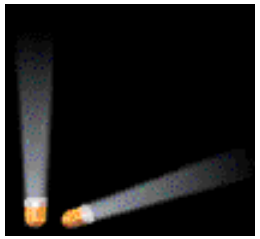
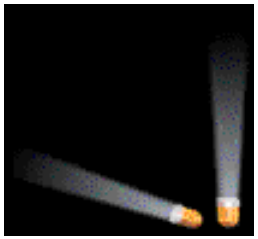
Stacks

Data structures

Fall 2018

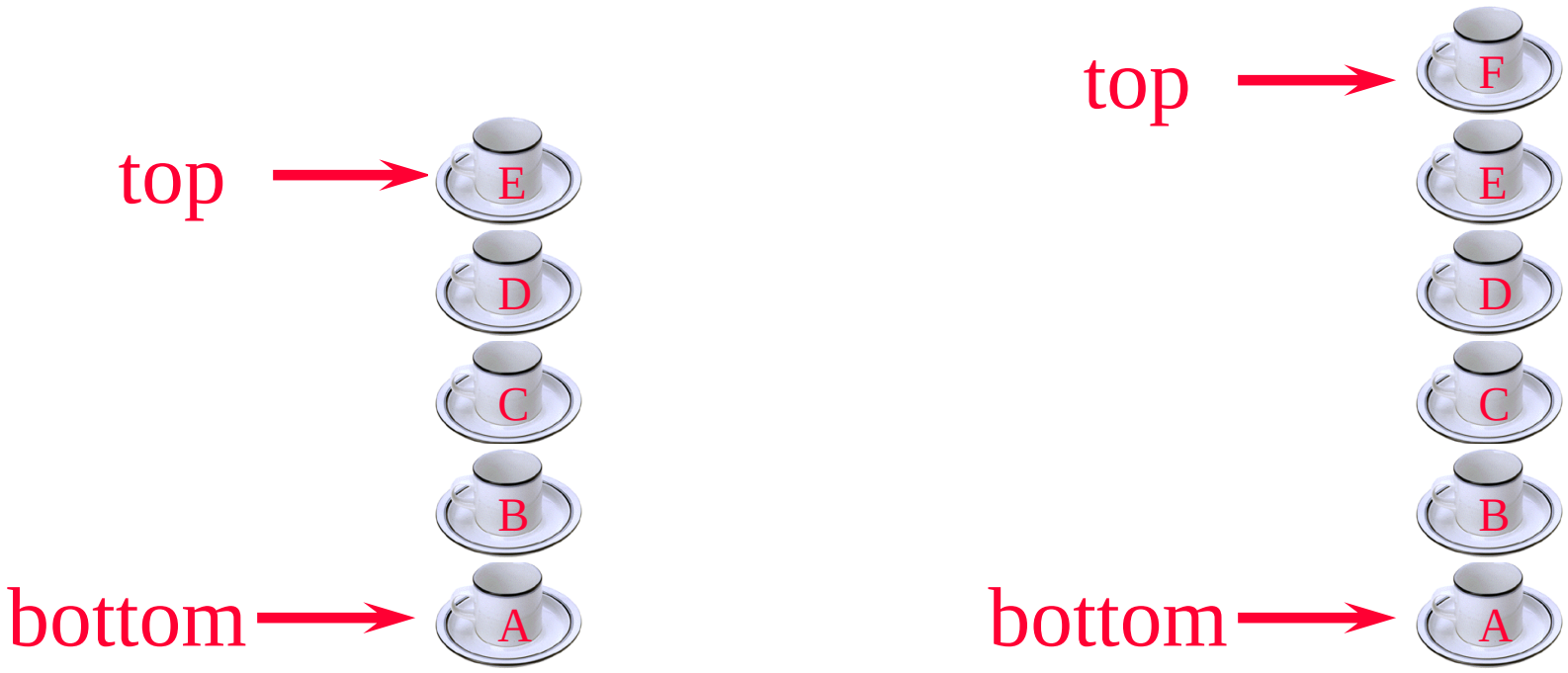
Stack





- Linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from the top end only.

Stack Of Cups



- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a LIFO list.

Parentheses Matching

- ((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis at 7 has no matching right parenthesis

Parentheses Matching

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack

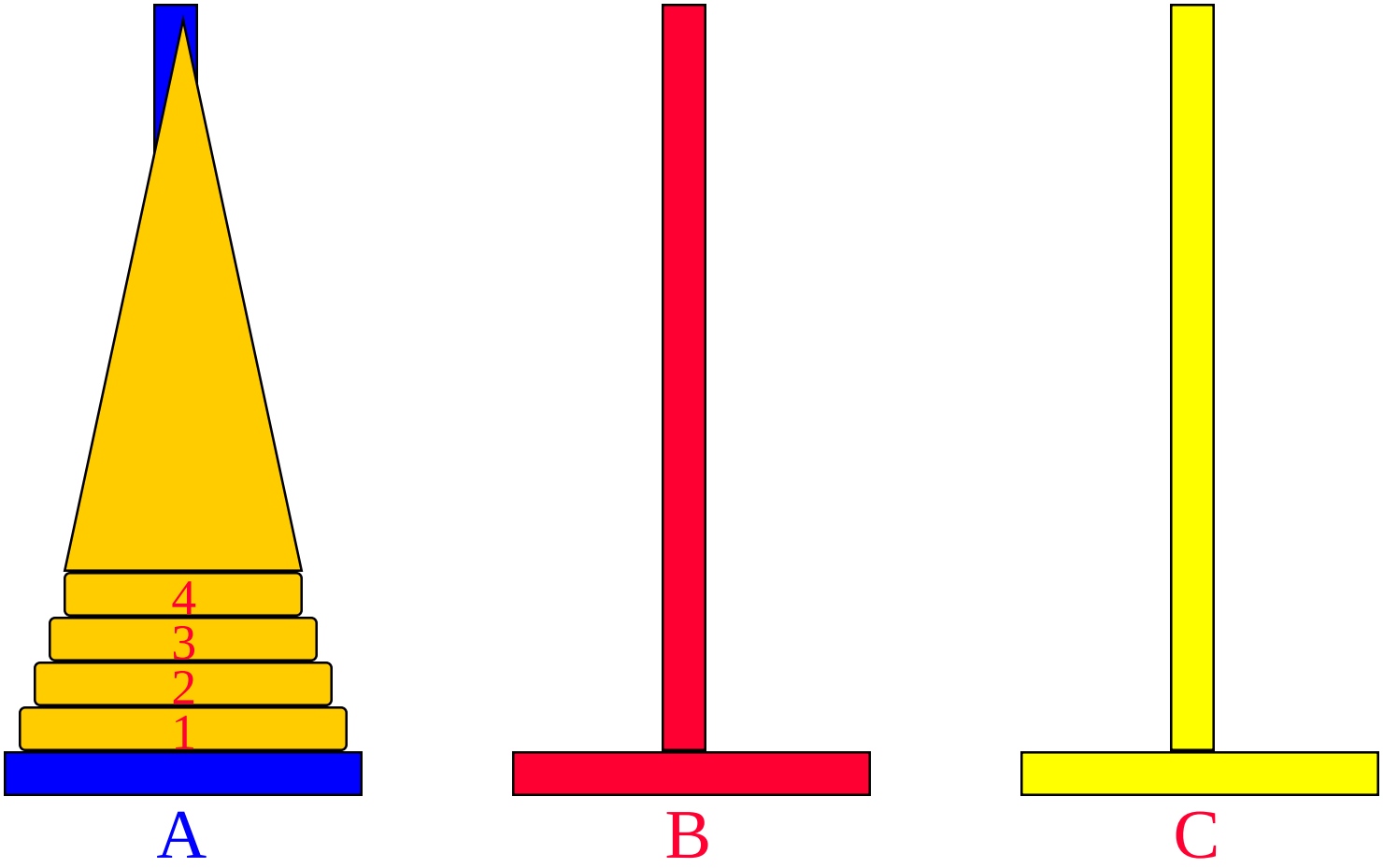
Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13) (15,19) (21,25) (27,31) (0,32)

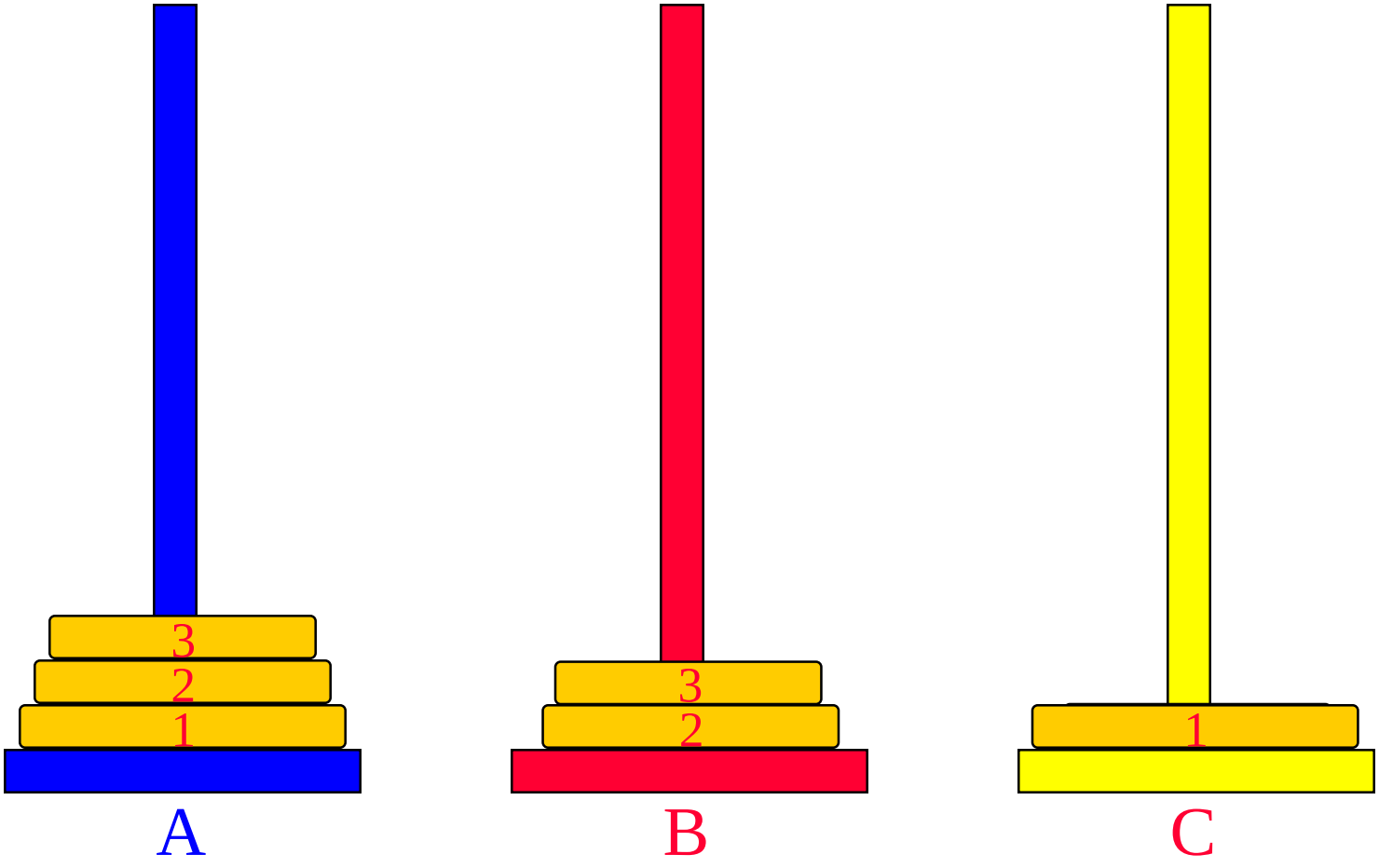
- and so on

Towers Of Hanoi



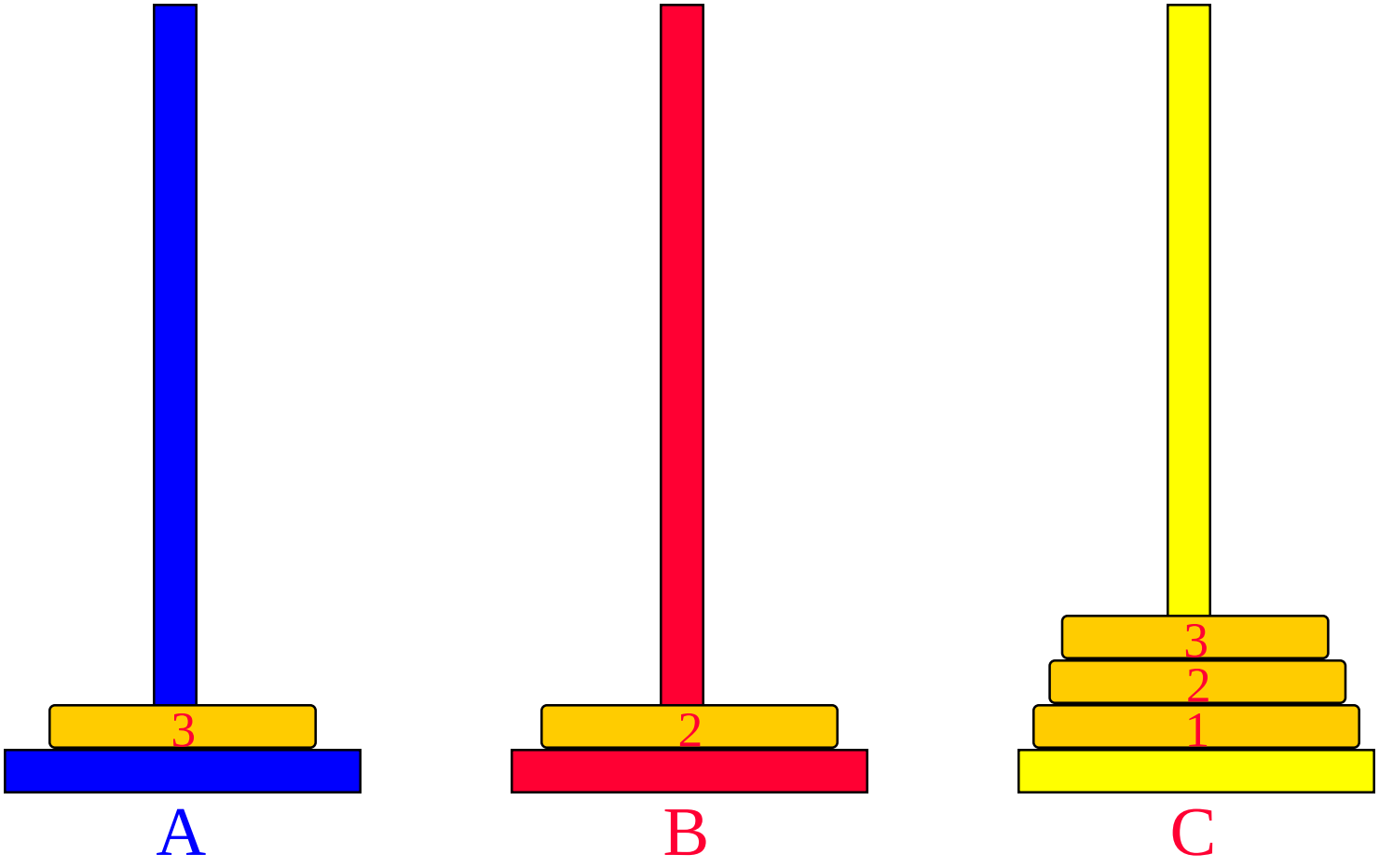
- 64 gold disks to be moved from tower A to tower C
- each tower operates as a stack
- cannot place bigger disk on top of a smaller one

Towers Of Hanoi



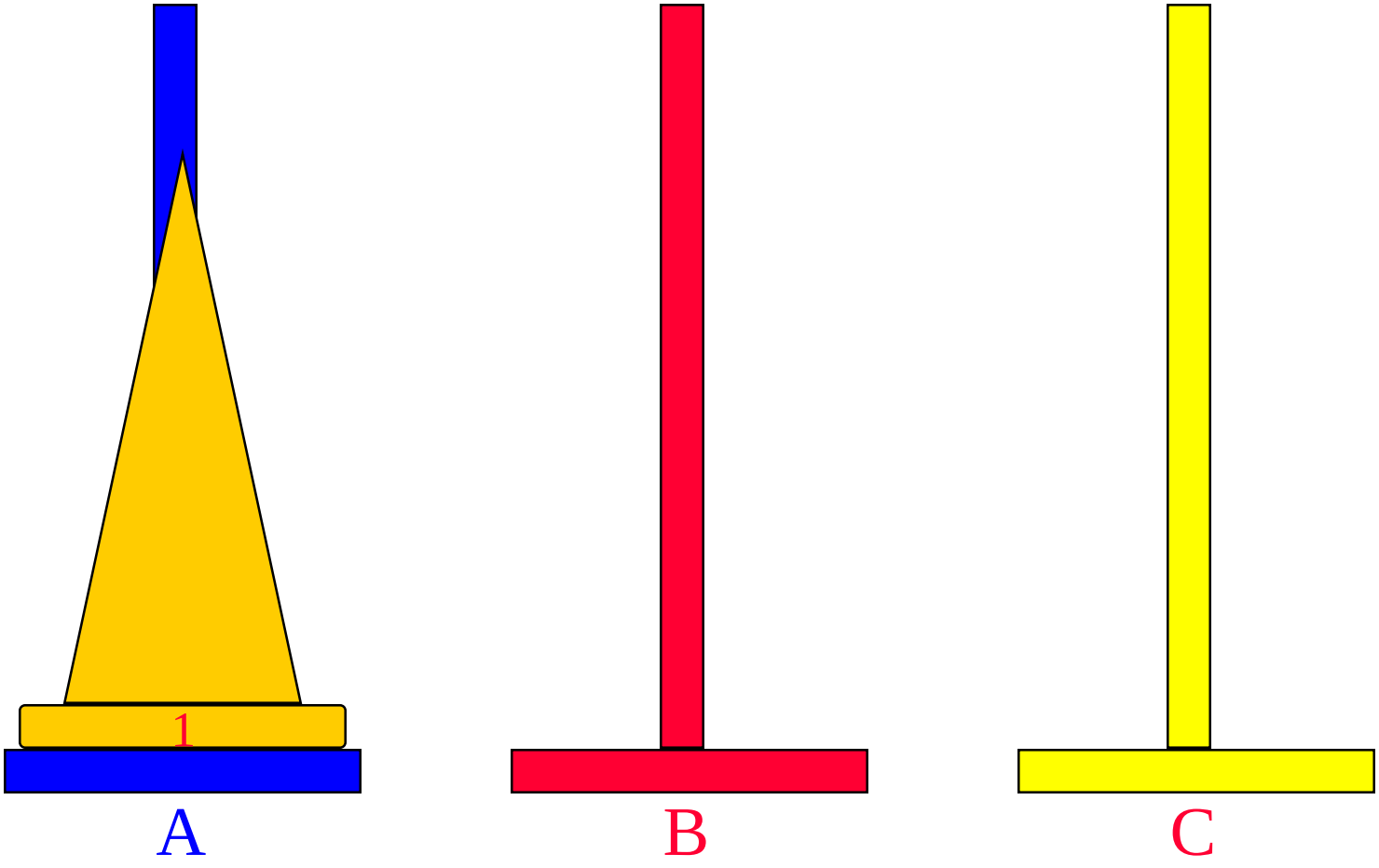
- 3-disk Towers Of Hanoi

Towers Of Hanoi



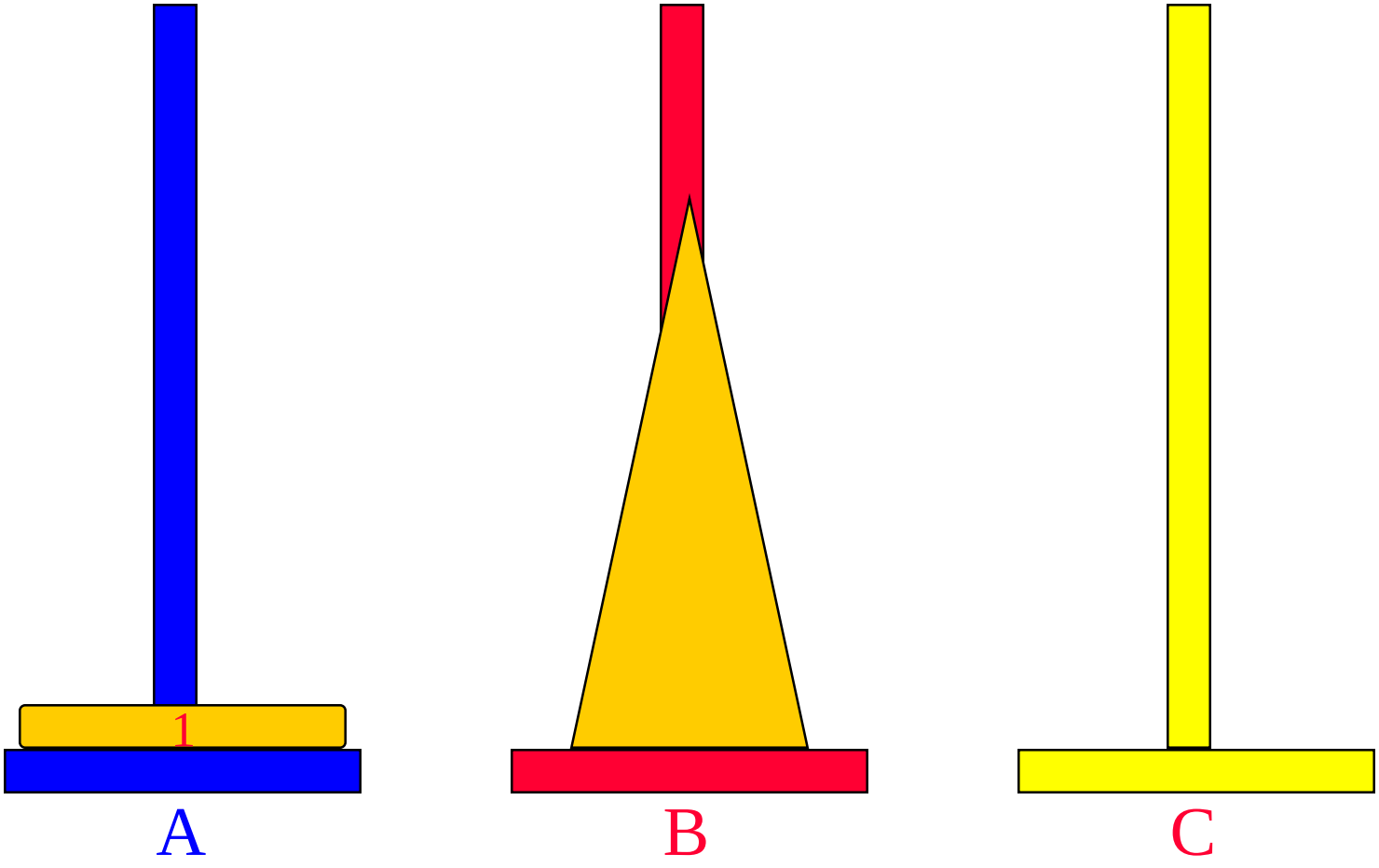
- 3-disk Towers Of Hanoi
- 7 disk moves

Recursive Solution



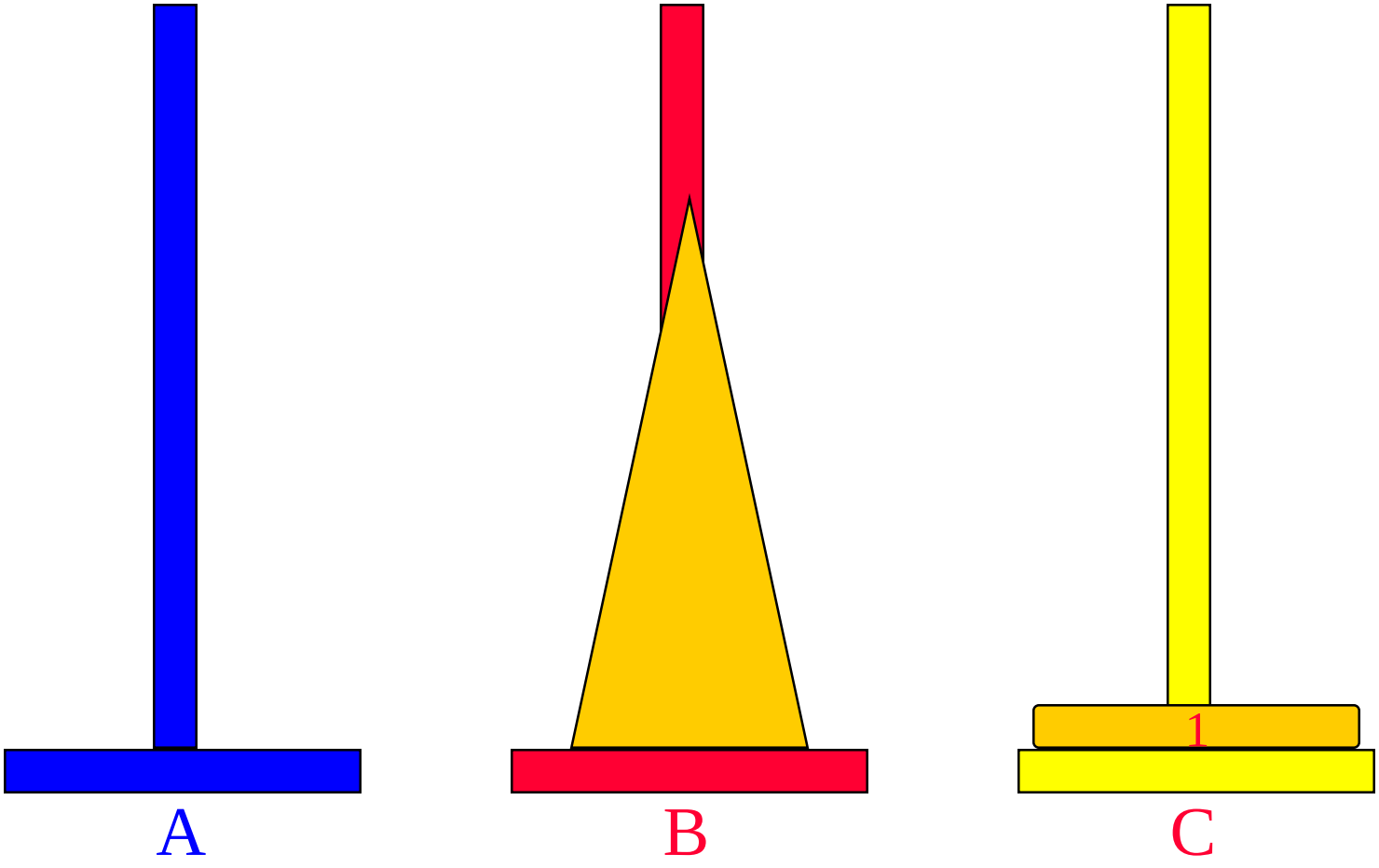
- n > 0 gold disks to be moved from A to C using B
- move top n-1 disks from A to B using C

Recursive Solution



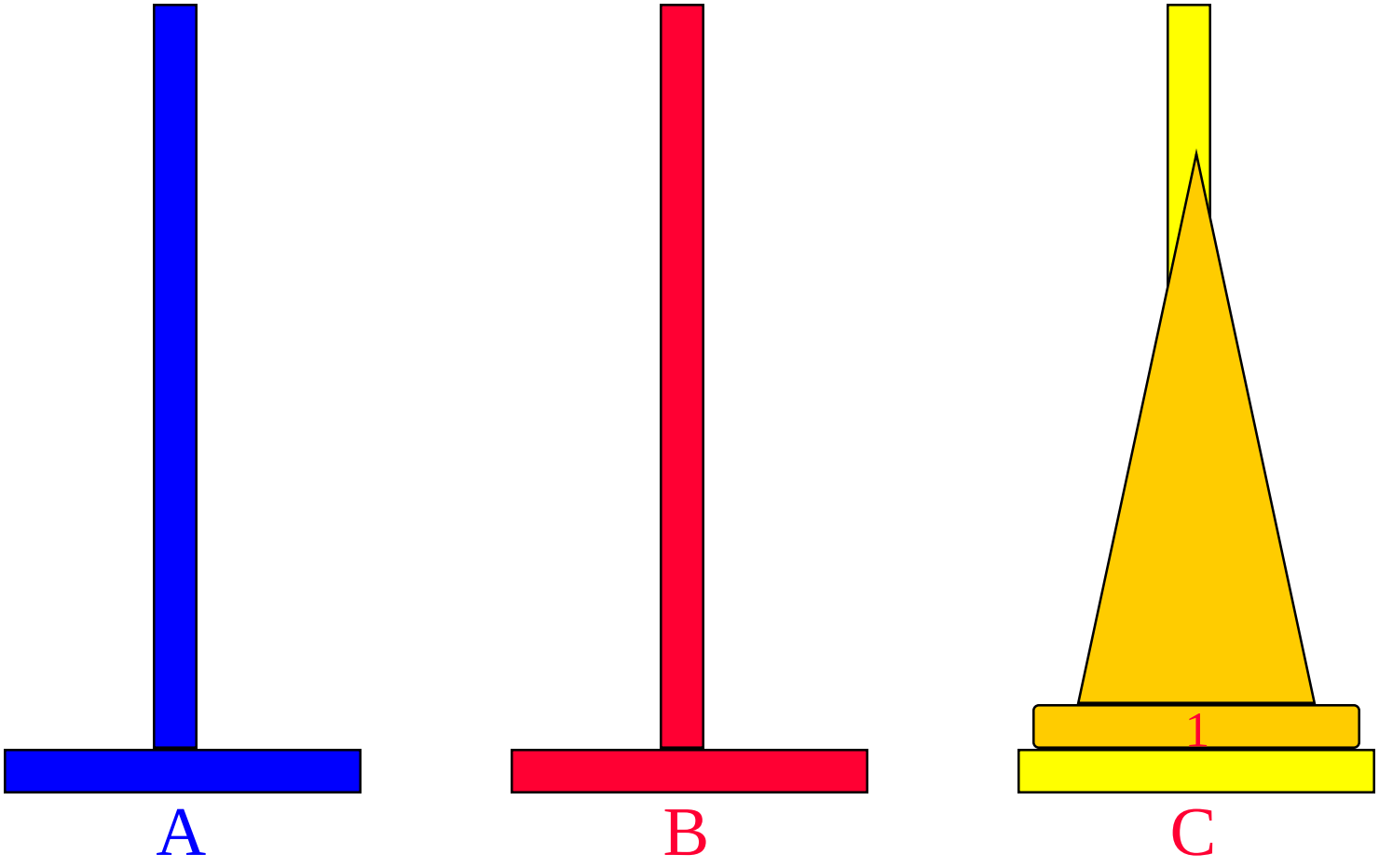
- move top disk from A to C

Recursive Solution



- move top n-1 disks from B to C using A

Recursive Solution

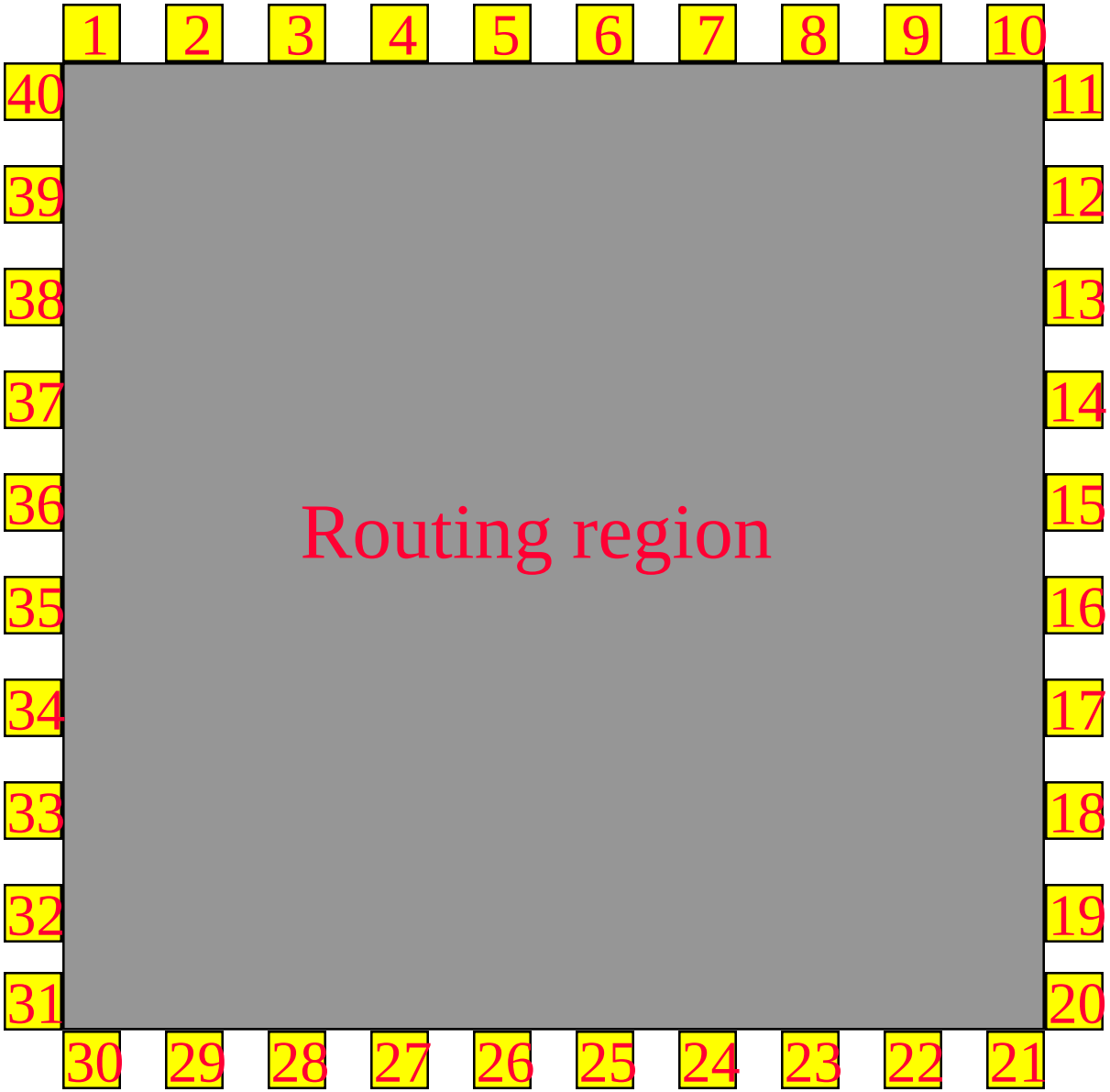


- moves(n) = 0 when n = 0
- moves(n) = 2*moves(n-1) + 1 = 2^n-1 when n > 0

Towers Of Hanoi

- moves(64) = $1.8 * 10^{19}$ (approximately)
- Performing 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.

Switch Box Routing

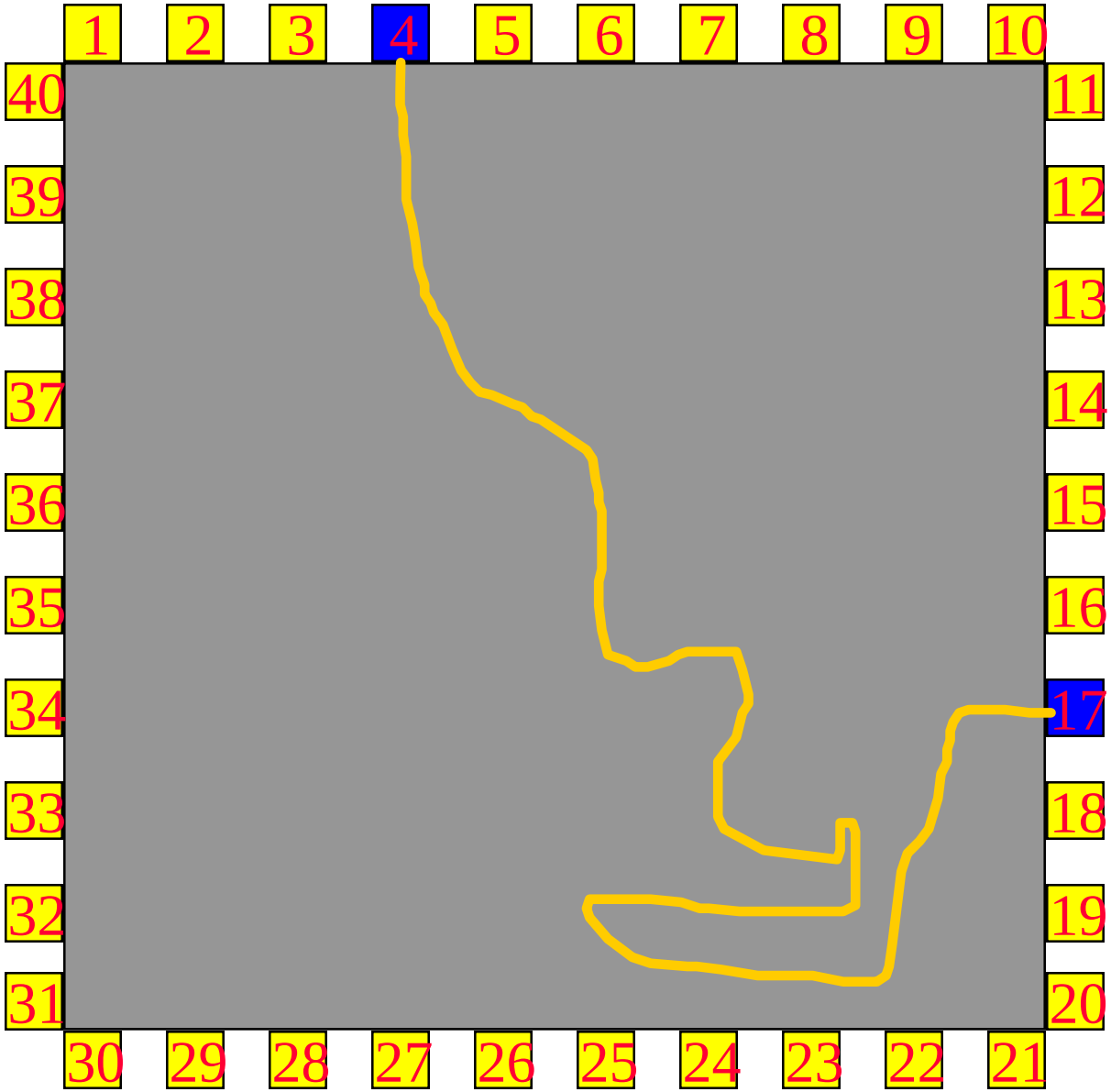


Input:
A set of pairs of pins.

Goal:
Determine whether it is possible to connect the pins without crossovers.

Routing A 2-pin Net

Routing for pins 1-3 and 18-40 is confined to lower left region.



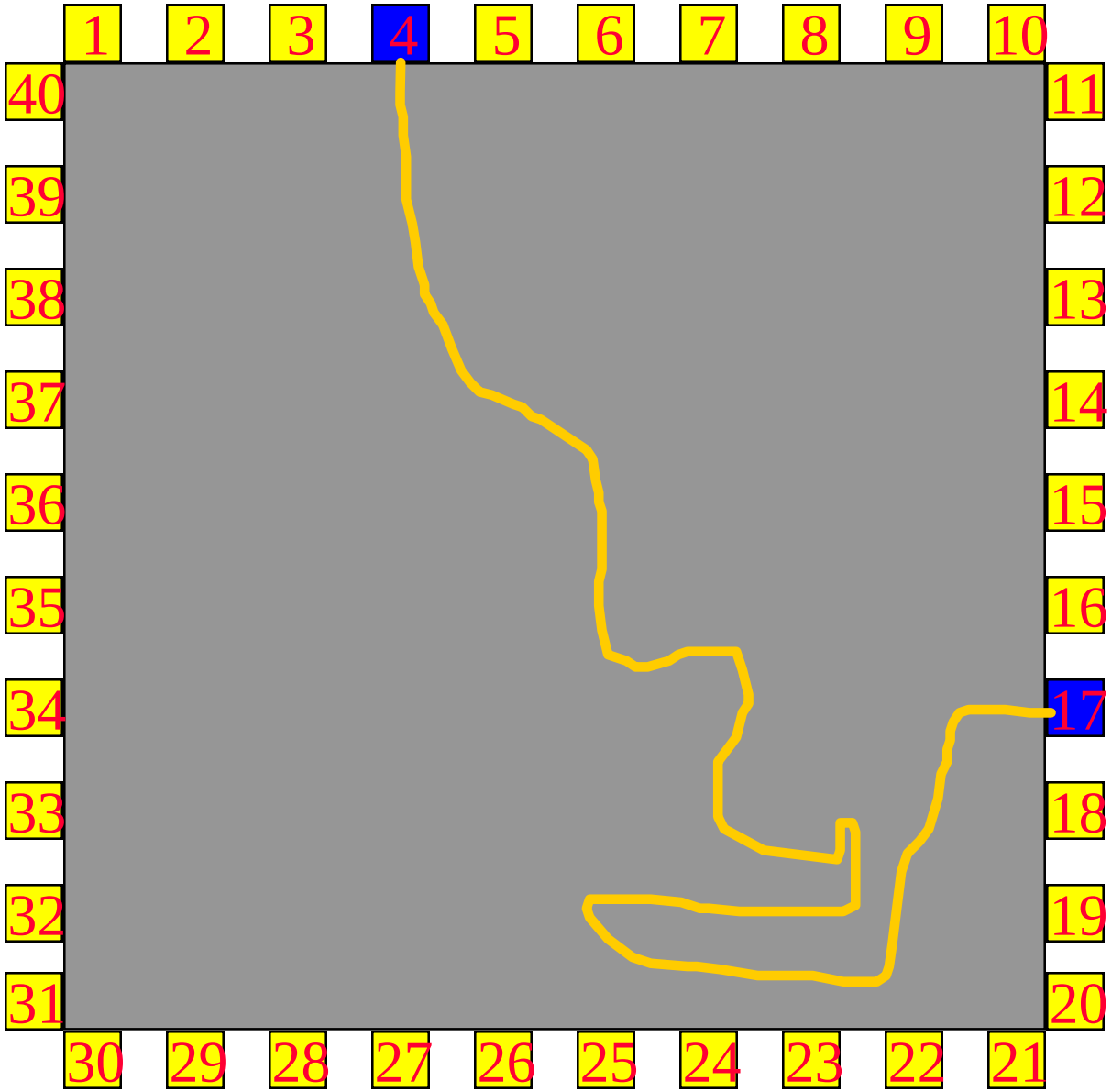
Routing for pins 5 through 16 is confined to upper right region.

Routing A 2-pin Net

(u,v), u<v is a 2-pin net.

u is start pin.

v is end pin.

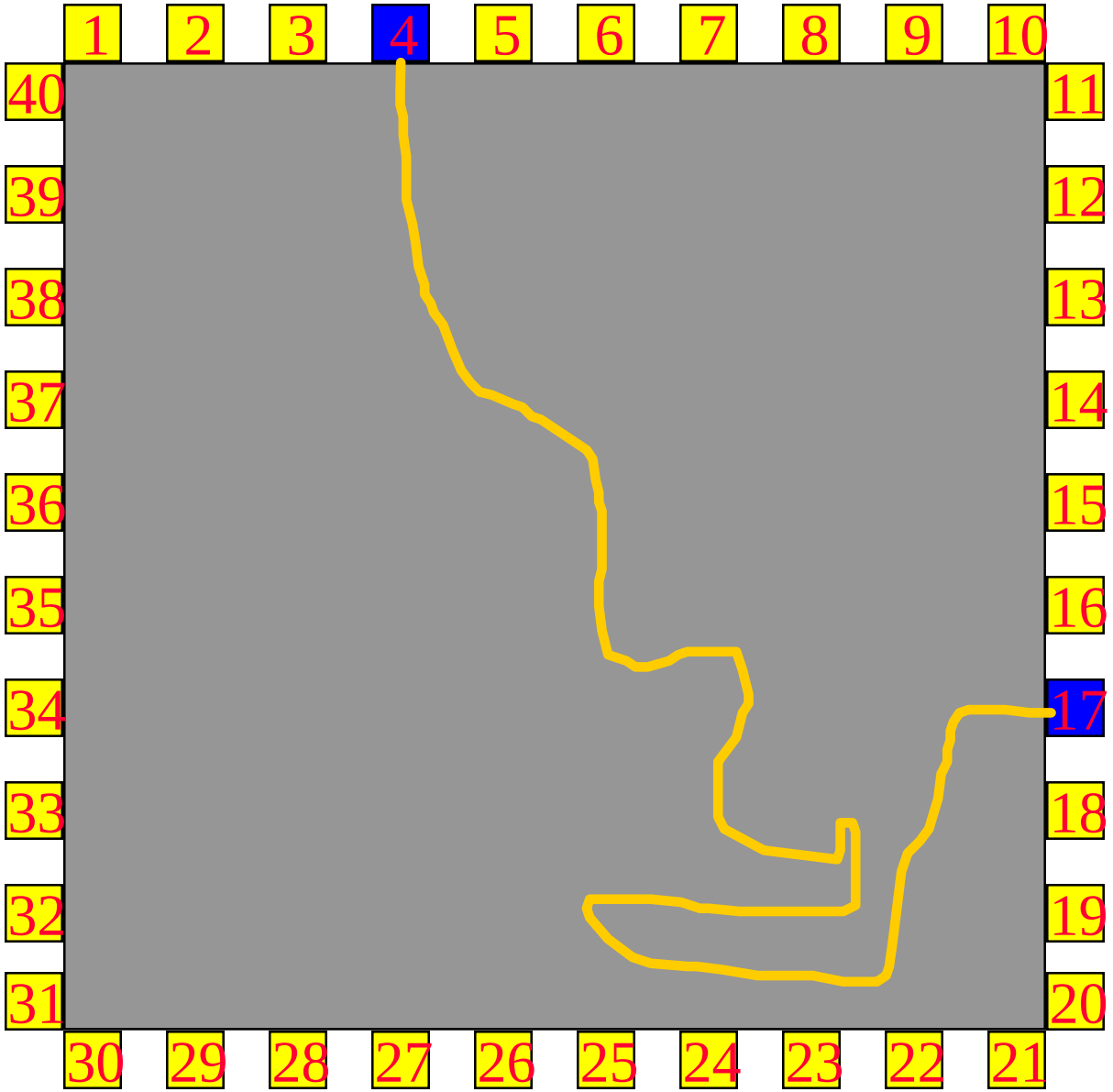


Examine pins in clock-wise order beginn-ing with pin 1.

Routing A 2-pin Net

Start pin => push onto stack.

End pin => start pin must be at top of stack.



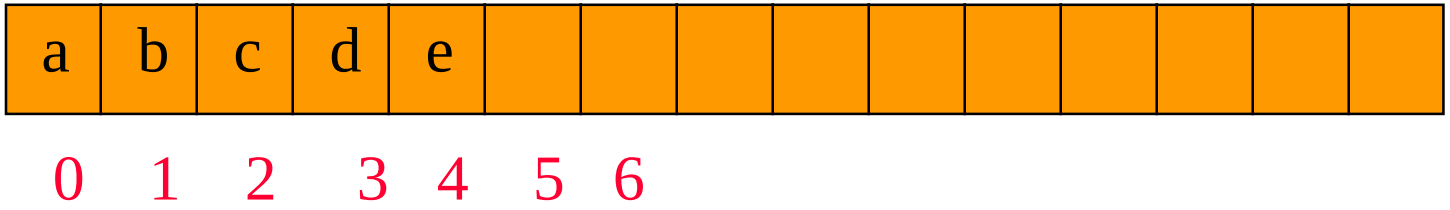
Stack Interface

```
public interface Stack
{
    public boolean empty();
    public Object peek();
    public void push(Object theObject);
    public Object pop();
}
```

Derive From A Linear List Class

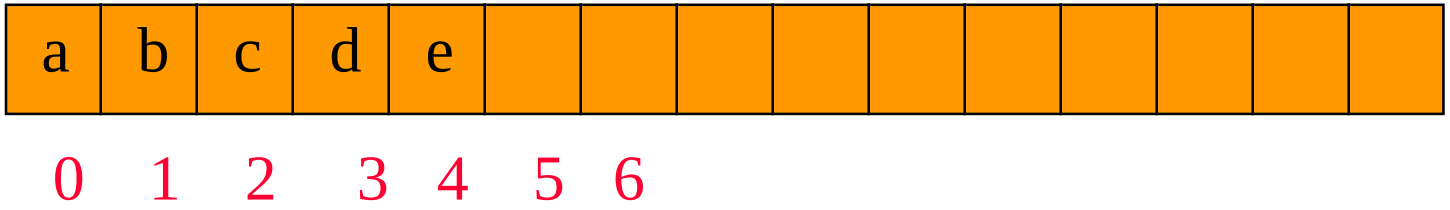
- ArrayLinearList
- Chain

Derive From ArrayLinearList



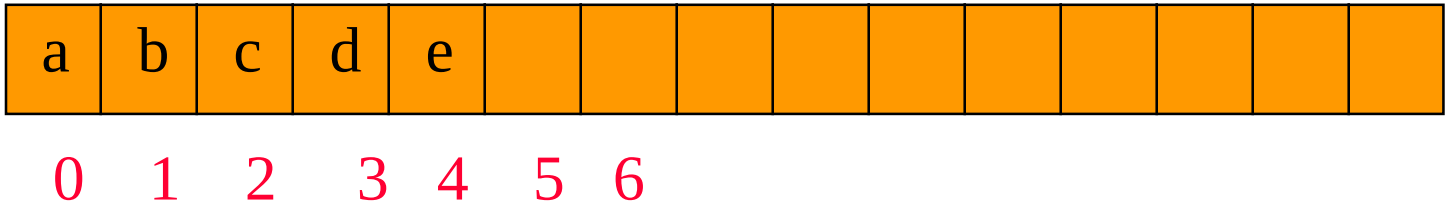
- stack top is either left end or right end of linear list
- empty() => isEmpty()
  - O(1) time
- peek() => get(0) or get(size() - 1)
  - O(1) time

Derive From ArrayLinearList



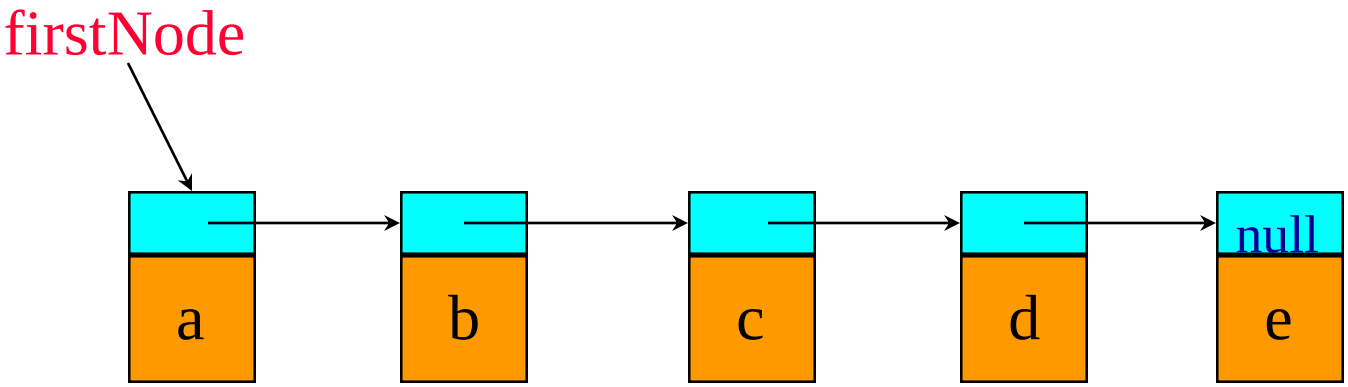
- when top is left end of linear list
  - push(theObject) => add(0, theObject)
  - O(size) time
  - pop() => remove(0)
  - O(size) time

Derive From ArrayLinearList



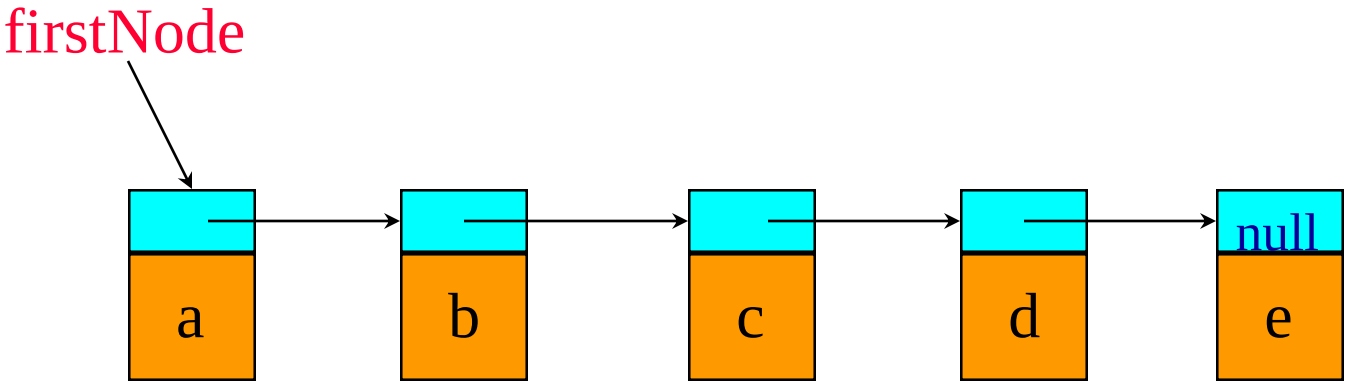
- when top is right end of linear list
  - push(theObject) => add(size(), theObject)
  - O(1) time
  - pop() => remove(size()-1)
  - O(1) time
- use right end of list as top of stack

Derive From Chain



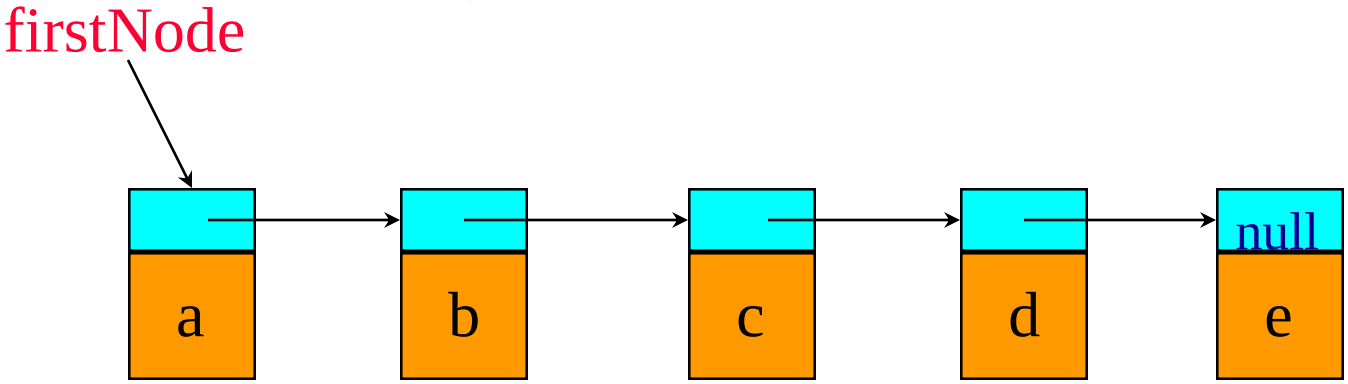
- stack top is either left end or right end of linear list
- empty() => isEmpty()
  - O(1) time

Derive From Chain



- when top is left end of linear list
  - peek() => get(0)
  - O(1) time
  - push(theObject) => add(0, theObject)
  - O(1) time
  - pop() => remove(0)
  - O(1) time

Derive From Chain



- when top is right end of linear list
  - peek() => get(size() - 1)
  - O(size) time
  - push(theObject) => add(size(), theObject)
  - O(size) time
  - pop() => remove(size()-1)
  - O(size) time
- use left end of list as top of stack

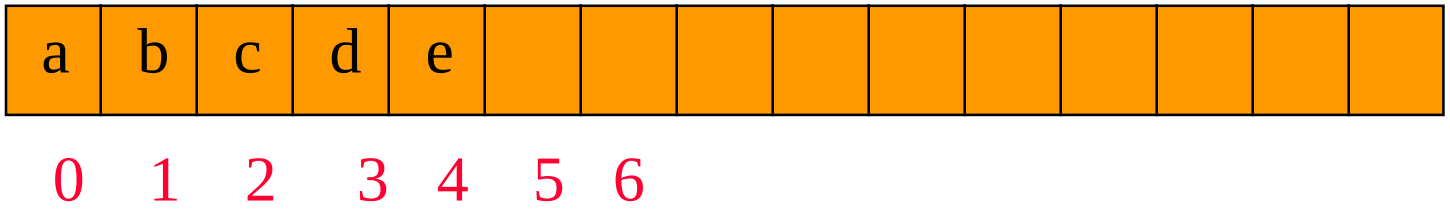
Derive From ArrayLinearList

```
package dataStructures;
import java.util.*;  // has stack exception

public class DerivedArrayStack
                extends ArrayLinearList
                implements Stack
{
   // constructors come here
   // Stack interface methods come here
}
```

Constructors

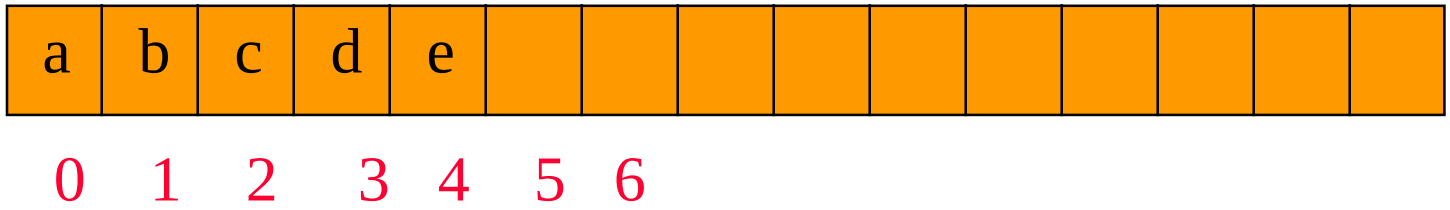
```
/** create a stack with the given initial
  * capacity */
public DerivedArrayStack(int initialCapacity)
  {super(initialCapacity);}

/** create a stack with initial capacity 10 */
public DerivedArrayStack()
  {this(10);}
```

empty() And peek()




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


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

push(theObject) And pop()




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


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

Evaluation

- 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.

Demerits

- All public methods of ArrayLinearList may be performed on a stack.
  - get(0) … get bottom element
  - remove(5)
  - add(3, x)
  - So we do not have a true stack implementation.
  - Must override undesired methods.

```
public Object get(int theIndex)
  {throw new UnsupportedOperationException();}
```

Change earlier use of get(i) to super.get(i).

Demerits

- Unecessary work is done by the code.
  - peek() verifies that the stack is not empty before get is invoked. The index check done by get is, therefore, not needed.
  - add(size(), theElement) does an index check and a for loop that is not entered. Neither is needed.
  - pop() verifies that the stack is not empty before remove is invoked. remove does an index check and a for loop that is not entered. Neither is needed.
  - So the derived code runs slower than necessary.

Evaluation

- Code developed from scratch will run faster but will take more time (cost) to develop.
- Tradeoff between software development cost and performance.
- Tradeoff between time to market and performance.
- Could develop easy code first and later refine it to improve performance.

A Faster pop()

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

vs.

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

Code From Scratch

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

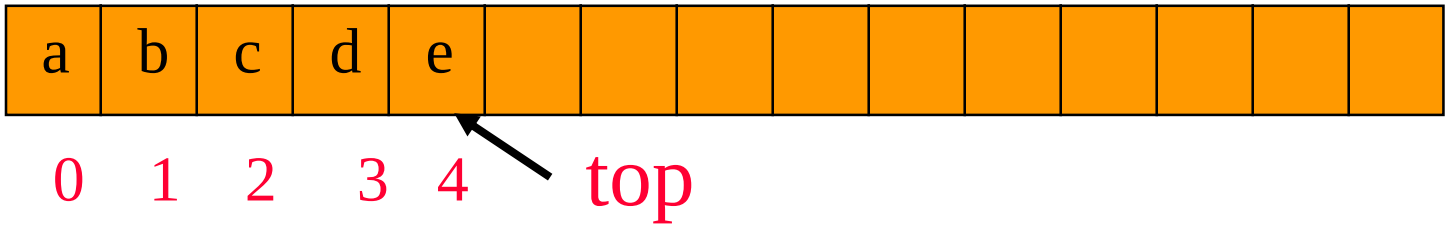
Code From Scratch

```
package dataStructures;
import java.util.EmptyStackException;
import utilities.*;  // ChangeArrayLength
public class ArrayStack implements Stack
{
   // data members
   int top;                // current top of stack
   Object [] stack;  // element array
   // constructors come here
   // Stack interface methods come here
}
```

Constructors

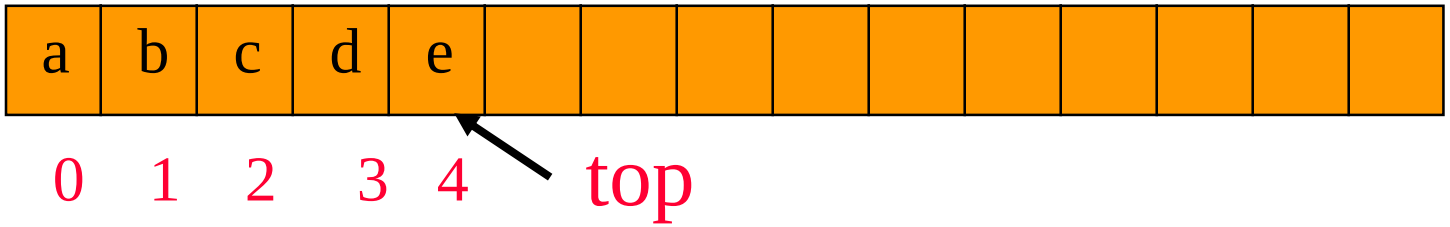
```
public ArrayStack(int initialCapacity)
{
  if (initialCapacity < 1)
     throw new IllegalArgumentException
      ("initialCapacity must be >= 1");
  stack = new Object [initialCapacity];
  top = -1;
}
public ArrayStack()
  {this(10);}
```

push(…)




```
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;
}
```

pop()




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

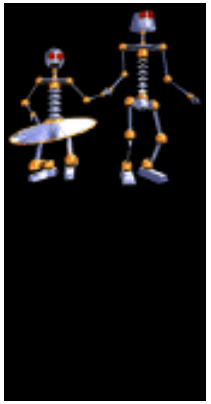
Linked Stack From Scratch

- See text.

java.util.Stack

- Derives from java.util.Vector.
- java.util.Vector is an array implementation of a linear list.
- Does not override all the methods of Vector class.

Performance



500,000 pop, push, and peek operations

Class	initial capacity 10	initial capacity 500,000
ArrayStack	0.44s	0.22s
DerivedArrayStack	0.60s	0.38s
DerivedArrayStackWithCatch	0.55s	0.33s
java.util.Stack	1.15s	-
DerivedLinkedStack	3.20s	3.20s
LinkedStack	2.96s	2.96s