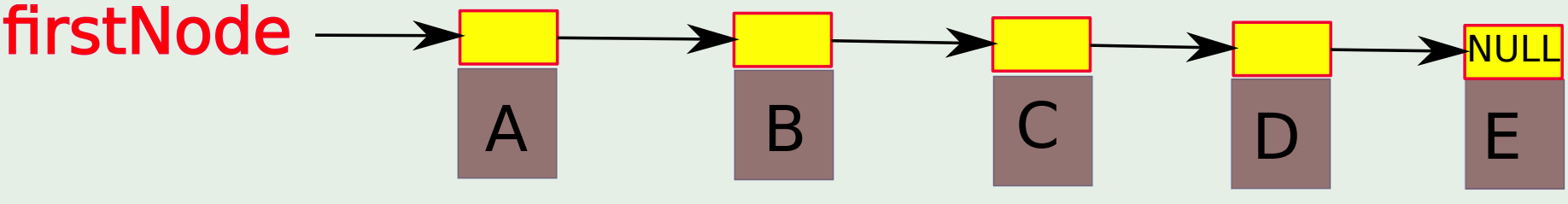


Thus

- peek() $\Longrightarrow$ get(0)
  - O(1) time
- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(0)
  - O(1) time

Derivation from a Chain

When top is left end of linear list

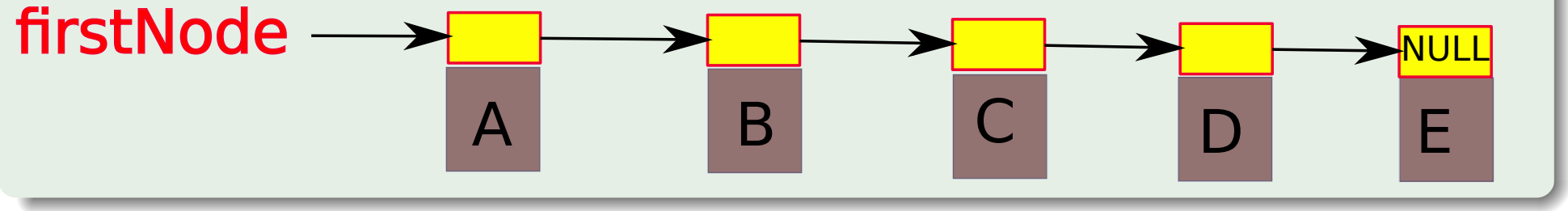


Thus

- peek() $\Longrightarrow$ get(0)
  - O(1) time
- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(0)
  - O(1) time

Derivation from a Chain

When top is left end of linear list

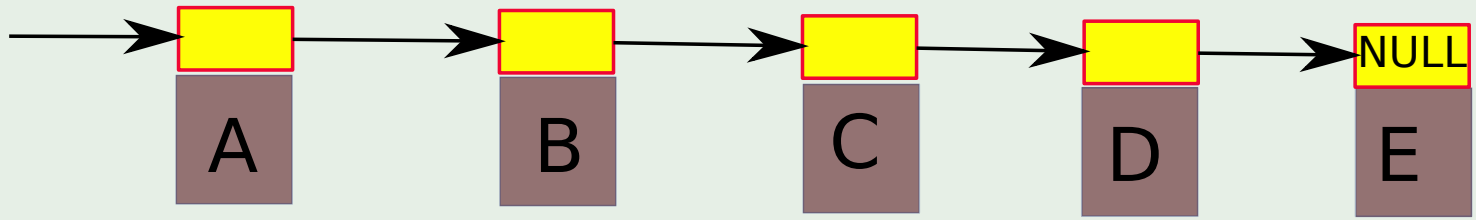


Thus

- peek() $\Longrightarrow$ get(0)
  - O(1) time
- push(theObject) $\Longrightarrow$ add(0, theObject)
  - O(1) time
- pop() $\Longrightarrow$ remove(0)
  - O(1) time

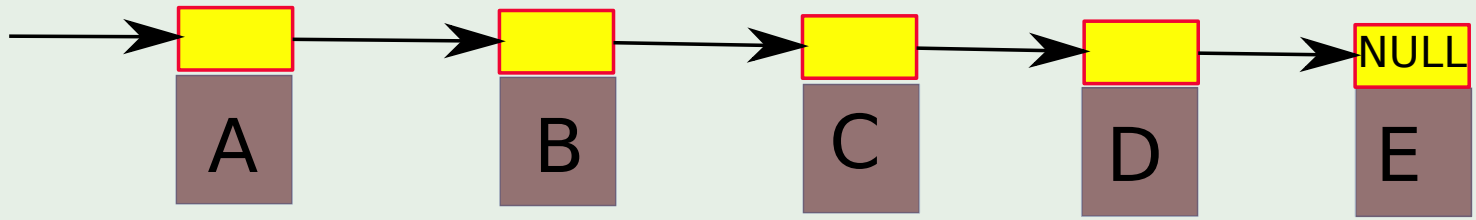
Derivation from a Chain

When top is left end of linear list



Derivation from a Chain

When top is left end of linear list

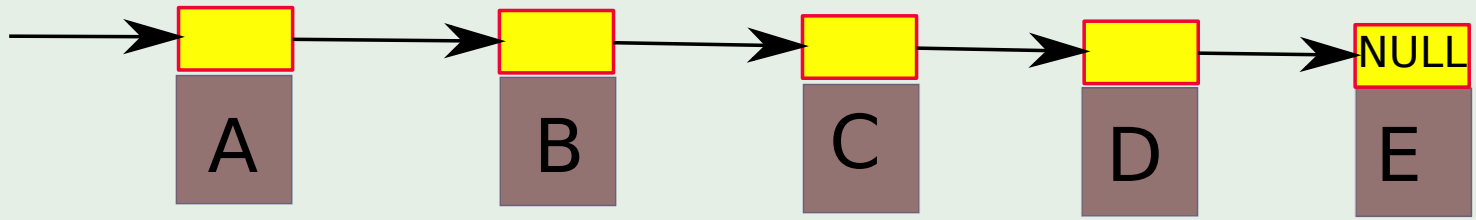


Thus

- peek() $\Longrightarrow$ get(size()-1)
  - O(size) time
- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(size) time

Derivation from a Chain

When top is left end of linear list

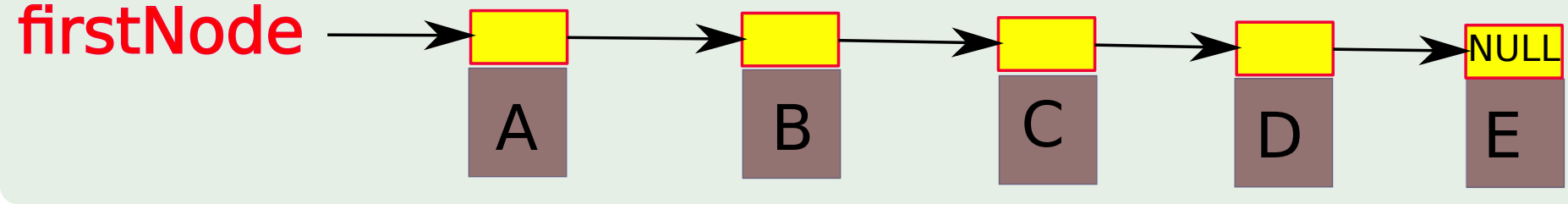


Thus

- peek() $\Longrightarrow$ get(size()-1)
  - O(size) time
- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(size) time

Derivation from a Chain

When top is left end of linear list

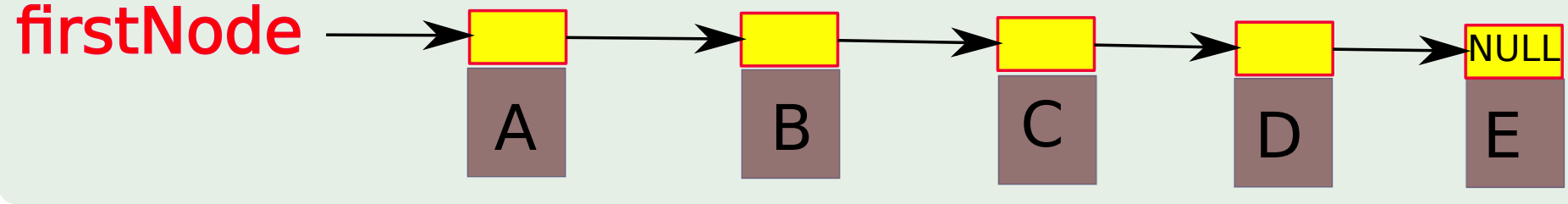


Thus

- peek() $\Longrightarrow$ get(size()-1)
  - O(size) time
- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(size) time

Derivation from a Chain

When top is left end of linear list

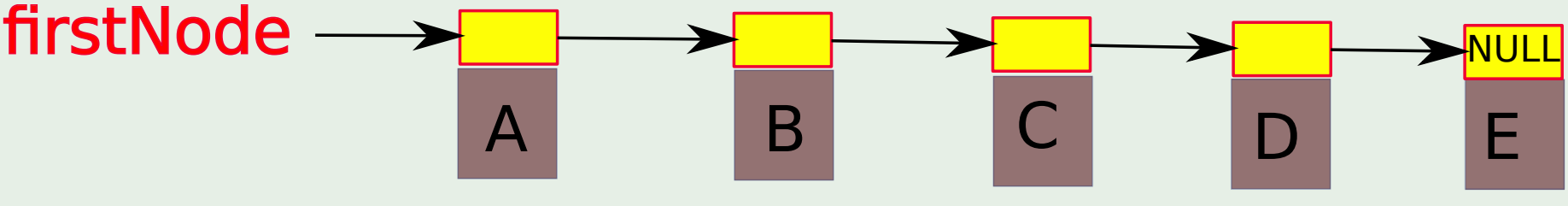


Thus

- peek() $\Longrightarrow$ get(size()-1)
  - O(size) time
- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(size) time

Derivation from a Chain

When top is left end of linear list

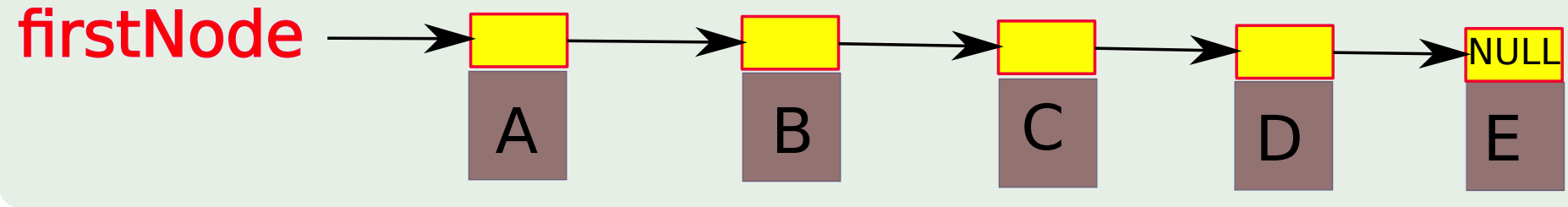


Thus

- peek() $\Longrightarrow$ get(size()-1)
  - O(size) time
- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(size) time

Derivation from a Chain

When top is left end of linear list



Thus

- peek() $\Longrightarrow$ get(size()-1)
  - O(size) time
- push(theObject) $\Longrightarrow$ add(size(), theObject)
  - O(size) time
- pop() $\Longrightarrow$ remove(size()-1)
  - O(size) time

Thus

The Moral of the Story

You must use the left end of list as top of stack

Derive From ArrayLinearList

Code

```
package MyStack;
import java.util.*;  // It has stack exception
public class DerivedArrayStack extends
             ArrayLinearList<Item>
{
          // constructors come here
          // Stack interface methods come here
}
```

Derive From ArrayLinearList

Constructors

```
/** create a stack with the given initial
  * capacity */
public DerivedArrayStack(int initialCapacity)
          {super(initialCapacity);}
/** create a stack with initial capacity 10 */
public DerivedArrayStack()
          {this(10);}
```

empty() And peek()

Code

```
public boolean empty()
   {return isEmpty();}

public Object peek()
{
if (empty())
   throw new EmptyStackException();
return get(size() - 1);
}
```

push(theObject) And pop()

Code

```
public void push(Object theElement)
   {add(size(), theElement);}

public Object pop()
{
if (empty())
   throw new EmptyStackException();
return remove(size() - 1);
}
```

The Pros

The merits of deriving from ArrayLinearList

- Code for derived class is quite simple and easy to develop.
- Code is expected to require little debugging.
- Code for other stack implementations such as a linked implementation are easily obtained.
  - Just replace extends ArrayLinearList with extends Chain.
  - For efficiency reasons we must also make changes to use the left end of the list as the stack top rather than the right end.

The Pros

The merits of deriving from ArrayLinearList

- Code for derived class is quite simple and easy to develop.
- Code is expected to require little debugging.
- Code for other stack implementations such as a linked implementation are easily obtained.
  - Just replace extends ArrayLinearList with extends Chain.
  - For efficiency reasons we must also make changes to use the left end of the list as the stack top rather than the right end.

The Pros

The merits of deriving from ArrayLinearList

- Code for derived class is quite simple and easy to develop.
- Code is expected to require little debugging.
- Code for other stack implementations such as a linked implementation are easily obtained.
  - Just replace extends ArrayLinearList with extends Chain.
  - For efficiency reasons we must also make changes to use the left end of the list as the stack top rather than the right end.

The Pros

The merits of deriving from ArrayLinearList

- Code for derived class is quite simple and easy to develop.
- Code is expected to require little debugging.
- Code for other stack implementations such as a linked implementation are easily obtained.
  - Just replace extends ArrayLinearList with extends Chain.
  - For efficiency reasons we must also make changes to use the left end of the list as the stack top rather than the right end.

The Pros

The merits of deriving from ArrayLinearList

- Code for derived class is quite simple and easy to develop.
- Code is expected to require little debugging.
- Code for other stack implementations such as a linked implementation are easily obtained.
  - Just replace extends ArrayLinearList with extends Chain.
  - For efficiency reasons we must also make changes to use the left end of the list as the stack top rather than the right end.

The Cons

Then

- All public methods of ArrayLinearList are performed following the stack structure:
  - get(0) ... get bottom element
  - remove(5)... pop
  - add(3, x) .... push
- So we do not have a true stack implementation.
  - We must override undesired methods.

The Cons

Then

- All public methods of ArrayLinearList are performed following the stack structure:
  - get(0) ... get bottom element
  - remove(5) ... pop
  - add(3, x) ... push
- So we do not have a true stack implementation.
  - We must override undesired methods.

The Cons

Then

- All public methods of ArrayLinearList are performed following the stack structure:
  - get(0) ... get bottom element
  - remove(5)... pop
  - add(3, x) ... push
- So we do not have a true stack implementation.
  - We must override undesired methods.

The Cons

Then

- All public methods of ArrayLinearList are performed following the stack structure:
  - get(0) ... get bottom element
  - remove(5)... pop
  - add(3, x) ... push
- So we do not have a true stack implementation.
  - We must override undesired methods.

The Cons

Then

- All public methods of ArrayLinearList are performed following the stack structure:
  - get(0) ... get bottom element
  - remove(5)... pop
  - add(3, x) ... push
- So we do not have a true stack implementation.
  - We must override undesired methods.

The Cons

Unnecessary work is done by the code.

Examples

The Cons

Unnecessary work is done by the code.

Examples

Example I

- **peek()** verifies that the stack is not empty before get is invoked.
  - The index check done by get is, therefore, not needed.

The Cons

Unnecessary work is done by the code.

Examples

Example I

- **peek()** verifies that the stack is not empty before get is invoked.
  - The index check done by get is, therefore, not needed.

Example II

- **add(size(), theElement)** does an index check and a for loop that is not entered.
  - Neither is needed.

The Cons

Unnecessary work is done by the code.

Examples

Example I

- **peek()** verifies that the stack is not empty before get is invoked.
  - The index check done by get is, therefore, not needed.

Example II

- **add(size(), theElement)** does an index check and a for loop that is not entered.
  - Neither is needed.

The Cons

Unnecessary work is done by the code.

Examples

Example I

- **peek()** verifies that the stack is not empty before get is invoked.
  - The index check done by get is, therefore, not needed.

Example II

- **add(size(), theElement)** does an index check and a for loop that is not entered.
  - Neither is needed.

Then

Thus

So the derived code runs slower than necessary.

Moral of the Story

First

Code developed from scratch will run faster but will take more time (cost) to develop.

Second

Tradeoff between software development cost and performance.

Third

Tradeoff between time to market and performance.

Fourth

It could be easy to develop the code first and later refine it to improve performance.

Moral of the Story

First

Code developed from scratch will run faster but will take more time (cost) to develop.

Second

Tradeoff between software development cost and performance.

Third

Tradeoff between time to market and performance.

Fourth

It could be easy to develop the code first and later refine it to improve performance.

Moral of the Story

First

Code developed from scratch will run faster but will take more time (cost) to develop.

Second

Tradeoff between software development cost and performance.

Third

Tradeoff between time to market and performance.

Fourth

It could be easy to develop the code first and later refine it to improve performance.

Moral of the Story

First

Code developed from scratch will run faster but will take more time (cost) to develop.

Second

Tradeoff between software development cost and performance.

Third

Tradeoff between time to market and performance.

Fourth

It could be easy to develop the code first and later refine it to improve performance.

Example

A slow pop

1. if (empty())
2. throw new EmptyStackException();
3. return remove(size() - 1);

Example

A slow pop

```
1 if (empty())
2         throw new EmptyStackException();
3 return remove(size() - 1);
```

Faster Code

```
1 try {return remove(size() - 1);}
2 catch(IndexOutOfBoundsException e)
3         {throw new EmptyStackException();}
```

Example

A slow pop

```
1 if (empty())
2         throw new EmptyStackException();
3 return remove(size() - 1);
```

Faster Code

```
1 try {return remove(size() - 1);}
2 catch(IndexOutOfBoundsException e)
3         {throw new EmptyStackException();}
```

Example

A slow pop

1. if (empty())
2. throw new EmptyStackException();
3. return remove(size() - 1);

Faster Code

1. try {return remove(size() - 1);}
2. catch(IndexOutOfBoundsException e)
3. {throw new EmptyStackException();}

Example

A slow pop

1. if (empty())
2. throw new EmptyStackException();
3. return remove(size() - 1);

Faster Code

1. try {return remove(size() - 1);}
2. catch(IndexOutOfBoundsException e)
3. {throw new EmptyStackException();}

Example

A slow pop

```
1 if (empty())
2         throw new EmptyStackException();
3 return remove(size() - 1);
```

Faster Code

```
1 try {return remove(size() - 1);}
2 catch(IndexOutOfBoundsException e)
3         {throw new EmptyStackException();}
```

Code From Scratch

First

- Use a 1D array stack whose data type is Item.
  - same as using array element in ArrayLinearList

Code From Scratch

First

- Use a 1D array stack whose data type is Item.
  - same as using array element in ArrayLinearList

Second

- Use an int variable top.
  - Stack elements are in stack[0:top].
  - Top element is in stack[top].
  - Bottom element is in stack[0].
  - Stack is empty iff top == -1.
  - Number of elements in stack is top+1.

Code From Scratch

First

- Use a 1D array stack whose data type is Item.
  - same as using array element in ArrayLinearList

Second

- Use an int variable top.
  - Stack elements are in stack[0:top].
  - Top element is in stack[top].
  - Bottom element is in stack[0].
  - Stack is empty iff top == -1.
  - Number of elements in stack is top+1.

Code From Scratch

First

- Use a 1D array stack whose data type is Item.
  - same as using array element in ArrayLinearList

Second

- Use an int variable top.
  - Stack elements are in stack[0:top].
  - Top element is in stack[top].
  - Bottom element is in stack[0].
  - Stack is empty iff top = -1.
  - Number of elements in stack is top+1.

Code From Scratch

First

- Use a 1D array stack whose data type is Item.
  - same as using array element in ArrayLinearList

Second

- Use an int variable top.
  - Stack elements are in stack[0:top].
  - Top element is in stack[top].
  - Bottom element is in stack[0].
  - Stack is empty iff top = -1.
  - Number of elements in stack is top+1.

Code From Scratch

First

- Use a 1D array stack whose data type is Item.
  - same as using array element in ArrayLinearList

Second

- Use an int variable top.
  - Stack elements are in stack[0:top].
  - Top element is in stack[top].
  - Bottom element is in stack[0].
  - Stack is empty iff top == -1.
  - Number of elements in stack is top+1.

Code From Scratch

First

- Use a 1D array stack whose data type is Item.
  - same as using array element in ArrayLinearList

Second

- Use an int variable top.
  - Stack elements are in stack[0:top].
  - Top element is in stack[top].
  - Bottom element is in stack[0].
  - Stack is empty iff top = -1.
  - Number of elements in stack is top+1.

Code From Scratch

First

- Use a 1D array stack whose data type is Item.
  - same as using array element in ArrayLinearList

Second

- Use an int variable top.
  - Stack elements are in stack[0:top].
  - Top element is in stack[top].
  - Bottom element is in stack[0].
  - Stack is empty iff top = -1.
  - Number of elements in stack is top+1.

Code From Scratch

Data Memember

```
package Stack;
import java.util.EmptyStackException;
import utilities.*;
public class ArrayStack<Item>
          implements Stack<Item>
{
// data members
int top;          // current top of stack
Item [] stack;    // element array

// Etc
}
```

Constructors

Code

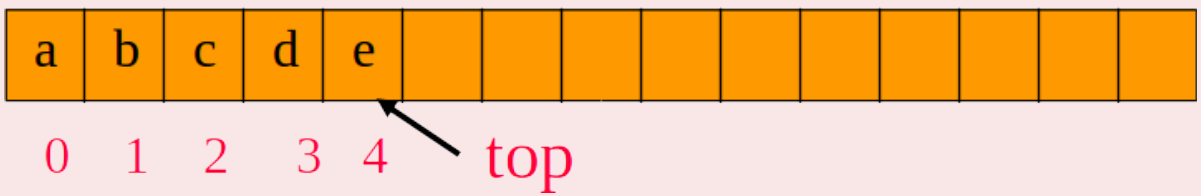
```
public ArrayStack(int initialCapacity)
{
  if (initialCapacity < 1)
    throw new IllegalArgumentException
     ("initialCapacity must be >= 1");
  this.stack = (Item[]) new Object[initialCapacity];
  top = -1;
}
public ArrayStack()
  {this(10);}
```

Push(...)

Code

```
public void push(Object theElement)
{
// increase array size if necessary
  if (top == stack.length - 1)
       stack = ChangeArrayLength.changeLength1D
               (stack, 2 * stack.length);
// put theElement at the top of the stack
  stack[++top] = theElement;
}
```

Thus

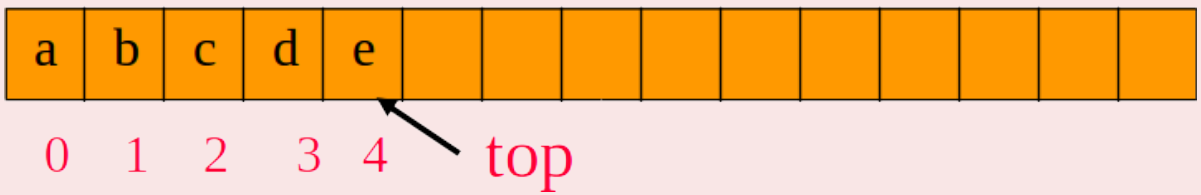


Pop()

Code

```
public Item pop()
{
 if (empty())
      throw new EmptyStackException();
 Object topElement = stack[top];
 stack[top--] = null;  // enable garbage collection
 return topElement;
}
```

Thus



Actually

We have the following

java.util.Stack

- It derives from java.util.Vector.
- java.util.Vector is an array implementation of a linear list.

Performance of 500,000 pop, push, and peek operations

We have

Class	500,000
DerivedArrayStack	0.38 s
Scratch Array Stack	0.22 s
DerivedArrayStackWithCatch	0.33 s
DerivedLinkedStack	3.20 s
Scratch LinkedStack	2.96 s